Навчальне видання

БЄЛОВ Юрій Анатолійович
КАРНАУХ Тетяна Олександрівна
КОВАЛЬ Юрій Віталійович
СТАВРОВСЬКИЙ Андрій Борисович

Вступ до програмування мовою С++
Том 1. Організація обчислень

Друкується за авторською редакцією

Ю.А. Бєлов, Т.О. Карнаух, Ю.В. Коваль, А.Б. Ставровський

Вступ до програмування мовою С++
Том 1. Організація обчислень

Навчальний посібник

ДП "Інформаційно-аналітичне агентство"-2010

УДК 004.432.2
ББК 32.973.26-018.2.75
С76

Рецензенти:
академік НАНУ, д-р. фіз.-мат. наук, проф. О.А.Летичевський,
д-р фіз.-мат. наук, проф. М.М. Глибовець,
д-р фіз.-мат. наук, проф. М.С. Никитченко

Рекомендовано до друку вченою радою факультету кібернетики
Київського національного університету імені Тараса Шевченка
(протокол № 11 від 21 червня 2010 року)

Бєлов Ю. А., Карнаух Т. О., Коваль Ю. В, Ставровський А. Б.
Вступ до програмування мовою С++. Том 1. Організація обчислень. : К.: ДП "Інформаційно-аналітичне агентство"-, 2010. — 132 с.:іл. ISBN (укр.)

У цьому посібнику представлено першу частину базового курсу програмування. Матеріал розраховано на студентів математичних напрямів навчання та старшокласників шкіл з поглибленим вивченням інформатики. Посібник висвітлює основні поняття програмування, базові типи, операції та засоби організації обчислень у мові С++, а також елементи технології створення програм. Посібник містить численні приклади програм і окремих фрагментів коду, які наочно ілюструють викладений матеріал, та завдання для самостійного опрацювання. Посібник можна використовувати на практичних заняттях і в самостійній роботі студентів.

ББК 32.973.26-018.2.75

ISBN (укр.)

Передмова

Розробка програм — це складний інженерний процес, неможливий без відповідної технології. Автори цього посібника, багато років викладаючи програмування в Київському національному університеті імені Тараса Шевченка, переконані, що сучасний вступний курс програмування повинен не лише навчати конкретній мові, але й знайомити студентів з *технологією створення програм*. Саме знайомство з технологією й є характерною рисою посібника.

Посібник призначено, у першу чергу, студентам математичних напрямів навчання, але він може бути корисним і на факультативних заняттях у школі.

Цей посібник розпочинає серію книжок з базового курсу програмування. Він висвітлює основні поняття програмування, дані базових типів і операції мови C++, інструкції, підпрограми. Мова програмування C++ — це «жива мова», засоби якої розвиваються й удосконалюються, проте ґрунтовне вивчення всіх можливостей мови та численних бібліотечних засобів не може бути предметом вступного курсу програмування. Отже, найбільшу увагу в посібнику приділено основним мовним конструкціям та техніці їх застосування, а не можливостям бібліотек або нюансам компіляторів. Посібник також знайомить з елементами технології програмування: проектування та уточнення програми, вибір імен, коментування, розподіл обов'язків між частинами коду, використання підпрограм, створення програми з кількох файлів. Основи організації даних мають стати предметом наступної книжки серії.

Передбачається використання посібника на практичних заняттях і в самостійній роботі студентів. Він містить численні приклади програм, функцій та окремих фрагментів коду, які наочно ілюструють викладений матеріал. Дуже бажано, щоб студент, працюючи над текстом і вправами посібника, мав можливість відразу *створювати й запускати програми на комп'ютері*. Наведені в посібнику задачі є міні-проектами для самостійної розробки.

ЗМІСТ

Предметний покажчик

Бібліографічний список

Основна література

1. Дэвис С. C++ для «чайников». 6-е издание. – М.: Издательский Дом «Вильямс», 2010. – 336 с.

2. Либерти Дж., Брэдли Дж. Освой самостоятельно C++ за 21 день. 5-е издание. – М.: Издательский Дом «Вильямс», 2010. – 784 с.

3. Страуструп Б. программирование с примерами на C++: принципы и практика. – М.: Издательский Дом «Вильямс», 2010.

4. Шилдт Г. Полный справочник по C++. – М.: Издательский Дом «Вильямс», 2010. – 800 с.

Додаткова література

5. Мюссер Д., Дердж Ж., Сейни А. C++ и STL: справочное руководство. – М.: Издательский Дом «Вильямс», 2010. – 432 с.

6. Саттер Г., Александреску А. Стандарты программирования на C++. Серия «C++ In-Depth» – М.: Издательский Дом «Вильямс», 2008. – 224 с.

7. Седжвик Р. Алгоритмы на C++. – М.: Издательский Дом «Вильямс», 2010. – 1056 с.

8. Шилдт Г. C++: методики программирования Шилдта. – М.: Издательский Дом «Вильямс», 2008. – 480 с.

конання **cout<<scientific**, навпаки, встановлюється **ios_base::scientific** та скидається **ios_base::fixed**. Отже, використання маніпуляторів безпечніше ніж явне встановлення прапорців. Для керування форматом виведення у потік, окрім маніпуляторів **fixed** і **scientific**, можна користуватися маніпуляторами **showpos**, **showpoint**, **right**, **left**, **boolalpha**, які встановлюють однойменні прапорці (та скидають «протилежні»). Дія маніпуляторів **noshowpos** та **noboolalpha** протилежна дії маніпуляторів **showpos** та **boolalpha**.

Вправа Г.2. Що буде надруковано за виконання програми з такими варіантами вхідних даних:

а) **3 5**; б) **3 -4**; в) **3 0**?

```
#include <iostream>
using namespace std;
int main() {
   int i, j;
   cout<<"Enter 2 integers:"; cin>>i>>j;
   cout<<i<<showpos<<j<<'='<<noshowpos<<i+j<<endl;
   system("pause"); return 0;
}
```

Вправа Г.3. Що зміниться у відповіді до попередньої вправи, якщо замість інструкції

```
cout<<i<<showpos<<j<<'='<<noshowpos<<i+j<<endl;
```

записати таку?

```
cout<<i<<'+'<<j<<'='<<i+j<<endl;
```

Виведення дійсних значень. За *встановлених* прапорців **ios_base::fixed** чи **ios_base::scientific** функція **precision** (укр. *точність*) встановлює кількість цифр у дробовій частині дійсних значень. Так, після інструкції **cout.precision(2);** у дійсних значеннях після десяткової крапки виводяться дві цифри (за необхідності відбувається округлення або додаються нулі). За *скинутих* прапорців ця функція встановлює кількість цифр у всьому вихідному записі числа. Значенням виразу **cout.precision()** є поточна точність виведення дробових чисел.

Прапорець **ios_base::showpoint** встановлює обов'язкове виведення десяткової крапки та цифр після неї. Так, за виконання інструкцій

```
cout.setf(ios_base::showpoint);
cout<<1.00000001;
```

на екран буде виведено **1.00000**.

Прапорці **ios_base::fixed** та **ios_base::scientific** можна встановити або скинути в довільній комбінації. За стандартними налаштувань *обох їх скинуто*. У цій ситуації компілятор сам обирає спосіб виведення. Ось чому, якщо прапорці явно не встановлено, деякі дійсні числа виводяться з фіксованою крапкою, а деякі — в нормалізованій формі. Наприклад, за виконання **cout<<0.00000001;** виводиться **1e-008**, а за **cout<<1.00000001;** — **1**. Чому саме так? Якщо в програмі явно не вказано інше, діє **cout.precision(6)**, тому мають виводитися шість цифр після крапки. У числі **0.00000001** їх більше, тому обрано нормалізовану форму. У числі **1.00000** дробова частина нульова, тому виводиться лише ціла, оскільки прапорець **ios_base::showpoint** за стандартних налаштувань скинуто.

Зауважимо: документація до системи програмування найчастіше замовчує правила вибору того чи іншого формату за умов, коли обидва прапорці скинуто або встановлено одночасно.

Маніпулятори. Одночасне встановлення прапорців **ios_base::fixed** та **ios_base::scientific** не призводить до аварійного завершення програми: система виводитиме дійсні значення на «власний розсуд». Але ця ситуація може бути наслідком логічної помилки програміста (якщо встановлено один з цих прапорців, природно скинути інший). Суперечить здоровому глузду й одночасне встановлення деяких інших прапорців, наприклад, **ios_base::right** та **ios_base::left**.

Щоб уникнути описаних проблем, краще використовувати ***маніпулятори потоків***. Деякі з них, хоча й не всі, встановлюють та скидають прапорці (наприклад, маніпулятор **endl** не пов'язаний з прапорцями). Найчастіше, їх імена збігаються з іменами прапорців, які вони встановлюють. Наприклад, після інструкції **cout<<fixed;** дійсні числа виводяться у форматі з фіксованою крапкою. Насправді, ця інструкція встановлює прапорець **ios_base::fixed** та скидає **ios_base::scientific**. За ви-

Глава 1. Загальні поняття

1.1. Програми, дані, моделі, мови

Сучасна людина використовує комп'ютер для розв'язання різноманітних задач — від виконання простих арифметичних дій до моделювання атмосферних явищ та керування польотами в космос. Насправді все, що «вміє» комп'ютер — це ***виконувати програми***. Щоб комп'ютер розв'язав потрібну задачу, необхідно спочатку створити відповідну програму, а потім запустити її на виконання.

- ***Програма*** (*program*) — це опис тих дій, які має виконати комп'ютер, щоб розв'язати деяку задачу. ***Програмування*** — це діяльність, яка полягає у створенні програм. Мета програмування — забезпечити, щоб замість людини ту чи іншу роботу виконував комп'ютер.

Усі дії комп'ютера пов'язано з ***обробкою даних*** — числових, символьних, текстових тощо. Слідуючи В.М. Глушкову, ***дані*** (*data*) зображують (позначають) деякий ***зміст***, або ***інформацію***. Поняття змісту та інформації не мають чіткого означення або тлумачення. Будемо розуміти їх як знання, відомості про щось.

Приклади. Слово НЕБО записано чотирма буквами. Зміст, який позначено ними, не є чітким, але для кожної людини це щось велике над головою, блакитне вдень та чорне вночі. Уточнювати далі не будемо.

Запис **1001** позначає число — уявне поняття, яке виражає кількість. Ми сприймаємо ці дані як «тисяча й один» (предмет, рік тощо) або «тисяча й одна» (ніч, дрібниця тощо).

Коли батьки реєструють народжену дитину, вони отримують свідоцтво про народження. У ньому вказано ім'я та прізвище нової людини, дату й місце народження, імена батьків. Ці ***дані про людину*** позначають факт і деякі з обставин її появи на світ. ◂

Отже, зображення об'єктів реального світу за допомогою даних є основою будь-якої взаємодії, у тому числі й комп'ютера із «зовнішнім» світом.

Комп'ютер має розв'язувати задачі для людини. У цих задачах фігурують різноманітні об'єкти реального світу — картинки, фізичні явища, особи, технологічні процеси тощо. Щоб запрограмувати обробку даних, пов'язаних із цими об'єктами, треба спочатку ***зобразити об'єкти у вигляді даних***.

Зображення об'єкта, явища або процесу за допомогою даних про нього є його ***моделлю***. Узагалі, ***модель*** — це спрощене зображення об'єкта або явища, яке відображає його необхідні суттєві властивості. В обробці даних модель являє собою сукупність властивостей та співвідношень між ними, що відображають суттєві риси досліджуваного об'єкта, явища або процесу. Модель, як правило, містить опис не лише даних, але й їх обробки, яка відтворює реальні процеси, пов'язані з об'єктом.

Існує безліч різновидів моделей. Один і той самий об'єкт реального світу може мати декілька різних моделей, що виражають «свої» властивості, потрібні з різних точок зору на нього.

Приклади

1. Свідоцтво про народження або паспорт є дуже простою моделлю людини. Складнішою є, наприклад, медична картка, яка включає дані про фізичні та фізіологічні властивості людини, зміну їх у часі тощо. Зовсім інші дані про людину присутні в його табелі навчання або атестаті.

2. Щоб розв'язати рівняння вигляду $ax^2+bx+c = 0$, потрібні лише його коефіцієнти — числа a, b, c тобто трійка цих чисел (дані) зображує рівняння.

3. Кожне сучасне підприємство має програмні системи, які автоматизують облік кадрів, заробітної платні, фінансової та виробничої діяльності. Кожна з цих систем має «свої» дані, пов'язані з підприємством, які використовуються та обробляються тільки в ній. Ці дані складають частину відповідної (кадрової, фінансової тощо) моделі підприємства. Інша частина моделі описує, як присутні в моделі дані використовуються та обробляються. ◂

Самі по собі дані — це деяке позначення, тобто запис, що позначає властивості об'єкта. Але, щоб правильно зрозуміти запис (відновити позначений ним зміст), необхідно знати, що саме позначають окремі елементи цього запису.

- Система позначень деякого змісту називається ***мовою***. Мова включає елементарні позначення, правила утворення складніших позначень із простіших та правила, за якими зміст і позначення відповідають одне одному. Правила утворення позначень визначають ***синтаксис*** мови, а правила, які задають зміст позначень, визначають ***семантику***.

Для спілкування та мислення людина використовує так звані ***природні мови*** (наприклад, українська, російська, англійська тощо). Проте в науці та техніці природні мови неефективні або недостатні, тому використовуються й спеціальні ***штучні мови***. Їх призначено насамперед для обміну інформацією між користувачем та/або прикладними процесами. Одним з класів штучних мов є ***мови програмування***, призначені для запису програм.

1.2. Алгоритми та їхні властивості

Програму, призначену для виконання комп'ютером, можна розглядати як різновид такого більш загального поняття, як алгоритм.

- ***Алгоритм*** — це опис послідовності дій, які треба виконати, щоб розв'язати деяку задачу. Позначення дій в алгоритмі називаються ***командами*** або ***інструкціями*** (*statement*).

Як правило, в алгоритмі вказано деякі ***вхідні***, ***результатні*** (***вихідні***) та ***проміжні дані***, що не є ні вхідними, ні вихідними.

- Послідовність дій, яка виконується за алгоритмом, називається ***процесом***. Алгоритм, як правило, визначає не один, а деяку множину процесів.

Приклад. Розглянемо задачу «обчислити корені рівняння $ax^2+bx+c = 0$, заданого коефіцієнтами a, b, c (за умови $a \neq 0$)». Алгоритм розв'язання цієї задачі, тобто опис визначення коренів, може мати такий вигляд.

```
1. Прочитати коефіцієнти a, b, c.
2. Обчислити дискримінант d = b*b – 4*a*c.
```


Додаток Г. Параметри виведення та маніпулятори

Прапорці. Для параметрів виведення характерним є те, що їх може бути або задано, або ні, тобто їх, немов прапорці, можна або встановити, або скинути. Встановлює прапорець, що позначає певний параметр виведення, функція **`setf`** (її ім'я є скороченням *set flag* — встановити прапорець), наприклад, **`cout.setf(ios_base::right);`**. Дія встановленого прапорця поширюється на всі подальші виведення, доки його не буде скинуто за допомогою функції **`unsetf`**, наприклад, **`cout.unsetf(ios_base::right);`**. Наведемо кілька прапорців, які часто використовуються.

Прапорець	*Результат встановлення*
`ios_base::fixed`	дійсні числа виводяться у форматі з фіксованою крапкою
`ios_base::scientific`	дійсні числа виводяться в нормалізованій формі
`ios_base::showpos`	невід'ємні числа виводяться зі знаком **`+`** попереду
`ios_base::noshowpos`	невід'ємні числа виводяться без знака **`+`** попереду
`ios_base::showpoint`	дійсні значення завжди виводяться з десятковою крапкою
`ios_base::noshowpoint`	дійсні значення, що зображають цілі числа, виводяться без десяткової крапки
`ios_base::right`	вивідні символи притискаються до правого краю поля
`ios_base::left`	вивідні символи притискаються до лівого краю поля
`ios_base::boolalpha`	значення логічного типу виводяться як **`true`** (істина) та **`false`** (хибність); за скинутого прапорця значення логічного типу виводяться як **`1`** (істина) та **`0`** (хибність)

Виведення логічних значень. Логічні значення виводяться у два способи: як **`1`** і **`0`** або як **`true`** й **`false`**. За стандартних налаштувань встановлено прапорець **`ios_base::noboolalpha`**, який визначає перший спосіб виведення; для другого способу треба встановити **`ios_base::boolalpha`**.

Вправа Г.1. Що буде виведено на екран за виконання програми?

```
#include <iostream>
using namespace std;
int main() {
   cout<<true<<' '<<false<<endl;
   cout.setf(ios_base::boolalpha);
   cout<<true<<' '<<false<<endl;
   system("pause"); return 0;
}
```

Додаток В. Деякі бібліотечні математичні функції

Склад бібліотек у різних середовищах може бути різним, тому вичерпну інформацію про вміст бібліотек може дати лише довідка в конкретному середовищі або самі бібліотечні файли. Якщо іншого не зазначено, функції можуть мати як цілі, так і дійсні аргументи, а обчислюють дійсні значення.

Виклик	***Що обчислюється***	***Примітки***
`acos(x)`	arccos x	x від –1 до 1, значення від 0 до π
`asin(x)`	arcsin x	x від –1 до 1, значення від –π/2 до π/2
`atan(x)`	arctg x	значення від –π/2 до π/2
`atan2(x,y)`	arctg (x/y)	значення від –π/2 до π/2
`ceil(x)`	ціле, найближче до x згори	ceil(–1.9) = –1, ceil(1.1) = 2
`cos(x)`	cos x	x — кількість радіан
`cosh(x)`	cosh $x = (e^x+e^{-x})/2$	
`exp(x)`	e^x	
`fabs(x)`	$\|x\|$	
`floor(x)`	ціле, найближче до x знизу	floor(–1.1) = –2, floor(1.9) = 1
`fmod(x,y)`	дійсна остача від ділення x на y націло	fmod(4.7, –2.2) = 0.3, fmod(–4.7, –2.2) = –0.3
`log(x)`	ln x	$x > 0$
`log10(x)`	lg x	$x > 0$
`pow(x,y)`	x^y	$0^0 = 1$; за $x<0$ і нецілого значення y або $x=0$ і $y<0$ обчислення помилкове
`sin(x)`	sin x	x — кількість радіан
`sinh(x)`	sinh $x = (e^x-e^{-x})/2$	
`sqrt(x)`	$\sqrt{x}$	$x \ge 0$
`tan(x)`	tg x	x — кількість радіан
`tanh(x)`	tanh x = sinh x / cosh x	

3. Якщо *d* > 0, то

обчислити *x1* = $\frac{-b-\sqrt{d}}{2a}$, *x2* = $\frac{-b+\sqrt{d}}{2a}$ та написати ці числа;

інакше, якщо *d* = 0, то

обчислити *x* = $\frac{-b}{2a}$ й написати це число;

інакше написати «дійсних коренів немає».

У цьому алгоритмі вхідними даними є коефіцієнти a, b, c, проміжними — дискримінант d, вихідними — два корені (можливо, один) або текст «дійсних коренів немає».

За цим алгоритмом, залежно від конкретних вхідних даних, можливе виконання однієї з трьох таких послідовностей дій.

1. Прочитати коефіцієнти, обчислити d, перевірити, що $d > 0$ (і це так), обчислити $x1$, $x2$ й написати ці числа.

2. Прочитати коефіцієнти, обчислити d, перевірити, що $d > 0$ (і це не так), перевірити, що $d = 0$ (і це так), обчислити x і написати це число.

3. Прочитати коефіцієнти, обчислити d, перевірити, що $d > 0$ (і це не так), перевірити, що $d = 0$ (і це не так), і написати, що дійсних коренів немає.

◂

Вправа 1.1. Модифікуйте наведений алгоритм для розв'язання задачі обчислення коренів рівняння $ax^2+bx+c = 0$, заданого коефіцієнтами a, b, c, на випадок відсутності додаткової умови $a \ne 0$.

Алгоритми мають кілька загальних властивостей: зрозумілість, результативність, однозначність, дискретність, масовість та виконуваність. Розглянемо їх.

Зрозумілість. Для виконання алгоритму завжди потрібен ***виконавець***. Це може бути людина або деяка технічна система, зокрема, й комп'ютер. Наприклад, виконувати арифметичні дії, розв'язуючи квадратне рівняння (і не тільки), може людина. Але вона може перекласти цю роботу на комп'ютер, якщо створить відповідну програму й примусить комп'ютер її виконати. Так само, збирати прилади може спеціальна автоматична лінія, якщо виконує відповідну програму.

Зрозумілість алгоритму полягає в тому, що виконавець може правильно зрозуміти та виконати команди, записані в алгоритмі. А команди завжди записуються за допомогою певної системи позначень, тобто мови. Отже, виконавець повинен розуміти мову запису алгоритму.

Результативність. Виконання будь-якого алгоритму має приносити його виконавцю або іншій особі відчутні результати. Наприклад, «корені рівняння визначено», «прилад зібрано» тощо.

Однозначність. Алгоритм не повинен містити команд, зміст яких можна сприйняти неоднозначно. Наприклад, якби в алгоритмі розв'язання квадратного рівняння була команда «обчислити x або написати, що коренів немає», виконавець не знав би, що ж саме йому робити. Окрім того, після виконання кожної команди виконавець має точно знати, що йому робити далі.

Дискретність. Дискретність алгоритму полягає в тому, що він задає

послідовність дій, чітко відокремлених одна від одної. Отже, дії, задані командою, мають починатися лише після закінчення дій за попередньою командою. Окрім того, виконання кожної команди повинно займати обмежений проміжок часу.

Масовість. Конкретні об'єкти, до яких застосовуються дії під час виконання алгоритму, визначають конкретні задачі, які часто називаються ***екземплярами задачі***. Наприклад, конкретна трійка чисел 3, 10, 2 відповідає квадратному рівнянню, яке треба розв'язати. Масовість алгоритму полягає в тому, що він застосовний до різних наборів вхідних даних, тобто до різних екземплярів задач. Найчастіше алгоритм описує не один, а деяку ***множину процесів***, які відбуваються при розв'язанні всіх можливих екземплярів задачі, хоча існують і алгоритми, що задають тільки один процес.

Виконуваність та скінченність. Алгоритм має бути таким, щоб на кожному екземплярі задачі його можна було ***виконати до кінця*** (й отримати результат). Кожен процес, заданий алгоритмом, має бути ***скінченним*** і тривати скінченний час. Окрім того, процес ***не повинен обриватися*** без отримання результату.

Приклад. Ділення числа в стовпчик, тобто утворення десяткового дробу, може бути залежно від конкретного числа скінченним або нескінченним (для 5/4 дріб скінченний, для 5/3 — ні). Якщо за деяким алгоритмом потрібно ділити числа, і дільником є 0, то ділення неможливе, і незрозуміло, що робити далі. ◂

- Алгоритм має враховувати можливість нескінченного виконання дій або неможливість виконати наступну дію й містити команди, які забезпечують закінчення дій з деяким результатом ***за будь-яких умов***.

У деяких ситуаціях розглядають ***реальну виконуваність***, враховуючи вимоги до тривалості процесу, які висуваються в конкретних задачах. Існують такі задачі, для яких секунда — це занадто довго, а є й такі, що розв'язуються протягом кількох діб.

1.3. Модель комп'ютера

Незважаючи на прогрес технології, більшість сучасних комп'ютерів побудовано за тими самими принципами, що й обчислювальні машини 40-х років XX століття. В їх основі лежить так звана архітектура фон Неймана (за ім'ям видатного американського вченого, який першим сформулював головні засади архітектури електронних обчислювальних машин).

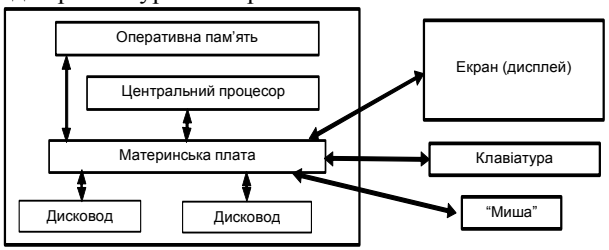


Рис. 1.1. Загальна схема комп'ютера

Додаток Б. Пріоритет операторів мови C++

У таблиці присутні не всі оператори мови C++, а лише ті, що зустрічаються в цій книжці. Оператори вказано за спаданням пріоритету згори донизу.

Операції		***Оператори***	***Зв'язування***
унарні	постфіксні	`++`, `--`	
	префіксні	`sizeof`, `++`, `--`, `!`, `-`, `+`	←
бінарні	мультиплікативні	`*`, `/`, `%`	
	адитивні	`+`, `-`	
	вставки	`<<`, `>>`	
	відношення	`<`, `>`, `<=`, `>=`	
	рівності	`==`, `!=`	
	логічне «і»	`&&`	
	логічне «або»	`||`	
	умовна	`?:`	←
	прості та складені присвоювання	`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `&&=`, `||=`	←
	слідування	`,`	

Знак ← відмічає операції з правобічним зв'язуванням; решта є лівобічними.

ний двійковий запис десяткового 0,1 є періодичним **0,0(0011)**. Будь-який початок цього запису позначає наближення до десяткового 0,1, наприклад,

$0{,}00011_2 = \frac{1}{16} + \frac{1}{32} = 0{,}09375$ (помилка — приблизно шість тисячних).

Утворимо 16-ковий запис десяткового дробу 0,8.

0,8·16 = 12,8, [12,8] = 12 (перша цифра **C**), {12,8} = 0,8.

Далі цифра **C** буде нескінченно повторюватися, тому **0,(C)** є точним 16-ковим записом десяткового 0,8.

◂

- Якщо V є скінченним десятковим дробом і основа P має обидва прості множники 2 та 5, то число V можна зобразити скінченним P-ковим записом. Інакше P-ковий запис може бути нескінченним; тоді скінченна кількість кроків дає *наближене зображення* числа V.

Отже, за дійсним числом V, $V < 1$, можна одержати R перших цифр його P-кового зображення, виконуючи такий алгоритм.

```
1. Спочатку зображенням є «0.».
2. Поки одержано менше за R дробових цифр,
      обчислити V·P, обчислити d як [V·P]
      (ціле число від 0 до P–1) та V як {V·P}.
      Записати значення d як P-кову цифру та
      дописати її до зображення праворуч.
```

Записи цілих та дробових чисел перетворюються з однієї системи числення в іншу *за різними алгоритмами*. Тому, перекладаючи запис числа, що має цілу та дробову частини, треба спочатку виділити ці частини й потім перекласти їх *окремо*.

Узагалі, щоб перекласти запис числа з довільної системи числення в іншу, необхідно виконати багато арифметичних дій у незвичній недесятковій системі. Інколи простіше скористатися десятковою системою як проміжною.

Вправа А.5. Записати P-кове зображення десяткового дробу Q, де:

а) $Q = 0{,}5$; $P = 2, 3, 8, 16$; в) $Q = 0{,}75$; $P = 3, 8, 16$;

б) $Q = 0{,}1$; $P = 3, 8, 16$; г) $Q = 0{,}2$; $P = 2, 3, 8, 16$.

Вправа А.6. Перетворити десятковий запис на двійковий, вісімковий та шістнадцятковий:

а) 94,341; б) 16382,55; в) 2948,77; г) 65537,33.

Контрольні запитання

А.1. Які системи числення ви знаєте?

А.2. Що таке позиційна система числення?

А.3. Як кодуються цифри в системах числення, основа яких більше десяти?

А.4. Розповісти алгоритм перетворення десяткового запису числа в систему числення, відмінну від десяткової.

А.5. Розповісти алгоритм перетворення запису числа з недесяткової системи числення в десяткову.

Загальну структуру комп'ютера наведено на рис. 1.1 (насправді вона набагато складніша). З основних елементів комп'ютера назвемо лише материнську ***плату***, центральний ***процесор***, оперативну ***пам'ять*** та зовнішні пристрої. На материнській платі розташовані: центральний процесор, оперативна пам'ять (ОП), центральна магістраль для зв'язку між усіма пристроями комп'ютера, гнізда для підключення інших плат керування зовнішніми пристроями та деякі інші елементи.

Зовнішні пристрої (пристрої уведення-виведення, або ПУВ) — це дисплей (монітор), клавіатура, маніпулятор «миша», дисководи та інші (сканер, модем тощо). Вони керують обробкою даних на зовнішніх носіях. ПУВ мають власні процесори, які можуть переносити дані з зовнішніх носіїв до оперативної пам'яті або навпаки.

Програма для комп'ютера являє собою послідовність команд, основний зміст яких — ***обробка даних***. Центральний процесор зчитує дані й команди програми з оперативної пам'яті та виконує їх. Команди задають зчитування даних з пам'яті, створення нових даних і запис їх у пам'ять. Є також команди, за якими дані надходять до зовнішніх пристроїв або зчитуються з них.

- Усі дані в комп'ютері являють собою послідовності **0** і **1**. Значення **0** і **1** відповідають двом стійким станам елемента пам'яті, що називається ***біт*** (bit, або binary digit — двійкова цифра). Вісім послідовних бітів утворюють ***байт***.
- Оперативна пам'ять являє собою послідовність байтів, в якій кожен байт має свій номер — ***адресу***. Деяке ***значення*** (число, символ тощо) в пам'яті, як правило, займає кілька сусідніх байтів і вказується адресою першого з них.

Абревіатурою для біту є «б», а для байту — «Б». Від байту походять такі одиниці інформації, як КБ («кілобайт», $2^{10} = 1024$ Б), МБ («мегабайт», $2^{20} = 1048576$ Б), ГБ («гігабайт», $2^{30} = 1073741824$ Б) і ТБ («терабайт», 1024 ГБ, тобто 2^{40} Б)[1]. Розмір оперативної пам'яті вимірюється, як правило, сотнями й тисячами МБ, і середній розмір пам'яті комп'ютерів щороку зростає.

Машинні команди, як і дані, також записуються послідовностями **0** і **1**. Спрощено можна сказати, що команда містить ***код машинної операції*** та ***адреси даних***, до яких застосовується операція. Система команд, які може виконувати процесор, називається ***машинною мовою***. Для людини машинні мови дуже незручні, вони вимагають глибоких знань про устрій вузлів комп'ютера та подробиці виконання програми. Цими мовами користуються розробники комп'ютерів та деякі інші спеціалісти.

[1] У метричній системі одиниць СІ префікс «кіло» відповідає 1000, а «К» позначає Кельвін (одиниця виміру температури). Тому позначення 1024 байтів як 1 КБ, хоча й є усталеним у програмістському середовищі, але не відповідає міжнародним стандартам. Зокрема, швидкість передавання даних 1 кілобіт/с розуміється як 1000 біт/с, а не як 1024 біт/с.

1.4. Зображення чисел

1.4.1. Двійковий запис чисел

Усі дані в комп'ютері зображуються, або кодуються, у вигляді послідовностей **0** та **1**. Зображення чисел за допомогою **0** та **1** називається ***двійковим***, тобто зображенням у двійковій системі числення.

Звична десяткова система має 10 цифр. У записі числа молодша цифра позначає кількість одиниць, наступні — кількість десятків, сотень і подальших степенів числа 10. Кожна кількість може бути від 0 до 9. У двійковій системі роль «десятки» відіграє число 2, а цифри **0** і **1** позначають можливу кількість 0 або 1. Цифри позначають кількість відповідних степенів числа 2. Отже, послідовні натуральні числа 0, 1, 2, 3, 4, 5 мають двійкові записи **0**, **1**, **10**, **11**, **100**, **101**. Зокрема, в записі числа вигляду 2^n (тобто 1, 2, 4, 8 тощо) старша цифра **1** і далі n нулів, а запис 2^n–1 — це n одиниць, що позначають суму $2^{n-1} + 2^{n-2} + \ldots + 2^1 + 2^0$.

Розглянемо додавання цілих чисел у двійковому записі. 0+0 = 0, 1+0 = 0+1 = 1, 1+1 = 10, 1+1+1 = 11. Звідси, додавання двійкових чисел у стовпчик, починаючи з молодших цифр (розрядів) дає значення 0 або 1 у розряді та перенос 0 або 1 у наступний розряд. Наприклад, за додавання 111+110 = 1101 з молодшого розряду (він перший праворуч) переноситься 0 у другий, з другого — 1 у третій, а перенос 1 з третього стає четвертим розрядом суми.

У десятковому дробі кожна цифра позначає кількість (від 0 до 9) від'ємних степенів числа 10. Аналогічно, у двійковому дробі цифри 0 або 1 позначають кількість від'ємних степенів числа 2, тобто 1/2, 1/4, 1/8 тощо. Так, запис **0,11** позначає $2^{-1} + 2^{-2} = 3/4$, а запис **0,0001** — число 2^{-4}, тобто 1/16, що має десятковий запис **0,0625**.

1.4.2. Принципи зображення чисел у комп'ютері

Стандартне зображення чисел у комп'ютері займає кілька послідовних байтів. Один байт може мати $2^8 = 256$ станів і зображати кожним з них одне з 256 значень, наприклад, ціле число від 0 до 255, або ціле число від –128 до 127 (яких теж 256), або щось інше. Два байти можуть мати $2^{16} = 65536$ станів і зображати ними 2^{16} різних значень тощо.

Цілі числа

Цілі числа зображуються в комп'ютері переважно у двох формах — без знаку та зі знаком. Ці форми називаються ***кодами***[2]. Числа ототожнюються

[2] Код цілого — це зображення цілого числа, яке набуває свого змісту (значення) за його інтерпретації. У різних мовах програмування інтерпретації можуть відрізнятися. Тут описано загальні принципи побудови байтів коду числа, але не зафіксовано порядок їх фізичного розташування. Проте у 4-байтовому коді можна спочатку записати два молодших, а потім два

Приклади. Утворимо вісімковий запис числа 202.

202 : 8 = 25 (остача **2** — молодша цифра),
25 : 8 = 3 (остача **1** — наступна цифра).
3 : 8 = 0 (остача **3** — остання, старша цифра).

Отже, отримано цифри **2**, **1**, **3** (від молодшої до старшої), тобто вісімковий запис **312**.

Запишемо 202 в 16-овій системі.

202 : 16 = 12 (остача **10** — значення молодшої цифри **A**),
12 : 16 = 0 (остача **12** — значення старшої цифри **C**),

Отже, десяткове 202 має 16-овий запис **CA**. ◂

Позначимо остачу від ділення T на P націло як T % P, частку — як T / P. Опишемо утворення P-кового запису числа N таким алгоритмом.

```
1. Спочатку частка T дорівнює N, а запис порожній.
2. Поки T > 0,
      обчислити остачу R як T % P та нову частку T як T / P
      (шляхом ділення в стовпчик),
      зобразити R у P-ковій системі числення
      як цифру C, відповідну R,
      та дописати цифру C до запису ліворуч.
```

Вправа А.4. Перетворити десятковий запис у двійкову, вісімкову та шістнадцяткову системи:

а) 94; б) 768; в) 16382; г) 65537.

Перетворення десяткового запису дробових чисел у недесятковий

Розглянемо, як за додатним дійсним числом $V < 1$ отримати цифри його P-кового запису (принаймні, кілька перших, оскільки запис може бути нескінченним). За способом запису,

$V = (0,x_{-1}\ldots x_{-k}\ldots)_P = x_{-1}\cdot P^{-1} + \ldots + x_{-k}\cdot P^{-k} + \ldots$

з невідомими значеннями цифр. Помножимо обидві частини рівності на P й отримаємо рівність

$V\cdot P = x_{-1} + x_{-2}\cdot P^{-1} + \ldots + x_{-k}\cdot P^{-k+1} \ldots = (x_{-1},x_{-2}\ldots x_{-k}\ldots)_P = x_{-1} + P^{-1}\cdot(0,x_{-2}\ldots x_{-k}\ldots)_P$.

Звідси $x_{-1} = [V\cdot P]$, а $P^{-1}\cdot(0,x_{-2}\ldots x_{-k}\ldots)_P = \{V\cdot P\}$, де $[V\cdot P]$ та $\{V\cdot P\}$ позначають цілу та дробову частини $V\cdot P$. Далі так само помножимо $\{V\cdot P\}$ на P, знову отримаємо цілу та дробову частину (ціла буде значенням x_{-2}), і так далі.

Приклади. Утворимо двійковий запис десяткового дробу 0,75.

0,75·2 = 1,5, [1,5] = **1** (перша цифра), {1,5} = 0,5;
0,5·2 = 1, [1] = **1** (наступна цифра), {1} = 0.

Усі подальші цифри будуть **0**, тому **0,11** є скінченним двійковим зображенням для десяткового 0,75.

Утворимо двійковий запис десяткового дробу 0,1.

0,1·2 = 0,2, [0,2] = **0** (перша цифра), {0,2} = 0,2;
0,2·2 = 0,4, [0,4] = **0** (наступна цифра), {0,4} = 0,4;
0,4·2 = 0,8, [0,8] = **0** (наступна цифра), {0,8} = 0,8;
0,8·2 = 1,6, [1,6] = **1** (наступна цифра), {1,6} = 0,6;
0,6·2 = 1,2, [1,2] = **1** (наступна цифра), {1,2} = 0,2.

Далі цифри **0**, **0**, **1**, **1** будуть повторюватися до нескінченності, тобто точ-

A, 11 — **B**, 12 — **C**, 13 — **D**, 14 — **E**, 15 — **F**. Тоді $\mathbf{2A}_{16} = 2\cdot16^1+10\cdot16^0 = 42$, $\mathbf{FF}_{16} = 15\cdot16^1+15\cdot16^0 = 255$, $\mathbf{0,C}_{16} = 12\cdot16^{-1} = ¾ = 0{,}75$. ◂

Перетворення запису числа з недесяткової системи в десяткову

У комп'ютері числа зображуються у двійковій системи числення. Проте людині зручніше вводити числа в комп'ютер та отримувати їх від нього як десяткові. Виникає ***задача перетворення запису числа*** з однієї системи в іншу (зокрема, з двійкової в десяткову й навпаки).

Запис $(x_n\ldots x_1x_0,x_{-1}\ldots x_{-k})_P$ з P-ковими цифрами x_i задає число $x_n\cdot P^n+\ldots+x_1\cdot P^1+x_0\cdot P^0+x_{-1}\cdot P^{-1}+\ldots+x_{-k}\cdot P^{-k}$. Цифри x_i у цій сумі позначають числа від 0 до P–1. Звідси, щоб знайти десяткове зображення числа, потрібно:

записати в десятковій системі число *P* та *P*-кові цифри числа;
обчислити суму добутків значень цифр
та відповідних степенів числа *P*.

Приклади

$10011_2 = 1\cdot2^4+0\cdot2^3+0\cdot2^2+1\cdot2^1+1\cdot2^0 = 19$;

$10{,}011_2 = 1\cdot2^1+0\cdot2^0+0\cdot2^{-1}+1\cdot2^{-2}+1\cdot2^{-3} = 2{,}375$;

$\mathbf{1BC}_{16} = 1\cdot16^2+11\cdot16^1+12\cdot16^0 = 444$;

$\mathbf{1B,C}_{16} = 1\cdot16^1+11\cdot16^0+12\cdot16^{-1} = 27{,}75$;

$12{,}2_3 = 1\cdot3^1+2\cdot3^0+2\cdot3^{-1} = 5{,}666\ldots$, тобто число записується нескінченним періодичним десятковим дробом. ◂

Вправа А.1. Перетворити шістнадцяткові записи чисел на десяткові:
а) F1; б) FF; в) 4AB; г) FFFE.

Вправа А.2. Перетворити вісімкові записи чисел на десяткові:
а) 377; б) 1777; в) 1232; г) 400.

Вправа А.3. За основою P та P-ковими записами дробів Q навести їх десяткове зображення:
а) $P = 2$; $Q = 0{,}0001$; $0{,}1111$;
б) $P = 3$; $Q = 0{,}22$; $0{,}(11)$;
в) $P = 16$; $Q = 0{,}1$; $0{,}F$; $0{,}8$; $0{,}(7)$;
г) $P = 2$; $Q = 0{,}001$; $0{,}1101$; $0{,}000001$;
д) $P = 3$; $Q = 0{,}(20)$; $0{,}(02)$; $0{,}11$;
е) $P = 16$; $Q = 0{,}1F$; $0{,}FE$; $0{,}81$; $0{,}3(F)$.

Перетворення десяткового запису цілих чисел у недесятковий

Розглянемо, як за натуральним числом N у десятковому записі отримати цифри P-кового зображення. Невідомі ані самі цифри, ані їх кількість. Проте відомо, що P-ковий запис задає число як суму добутків значень цифр на відповідні степені числа P, тобто за деякого невідомого n і невідомих цифр x_i справджується рівність

$N = (x_n\ldots x_1x_0)_P = x_n\cdot P^n+\ldots+x_1\cdot P^1+x_0\cdot P^0$.

Помітимо: всі доданки, окрім останнього, мають множник P. Тоді значенням молодшої цифри x_0 є остача від ділення N на основу P, а сума $T = x_n\cdot P^{n-1}+\ldots+x_1\cdot P^0 = (x_n\ldots x_1)_P$ дорівнює цілій частці від ділення N на P. Поділивши цю суму на P з остачею, знайдемо остачу x_1 і наступну частку, і так далі, поки на якомусь кроці частка від ділення не стане рівною 0. Остання остача й буде старшою цифрою x_n.

з їх зображеннями, хоча з погляду математики це є хибним.

Беззнаковий код займає кілька байтів (найчастіше, 1, 2, 4 або 8). Один байт зображує числа від 0 до 255 в їх двійковому записі, можливо, з незначущими нулями. А саме, байт **00000000** — число 0, байт **00000001** — число 1, байт **00000010** — число 2 тощо. Байти **1111 1110** та **1111 1111** зображають «останні» числа $254 = 2^8-2$ і $255 = 2^8-1$.

Додавання 1 до числа з кодом **1111 1111** дає послідовність бітів **1 0000 0000**, тобто 255+1 = 256. Перенос 1 зі старшого розряду збільшує кількість цифр і не дозволяє зобразити суму 256 в одному байті. Перенос 1 зі старшого розряду називається ***переповненням*** (*overflow*). Якщо ігнорувати переповнення, то про однобайтові числа можна сказати ось що:

— «наступним» за найбільшим числом є найменше;

— якщо сума двох однобайтових чисел більше 255, то їх однобайтова сума на 256 менше справжньої суми.

За більшої кількості байтів, якими зображено числа, ситуація аналогічна, лише діапазон чисел інший. Так, два байти зображують числа від 0 до 65535 (це $2^{16}-1$), чотири — від 0 до 4294967295 (це $2^{32}-1$) тощо.

Знакові коди також займають 1, 2, 4 або 8 байт. Старший біт зображує знак числа: **0** — «+», **1** — «–». ***Додатні*** числа зображуються так само, як і беззнакові, лише за рахунок знакового біта їх діапазон — від 0 до $2^{8N-1}-1$, де N — кількість байтів. Таке зображення називається ***прямим кодом***. Наприклад, число $2^{8N-1}-1$ має прямий код **011…1**. ***Від'ємні*** числа мають так званий ***додатковий код***. Код від'ємного числа A позначається $D(A)$ й утворюється так.

1. За прямим кодом числа |A| заміною всіх 0 на 1 і всіх 1 на 0 отримуємо обернений код *R*(A).
2. За *R*(A) як беззнаковим цілим числом обчислюємо код *D*(A) = *R*(A)+1.

Приклад. Побудуємо однобайтовий додатковий код числа –72. $72 = 64+8 = 2^6+2^3$.

Прямий код $\|A\|$	**0100 1000**
Обернений код $R(A)$	**1011 0111**
Додавання 1	**1**
Додатковий код $D(A)$	**1011 1110**

◂

«Відновити» за додатковим кодом від'ємне число можна у зворотному порядку.

1. *D*(A) розглядаємо як беззнакове ціле, обчислюємо *R*(A) = *D*(A)–1.
2. Код, обернений до *R*(A), є прямим кодом числа |A|.

Отже, однобайтові коди від **0000 0000** до **0111 1111** зображують числа

старших байти, а можна й навпаки. У 90-х роках XX ст. подібні розбіжності існували між MS Visual Basic і MS Visual C++.

від 0 до 127, а коди від **1000 0000** до **1111 1111** — числа від –128 до –1. Аналогічно беззнаковим кодам, додавання 1 до 127 дає перенос 1 у знаковий розряд, тобто, ігноруючи перенос, отримуємо код числа –128. Так само, якщо сума двох додатних однобайтових чисел більше 127, то в одному байті залишається зображення від'ємного числа, яке на 256 менше справжньої суми.

У двох байтах зі знаком зображуються числа від –32768 до 32767, тобто від -2^{15} до $2^{15}-1$, у чотирьох — від –2147483648 (це -2^{31}) до 2147483647 (це $2^{31}-1$) тощо.

Вправа 1.2. Указати двобайтовий додатковий код чисел:
а) –1, –8, –9, –32767, –32768;
б) –255, –256, –640, –641.

Вправа 1.3. Нехай при додаванні та відніманні чисел у знаковому зображенні перенос із старшого цифрового розряду стає змістом знакового розряду, а перенос із знакового розряду втрачається. Нехай *maxi* та *mini* позначають максимальне та мінімальне цілі числа. Чому дорівнює: а) *maxi*+1; б) *mini*–1?

Дійсні числа

Деякі ***особливості зображення*** дійсних чисел тут опущено, щоб не загромаджувати викладення. Для дійсних чисел найчастіше використовують 6, 8 або 10 байт, поділених на ***поля*** (послідовності бітів) <*знак*><*порядок*><*мантиса*>. Ці поля означають таке.

Поле <*знак*> має довжину 1 біт, його значення зображує ***знак***: **0** — «+», **1** — знак «–». Поле <*порядок*> у деякий спеціальний спосіб задає ***показник степеня*** числа 2 (він може бути як додатним, так і від'ємним). Поле <*мантиса*> означає ***дробову частину*** — невід'ємне число строго менше 1.

Поля зображають число в такий спосіб. Мантиса додається до 1, сума множиться на степінь числа 2, заданий порядком, і враховується знак. Отже, число є значенням виразу вигляду

$\pm(1 + \textit{дробова частина}) \cdot 2^{\textit{показник}}$.

Сума в дужках не менше 1 і строго менше 2, тому кажуть, що це зображення числа є нормалізованим.

- Дійсні числа в описаному зображенні утворюють ***обмежену підмножину*** множини раціональних чисел; зокрема, серед них є число, ***найменше за модулем і відмінне від 0***, а також число, ***найбільше за модулем***.
- Між двома послідовними числами в описаному зображенні є ***ненульова різниця***. Вона дуже мала, якщо числа близькі до 0, і досягає десятків тисяч, якщо числа близькі до найбільшого за модулем.

Що більше байтів займає зображення, то більше чисел воно дозволяє закодувати.

1.5. Мови програмування

Під ***мовами програмування*** розуміють штучні мови, призначені для запису програм. Першими мовами програмування були машинні мови. Але

Додаток А. Системи числення

Поняття системи числення

- ***Система числення*** — це система правил запису чисел.

Людина звикла до десяткової системи числення, яка з'явилася в давніх Вавилоні та Індії, потім стала відома арабам і завдяки їм прийшла в Європу. У цій системі є ***десять знаків*** — цифр, якими позначають числа від **0** до **9**. Більші числа позначають *послідовностями* цифр. Знаки в послідовності займають різні ***позиції*** (***розряди***): цифра праворуч позначає кількість одиниць, наступна — кількість десятків тощо. Цифра залежно від позиції має «різну вагу» (у записі **32** цифра **2** задає дві одиниці, а в **23** — два десятки). Цю систему запису чисел називають ***позиційною***.

- Кількість знаків у системі числення називається її ***основою***.

Люди користуються записом чисел з основою не тільки 10, але й, наприклад, 12, 60 або 5. У програмуванні можна зустріти запис чисел з основою 8 та 16.

У десятковій системі цифра праворуч у записі числа називається ***молодшою***, ліворуч — ***старшою***. Розряд одиниць називається нульовим, десятків — першим, сотень — другим тощо. Розряд вказує, на який *степінь числа 10 треба помножити цифру* в цьому розряді (одиниці — на 10^0, десятки — на 10^1, сотні — на 10^2 тощо). Отже, число записується як сума добутків цифр числа на відповідні степені десятки, наприклад, $5834 = 5\cdot10^3+8\cdot10^2+3\cdot10^1+4\cdot10^0$.

Якщо запис числа має дробову частину, то додаються *цифри, ділені на 10 у відповідних степенях*, наприклад, $0{,}234 = 2\cdot10^{-1}+3\cdot10^{-2}+4\cdot10^{-3}$.

Так само числа записуються й з іншими основами більше 1. Питання лише, які знаки є цифрами. Якщо основа не більше 10, беруть звичні десяткові цифри (стільки знаків, скільки треба). Проте за основи більше 10 потрібні додаткові знаки, щоб позначати десяткові числа 10, 11, …. У XX столітті для цього стали використовувати послідовні великі літери латинського алфавіту — *A*, *B*, *C* тощо.

Приклади

1. У двійковій системі числення лише дві цифри — **0** та **1**. Двійкові записи **10**, **11**, **100**, **101** позначають десяткові 2, 3, 4, 5. Запис **1101** позначає число $1\cdot2^3+1\cdot2^2+0\cdot2^1+1\cdot2^0 = 13$, запис **0,101** — число $1\cdot2^{-1}+0\cdot2^{-2}+1\cdot2^{-3} = 0{,}625$.

2. У системі числення з основою 8 для запису чисел є цифри **0**, **1**, **2**, **3**, **4**, **5**, **6**, **7**, «вага» кожного розряду числа є відповідним степенем вісімки. Отже, перші числа записуються як **0**, **1**, … **6**, **7**, а далі йдуть записи **10**, **11**, …, **17**, **20**, **21**, …, **77**, **100**, …. Вісімкове **10** — це звичне десяткове 8, **11** — звичне 9, **20** — десяткове 16, тобто двічі по 8. А вісімкові **100**, **200** та **400** — це десяткові 64, 128 та 256, тобто один, два та чотири рази по 8 у квадраті. Так само, $342_8 = 3\cdot8^2+4\cdot8^1+2\cdot8^0$, $0{,}342_8 = 3\cdot8^{-1}+4\cdot8^{-2}+2\cdot8^{-3}$, $34{,}2_8 = 3\cdot8^1+4\cdot8^0+2\cdot8^{-1}$ (про вісімковий запис свідчить маленька цифра 8 внизу).

3. У шістнадцятковій системі числа від 10 до 15 позначають так: 10 —

```
using namespace std;
using namespace binary;
```
Завдяки ним далі в програмі можна користуватися некваліфікованими іменами **cin**, **cout** тощо, а також **kilo** та **mega**.

Досить часто в програмах бувають потрібні лише окремі локальні імена, оголошені в деякому просторі імен. Замість використання всього простору можна оголосити лише потрібні імена кожне в такому вигляді.

```
using ім'я-простору :: локальне-ім'я;
```
Наприклад, у більшості програм цієї книжки замість директиви **using namespace std;** можна було б написати два зовнішніх оголошення **using std::cin; using std::cout;**. Або, якщо з наведених вище функцій перетворення в одному блоці потрібні **binary::kilo** та **metric::mega**, можна оголосити їх у цьому блоці.

```
using binary::kilo; using metric::mega;
```
Це дозволить звертатися до функцій, не вказуючи просторів імен, яким вони належать.

- Усі імена, оголошені поза межами просторів імен, незалежно від їх розташування по одиницях трансляції належать єдиному глобальному простору імен.
- Створюючи бібліотеку функцій, призначених для багаторазового використання в різних програмах, варто означити весь код у певному просторі імен. Це може значно полегшити його використання.

Вправа 9.4. З використанням розроблених у розділі бібліотечних файлів написати програму, яка запитує в користувача одиницю виміру (кілобайт або кілограм) та кількість і перетворює отриману кількість на байти чи грами відповідно. Створити для неї проект і виконувану програму.

Контрольні запитання

9.1. Чи може програма мовою C++ складатися з декількох файлів?

9.2. З файлів якого різновиду «збирає програму» препроцесор, а якого — компонувальник?

9.3. Для чого в мові C++ використовуються заголовні файли? Що варто в них записувати?

9.4. Що таке простір імен?

9.5. Що таке кваліфіковане ім'я?

9.6. У чому полягає суть оператора розв'язання контексту?

Задачі

9.1. Підготувати реферат на тему: «Робота з просторами імен у мові C++».

9.2. Підготувати реферат на тему: «Директива та оголошення **using** у мові C++».

для людини вони дуже незручні, тому в 1950-х роках було розпочато винайдення мов програмування, які за формою мали бути ближчими до мови математичних формул та людських мов, а також вільними від «машинних» подробиць. Водночас, ці мови мали бути такими, щоб записані ними програми можна було ***перекласти*** у відповідні машинні програми ***за допомогою самої машини***. І такі мови програмування невдовзі було розроблено.

Дії комп'ютера в цих мовах було представлено з високим рівнем абстракції, тому ці мови стали називати ***мовами високого рівня***.

Програмувати мовою високого рівня можна, якщо є ***транслятор*** — спеціальна програма, яка здійснює переклад «високорівневої» програми на машинну мову. Цей переклад називається ***трансляцією*** програми.

Протягом кількох десятиліть було створено тисячі мов програмування й трансляторів, і кількість їх постійно зростає.

1.6. Види діяльності зі створення програми

Процес створення програми в найзагальніших рисах вимагає кількох видів діяльності:

– аналіз задачі та уточнення її постановки;
– проектування програми;
– розробка програми (кодування);
– перевірка програми (тестування)
– передача замовнику (впровадження).

Аналіз задачі та уточнення її постановки. Спочатку замовник формулює задачу, яку йому необхідно розв'язати. Задачу, як правило, сформульовано недостатньо точно, навіть може бути, що замовник чітко не уявляє, що ж йому насправді потрібно ☺. Саме тому початок роботи над програмою починається з аналізу задачі та уточнення її постановки. Для цього потрібно заглибитися в предметну область замовника та в діалозі з ним уточнити деякі питання. Як правило, за результатами аналізу задачі будується спрощена модель предметної області, у термінах якої й уточнюється задача. Необхідно також з'ясувати вхідні дані майбутньої програми та результат її роботи над ними. Якщо розв'язання поставленої задачі можливе не за всіх вхідних даних, то слід ***визначити поведінку програми на некоректних вхідних даних***.

Приклад. Розглянемо таку задачу: написати програму ділення двох чисел. Вхідними даними є пара чисел: перше – ділене, друге – дільник. Проте дані з дільником 0 є некоректними. Отже, уточнення постановки полягає в тому, що для коректних вхідних даних програма повинна виводити частку від ділення першого числа на друге, а для некоректних — повідомлення про неможливість ділення. ◂

В умовах навчального процесу замовником виступає викладач. Проте умова задачі все рівно не обов'язково формулюється цілком точно та однозначно (див. попередній приклад).

- Якщо не провести аналіз задачі та на підставі проведеного аналізу не уточнити її постановку, то можна розв'язати не ту задачу. Аналіз задачі

дозволяє визначити, якими саме засобами (математичними та/або програмними) розв'язувати задачу, а уточнена постановка — що саме й в яких ситуаціях повинна робити програма.

Проектування програми. Між написанням твору та програми є певна аналогія. Писати твір ми починаємо з плану, який далі розкриваємо у творі. Так само й з програмою, спочатку ми формулюємо «план» програми у вигляді проекту, а потім втілюємо цей план у життя під час кодування — написання коду (тексту) програми.

Під час проектування з'ясовують структуру програми, її складові частини та взаємодію між ними. Тут поступово ***уточнюють дії*** з розв'язання задачі ***та їх опис***. З'ясовують і ***уточнюють дані***, потрібні для розв'язання задачі. Дуже часто в задачі можна виділити декілька ***підзадач*** і описати їх розв'язання окремо. Тоді й алгоритм складається зі зв'язаних та узгоджених між собою частин (***допоміжних алгоритмів***), які описують розв'язання підзадач.

Одночасно з проектуванням, як правило, відбувається подальше уточнення постановки задачі та моделі предметної області. На практиці, замовник може вирішити дещо змінити умову задачі (причому будь-коли!), і тоді доводиться знов уточнювати постановку та аналізувати задачу.

Результатом проектування є модель (проект) програми, яка далі поступово перетворюється на текст програми. Ця модель може бути записана, наприклад, у вигляді алгоритму з достатньо абстрактними кроками. Продовжимо наш приклад.

Приклад. За уточненою постановкою задачі спроектуємо програму ділення двох чисел у вигляді такого алгоритму.

```
1. Увести пару чисел.
2. Якщо ділення неможливе, повідомити про помилку та завершити
виконання.
3. Обчислити частку.
4. Надрукувати обчислене значення. ◄
```

- У багатьох програм є чотири основні частини: отримати вхідні дані, обробити некоректні вхідні дані, обробити коректні вхідні дані, вивести результати обробки. У наведеному прикладі цим частинам відповідають кроки 1–4.

Для складніших задач під час проектування потрібно визначати не лише загальний алгоритм роботи програми, але й необхідні структури даних та, можливо, програмні засоби для створення програми та її окремих частин.

Розробка програми (кодування). Коли дії та дані уточнено до вигляду, в якому їх можна виразити мовою програмування, починають розробку програми. Найчастіше програму записують ***мовою високого рівня*** (інколи окремі її частини різними мовами).

Розробляючи програму, можна припуститися помилок, які виявляються при трансляції або виконанні програми. Помилки необхідно виявляти й виправляти, тобто налагоджувати програму. ***Наладка програми*** полягає в тім, що її багаторазово запускають зі ***спеціально підібраними*** вхідними да-

Проілюструємо використання розроблених **cpp**-файлів простою програмою.

```
#include <iostream>
#include "prefix2.h"
#include "prefix10.h"
using namespace std;
int main(){
  cout<<metric::kilo(1)<<endl;  // 1000
  cout<<binary::kilo(1)<<endl;  // 1024
  system("pause"); return 0;
}
```

Практично всі наведені вище програми містять директиву **using**.

```
using namespace std;
```

Вона задає використання «стандартного» простору імен **std**. Насправді, бібліотечні засоби мови C++ для роботи з потоками, наприклад, ім'я **cout**, оголошено в просторі імен **std**. За межами цього простору стандартний потік виведення для компілятора ідентифікується не ім'ям **cout**, а кваліфікованим ім'ям **std::cout**. Отже, інструкція виведення повинна була б мати вигляд **std::cout<<"some text";**. Проте директива **using namespace std;** дозволяє далі позначати стандартний потік виведення просто локальним ім'ям: **cout<<"some text";**.

Директиви вигляду **using namespace *ім'я-простору;*** забезпечують ще один спосіб уникати конфліктів імен. Областю дії директиви **using namespace**, записаної поза блоками, є текст програми до кінця файлу, в якому її записано. Якщо ж її записано в блоці, то вона діє в межах цього блоку. Звідси, у ***різних***, ***не вкладених*** один в одного, блоках можна вказати використання різних просторів імен, і далі в кожному з блоків записувати імена з відповідного простору, не кваліфікуючи їх. Наприклад, наведену вище головну функцію можна написати так.

```
int main(){
  { using namespace metric;
    cout<<kilo(1)<<endl;    // 1000
  }
  { using namespace binary;
    cout<<kilo(1)<<endl;    // 1024
  }
  system("pause"); return 0;
}
```

Кожна директива використання простору у своїй області дії, по суті, додає локальні імена цього простору. Якщо між іменами різних просторів, потрібних у програмі, немає конфліктів, то директиви використання всіх цих просторів можна записати на початку програми й далі працювати лише з локальними іменами. Наприклад, якщо в програмі потрібні імена лише з просторів **std** та **binary**, то десь на її початку варто записати обидві директиви.

Ім'я, оголошене всередині простору імен, «отримує» додатковий ідентифікатор — ім'я простору. Наприклад, нехай записано простір імен **mySpace**.

```
namespace mySpace { int name; }
```

Ім'я **name**, оголошене всередині простору, для компілятора задається ***глобальним*** ідентифікатором **mySpace::name**; його перша частина вказує *контекст* (тут це простір імен), друга — *локальне* ім'я в межах контексту (простору імен). Оператор **::** називають ***оператором розв'язання контексту*** (*scope resolution operator*). Неформально, вираз **mySpace::name** можна розуміти як «ім'я **name**, оголошене в межах **mySpace**». Результатом застосування оператору розв'язання контексту є так зване ***кваліфіковане*** ім'я (*qualified*, у перекладі з англ. кваліфікований, специфікований, уточнений). У межах простору **mySpace** компілятор «розуміє» як локальне ім'я **name**, так і кваліфіковане **mySpace::name**, але за межами цього простору «діє» лише кваліфіковане ім'я **mySpace::name**. Іншими словами, областю дії оголошення імені **name** є простір **mySpace**, тому за його межами оголошення імені **name** діяти й не повинно.

- Завдяки тому, що область дії оголошення імені обмежено простором імен, однакові імена, оголошені в різних просторах, стають для компілятора різними. Саме це й дозволяє уникати конфліктів імен.

Повернемося до задачі про одиниці вимірювання. Уведемо два простори імен функцій: **metric** — для обчислень за метричною системою, **binary** — за двійковою системою. Функції простору **binary** оголосимо у файлі **prefix2.h**.

```
namespace binary {
 double kilo (double x);//*1024
 double mega (double x);//*(1024*1024)
}
```

У файлі **prefix2.cpp** запишемо означення функцій (також у просторі).

```
namespace binary {
 double kilo (double x)
  { return x*1024;}
 double mega (double x)
  { return x*1048576;}
}
```

Так само, у файлі **prefix10.h** оголосимо функції простору **metric**.

```
namespace metric{
 double kilo (double x);//*1000
 double mega (double x);//*(1000*1000)
}
```

У файлі **prefix10.cpp** запишемо їх означення.

```
namespace metric{
 double kilo (double x)
  { return x*1000;}
 double mega (double x)
  { return x*1000000;}
}
```


ними, які допомагають виявити та відшукати помилки, а потім і виправити їх.

Проект програми може залежати від можливостей мови програмування та обладнання. У довготривалих проектах трапляється, що під час розробки змінюються програмні й апаратні засоби, замовник уточнює задачу відповідно до нових можливостей свого обладнання, тому всі процеси починаються наново.

Перевірка програми (тестування). У реальних виробничих процесах програму розробляють програмісти, а перевіряють інші спеціалісти — ***тестувальники***. Тестування починається, коли програмісти впевнені, що програма (або деяка її частина) є правильною. Задачею тестування є лише встановлення факту відсутності або наявності помилок. Як правило, спочатку тестувальники все-таки виявляють помилки й цикл «розробка-тестування» повторюється. В умовах навчання ситуація аналогічна, тільки в ролі програміста виступає студент, а тестувальника — спочатку студент, а потім викладач.

Передача замовнику (впровадження). У найпростішому випадку робота програміста над програмою завершується передачею програми та супроводжувальної документації з її використання замовникові. Для більш серйозних програм може знадобитися не просто передати код замовникові, але й установити програму в замовника, зокрема у випадках, коли програма потребує спеціального налаштування параметрів. Інколи потрібно навчити персонал замовника працювати з програмою. Усі ці дії (передача, встановлення, навчання), що дозволяють замовнику користуватися програмою у своїх виробничих процесах, є частиною ***впровадження*** програми.

Серйозні програми потребують ***супроводження***. Розробник виправляє помилки, виявлені під час експлуатації програми, модернізує її, передає оновлені варіанти користувачам тощо.

У реальних великих проектах одночасно можуть виконуватися кілька видів роботи. Як тільки постановку задачі більш-менш зрозуміло, можна проектувати програму, обирати програмні засоби для реалізації, писати частини коду та тестувати їх, демонструвати готові частини замовникові, отримувати від нього уточнення та навчати його користуватися програмою. Як правило, програма та її можливості нарощуються поступово, шляхом багаторазових повернень до аналізу задачі та інших видів роботи.

1.7. Кроки роботи з програмою

Типова послідовність роботи з програмою включає такі кроки: набирання тексту, компіляція, компонування, завантаження та виконання або інтерпретація.

Набирання тексту. Текст програми мовою високого рівня (вхідний текст) найчастіше набирають за допомогою спеціальної програми (***текстового редактора***) і, як правило, записують на диск у вигляді ***вхідного файлу*** (рис. 1.2). Програма може складатися з кількох файлів — у великих програмах їх можуть бути десятки й сотні.

Компіляція. Компілятор — це програма, при виконанні якої читається вхідний текст і створюється його машинний еквівалент — ***об'єктний код*** (рис. 1.2).

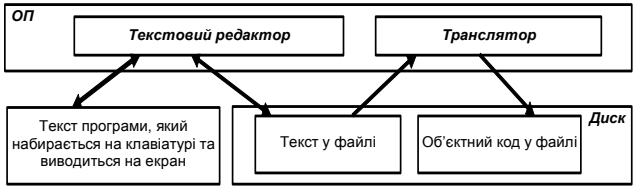


Рис. 1.2. Створення та компіляція тексту

Як правило, об'єктний код програми містить далеко не всі необхідні команди — програма може складатися з частин; деякі з них є стандартними.

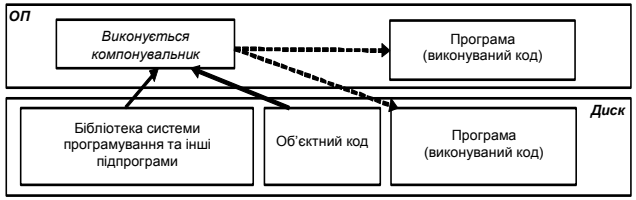


Рис. 1.3. Компонування та завантаження програми

Компонування. Об'єктний код обробляє ще одна програма — ***компонувальник***. Ця програма «збирає з частин» (компонує) виконуваний код, тобто машинну програму, й записує його або в оперативну пам'ять (***завантажує***), або на диск у вигляді файлу, ***готового до виконання*** (рис. 1.3). Такий файл можна завантажити пізніше.

Завантаження та виконання. Запис машинної програми в оперативну пам'ять називається ***завантаженням*** (рис. 1.4). Його здійснює спеціальна програма — ***завантажувач***, який може входити до складу компонувальника. Якщо завантаження здійснено успішно, починається процес виконання завантаженої програми.

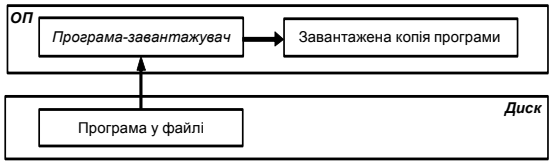


Рис. 1.4. Завантаження програми

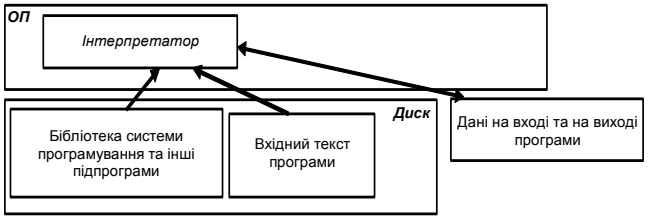


Рис. 1.5. Інтерпретація високорівневої програми

Інтерпретація. Це такий спосіб обробки високорівневої програми, за яким машинна програма не створюється (рис. 1.5). Вхідна високорівнева

лянка пам'яті. Наявність чи відсутність ініціалізації змінної тут ні на що не впливає: все рівно це означення.

Коли програма складається з декількох файлів, може виникнути необхідність у глобальних змінних, що використовуються в різних одиницях трансляції. Проте змінна не може мати більше одного означення! Тому її необхідно ***означати тільки в одній одиниці трансляції***, а ***в усіх інших — лише оголошувати***.

Оголошення глобальних змінних у мові C++ має такий вигляд.

```
extern ім'я-типу ім'я-змінної;
```

Так, означенням `int a;` та `int b=3;` відповідають оголошення `extern int a;` та `extern int b=3;`. За правилами мови C++, якщо глобальну змінну означено зі специфікатором `const`, наприклад: `const int OK=0;`, то відповідне оголошення теж повинно мати цей специфікатор й те саме ініціалізуюче значення: `extern const int OK=0;`.

Оголошення глобальних змінних, як правило, записують у заголовному файлі.

- За оголошенням змінної, яке не є означенням, окрема ділянка пам'яті під змінну не виділяється.

Вправа 9.3. Модифікувати програму **`prog021.cpp`** зі с. 106, виділивши функцію **`get`** в окрему одиницю трансляції, та створити для неї проект і виконувану програму в спосіб, описаний у розділі.

9.3. Поняття простору імен

У цьому розділі викладено лише початкові відомості про простори імен у мові C++. Детальніше дивіться книги, вказані в бібліографічному списку.

Почнемо з прикладу. Відомо, що в метричній системі префікси «кіло» та «мега» задають множення на 10^3 та 10^6 відповідно, але у випадку вимірювання інформації програмісти часто використовують ці префікси для множення на 2^{10} та 2^{20} відповідно. Нехай є **`cpp`**-файли (та відповідні **`h`**-файли), кожен з яких містить функції **`kilo`** та **`mega`** для перетворення одиниць вимірювання «кіло» та «мега» в прості одиниці. Відмінність між цими двома файлами лише в тому, що в першому функції виконують множення на відповідну степінь 10, а в другому — на степінь 2. Припустимо, що в програмі необхідні функції з обох файлів. Спроба включити обидва файли в проект буде неуспішною, оскільки функція **`kilo`** має в проекті два означення (у таких ситуаціях кажуть, що виник ***конфлікт імен***). Уникнути цих проблем можна, якщо скористатися ***простором імен*** (*namespace*).

Простір імен — це іменована частина програми, яка містить оголошення, означення та інші елементи мови. Простір імен має такий синтаксис.

```
namespace ім'я-простору {
вміст простору імен, тобто оголошення, означення тощо
}
```

points.h.) У цей **h**-файл записують оголошення імен функцій та змінних, які мають використовуватися за межами **cpp**-файлу. При цьому деякі допоміжні функції або змінні в заголовному файлі можуть бути не оголошені.

Тексти у файлах **points.cpp** та **prog022.cpp** є ***одиницями трансляції*** (*translation unit*), або ***програмними одиницями***. Кожна одиниця містить послідовність функцій, директив препроцесора та інструкцій оголошень імен. Як правило, С++-програма складається з кількох одиниць трансляції.

Кожну одиницю трансляції можна скомпілювати окремо. У нашому прикладі результатом будуть об'єктні файли з розширенням «**.obj**» (або «**.o**»). Далі за допомогою компонувальника з цих файлів можна зібрати виконуваний код програми. Проте зручніше скористатися засобами системи програмування, призначеними саме для створення програм з багатьох вхідних файлів.

У системі програмування Microsoft Visual C++ засобами меню створимо новий ***проект***. Типом проекту виберемо **Win32 Console Application** — консольна програма на платформі **Win32** (узагалі, у меню майстра створення проекту можуть указуватися й інші платформи). За допомогою меню додамо до проекту вхідні файли (**Source Files**) **points.cpp** та **prog022.cpp**. Далі залишається побудувати виконуваний код (він матиме ім'я проекту з розширенням «**.exe**»).

Список файлів проекту ведеться системою програмування в спеціальному файлі (його ім'я й зміст залежать від конкретної системи). Це значно полегшує розробку програми з кількох файлів, особливо, якщо їх багато.

Вправа 9.1. Модифікувати проект прикладу: додати до файлів **points.cpp** та **points.h** функцію **dist** обчислення відстані між двома точками (див. с. 67) та переробити головну функцію так, щоб програма запитувала в користувача координати двох точок площини та виводила відстань між ними.

Вправа 9.2. Створити проект і виконувану програму для задачі 6.1 у спосіб, описаний у розділі.

9.2. Означення та оголошення імен змінних

У підрозділі 5.2.2 на с. 68 розглядалися означення та оголошення функцій. Нагадаємо: ***оголошення*** повідомляє про наявність імені та його типу, а ***означення*** ставить у відповідність імені певну ділянку пам'яті (змінна зберігає значення певного типу, функція — опис виконуваних дій). За правилами мови, ***кожне оголошене ім'я має бути означеним*** (у деякій програмній одиниці проекту). Розглянемо означення та оголошення змінних детальніше.

Інструкція **int x;** не тільки оголошує, але й означає ім'я **x**, адже за цією інструкцію утворюється змінна **x**, тобто для імені **x** призначається ді-

програма обробляється спеціальною програмою — ***інтерпретатором***. При цьому дії вхідної програми, які вона задає, відразу виконуються. Як правило, інтерпретація вхідної програми відбувається повільніше, ніж виконання відповідної машинної програми.

Інтерпретація програми використовується в такому інструменті, як ***налагоджувач***. Він забезпечує інтерпретацію вхідної програми невеликими порціями (кроками) і дає можливість побачити результати виконання після кожного кроку. Це полегшує виявлення помилок у вхідній програмі.

Інколи компіляцію та інтерпретацію узагальнюють словом ***трансляція***, називаючи трансляторами усі види програм перетворення вхідних текстів до машинного чи проміжного вигляду.

- Описані засоби (текстовий редактор, компілятор та/або інтерпретатор, компонувальник, завантажувач і налагоджувач), як правило, утворюють ***систему програмування***, або ***інтегроване середовище***. Крім них, до складу цієї системи входить ***бібліотека стандартних підпрограм***, які можна використовувати під час створення програми.
- Що краще розробник програми володіє технологіями, інструментом та бібліотеками, то його робота ефективніше.

Контрольні запитання

1.1. Що таке модель? Назвіть приклади моделей.
1.2. Що таке мова?
1.3. Що таке синтаксис?
1.4. Що таке семантика?
1.5. Що таке мова програмування?
1.6. Що таке алгоритм?
1.7. Чим алгоритм відрізняється від процесу розв'язання задачі?
1.8. Назвіть властивості алгоритму.
1.9. Які функціональні блоки має комп'ютер? Яке їх призначення?
1.10. Що таке біт і байт? Скільки станів вони можуть мати?
1.11. Чому числа в комп'ютері зображуються з використанням двійкової системи числення?
1.12. Чим мови програмування високого рівня відрізняються від машинних мов?
1.13. Що таке трансляція і що таке транслятор?
1.14. Назвіть і опишіть основні види роботи зі створення програми.
1.15. Чому, створюючи програму, не можна відразу починати писати її текст?
1.16. У чому полягає наладка програми?
1.17. Чим відрізняється компіляція від інтерпретації?
1.18. Які засоби входять до складу інтегрованого середовища програмування?

Глава 2. Елементи мови C++

2.1. З історії створення мови C++

Історія мови C++ («Сі-плюс-плюс») почалася з мови C («Сі»). Компанія Bell Laboratories на початку 1970-х років створила мову C як інструмент розробки операційної системи UNIX. Спочатку Мартін Річардс розробив мову BCPL, потім на її основі Кен Томпсон — мову B, і нарешті, Денніс Річі — мову C. Завдяки своїй лаконічності, виразній потужності та надійним компіляторам ця мова програмування дуже швидко стала однією з найбільш популярних і поширених. Проте з часом розміри та вимоги до програм зростали, тому можливості мови C та інших мов, подібних їй, поступово вичерпувалися.

Кризу в програмуванні, яка намітилася в кінці 1970-х років, було подолано за допомогою, головним чином, ***об'єктно-орієнтованого програмування*** (***ООП***). Однією з об'єктно-орієнтованих мов була мова C++, яку на базі мови C розробив Бйорн Страуструп у компанії Bell Laboratories у кінці 1970-х років.

У наступні два десятиліття мову C++ було збагачено й стандартизовано. Її втілено в кількох сучасних системах програмування, зокрема в Microsoft Visual C++ (існує кілька версій цього дуже потужного засобу програмування), на яку й орієнтуються автори в прикладах цієї книги.[3]

Нащадками мови C++ є мови Java («Джава») та C# («Сі-шарп» або «Сі-дієз»), спеціалізовані для програмування в сучасних комп'ютерних мережах. Ці мови за своєю структурою схожі на C++, тому, володіючи C++, неважко перейти на Java або C#, і навпаки.

2.2. Перша програма — «Hello, World!»

Програма мовою C++ записується у файл з розширенням **.cpp**, наприклад, **prog001.cpp**.

Приклад. У книжках з мов програмування дуже часто першою наводять програму, яка «вітає світ», тобто виводить на екран рядок-привітання.

```
#include <iostream>
using namespace std;
int main()
{
   cout<<"Hello, World!";
   return 0;
}
```

prog001.cpp

У першому рядку записано ***директиву препроцесора***. ***Препроцесор*** —

[3] Є й інші системи програмування на базі C++, наприклад, Dev-C++, які мають відмінності в інтерфейсі та синтаксичних деталях. Для навчання автори рекомендують Microsoft Visual C++, оскільки, на їх думку, саме вона має найкращу довідкову систему.


```
#include <iostream>
using namespace std;
void inputPoint(double &x, double &y){
  ...
}
void writePoint(double x, double y){
  ...
}
```

points.cpp

Проте після перенесення імена **inputPoint** і **writePoint** у залишку програми не оголошено, тому використовувати їх не можна. Виправимо це. Запишемо прототипи перенесених функцій в окремий заголовний файл (з ім'ям **points.h**).

```
void inputPoint(double &x, double &y);
void writePoint(double x, double y);
```

points.h

Файл **points.cpp** залишимо без змін. У файлі з головною функцією додатково вкажемо включення файлу **points.h** та збережемо його як **prog022.cpp**.

```
#include <iostream>
#include "points.h"
using namespace std;
int main(){
...
}
```

Отже, маємо три файли. Файл **points.h** містить заголовки функцій роботи з координатами точок, файл **points.cpp** — самі функції та необхідні «стандартні» директиви, а файл **prog022.cpp** — включення **points.h** та головну функцію.

Файл зі складу системи програмування в директиві **include** вказується в кутових дужках, як **<iostream>**, а файл, створений програмістом — у лапках, як **"points.h"**. Кутові дужки кажуть, що препроцесор має шукати заголовний файл, починаючи з підкаталогу **include** каталогу з системою програмування та його підкаталогів, лапки — з каталогу з поточним **cpp**-файлом[15]. Отже, файли **points.h** та **prog022.cpp** краще розмістити в одному й тому самому каталозі.

- Як правило в заголовних файлах записують оголошення імен функцій, змінних та інших елементів програми.
- Зазвичай для кожного **cpp**-файлу, що містить функції, потрібні в різних частинах програми (або в різних програмах), створюють окремий заголовний **h**-файл. (У прикладі за файлом **points.cpp** створено

[15] Або в додаткових каталогах, які зазначає програміст (висвітлення цих можливостей системи програмування виходить за межі книги).

Глава 9. Структурна організація програми

Сучасні програми містять тисячі рядків коду та використовують різноманітні бібліотеки. Вони настільки великі, що під час написання просто необхідно розбивати їх на окремі частини, які можна було б окремо розробляти, компілювати, тестувати, а потім використовувати при побудові програми. Природно, що ці частини мають бути записані в різних файлах, які компілюються окремо один від одного (тобто роздільно), а потім компонуються в єдину програму.

9.1. Програма в кількох файлах

У кожній програмі цієї книги зустрічалася директива препроцесора **`#include <iostream>`**, яка підключає стандартний бібліотечний файл **`iostream`** з засобами роботи з стандартними потоками та багато чим іншим. Виникає природне питання, а чи не можна створити власний файл з означеннями певних функцій та так само використовувати його в багатьох різних програмах. Мова програмування C++ має такі засоби!

Спочатку розглянемо, що саме містить заголовний файл **`iostream`** і як це все працює. За директивою **`#include <iostream>`** препроцесор додає до тексту програми вміст файлу **`iostream`** (він, як правило, знаходиться в підкаталозі **`include`** каталогу з системою програмування). Вміст файлу **`iostream`** — це *оголошення* імен змінних, бібліотечних функцій та інших елементів програми, необхідних для використання потоків. Ці оголошення дозволяють компілятору перетворити інструкції програми до машинних кодів, тобто побудувати *об'єктний код*. Проте, щоб створити *виконуваний код*, до програми необхідно додати, серед іншого, коди бібліотечних функцій, які викликаються в програмі. І ці коди містяться у відповідних бібліотечних файлах, що також входять до складу системи програмування. Створити виконуваний код — задача компонувальника. Отже, як для компіляції програми насправді потрібно декілька вхідних файлів, так і для компонування — декілька об'єктних. І цими файлами можуть бути не лише файл з головною функцією та файли зі складу системи програмування, але й інші файли, створені програмістом.

Проілюструємо створення програми, складеної з кількох вхідних файлів, на простому прикладі. У розділі 5.3 (див. с. 70) наведено програму з функціями введення та виведення координат точки площини, які викликаються в головній функції. Ці функції роботи з точками можуть знадобитися в інших програмах, але дублювати їх у кожну нову програму зовсім не обов'язково.

Розглянемо спосіб, яким можна уникнути дублювання функцій у програмах. ***Перенесемо*** функції введення та виведення координат точки площини (а також додамо необхідні для їх успішної компіляції директиви) в новий файл з ім'ям **`points.cpp`**.

це складова частина компілятора, яка проводить попередню обробку програми. Директиви записуються в окремих рядках і починаються символом **`#`**. Слово **`include`** (включити) означає, що препроцесор перед компіляцією програми має включити в неї вміст спеціального файлу зі складу системи програмування, ім'я якого **`iostream`** записано в кутових дужках. У цьому файлі оголошено засоби введення та виведення (ім'я **`cout`**, операцію **`<<`** і багато інших). Без включення цього файлу ім'я **`cout`** буде невизначеним, і компілятор повідомить про цю помилку.

- Файл **`iostream`** є одним з багатьох ***заголовних*** (*header*) ***файлів*** (або **`h`**-файлів), що входять до складу системи програмування, тобто є стандартними. У директивах імена стандартних заголовних файлів записуються в кутових дужках. Багато стандартних заголовних файлів має порожнє розширення, для решти традиційно використовують розширення **`h`**.

У другому рядку розташовано інструкцію компілятору «використати простір імен **`std`**». Не пояснюючи значення слів «простір імен» (детальніше див. розділ 9.3), скажемо лише, що простір імен **`std`** є стандартним. У сучасних системах програмування мовою C++ у ньому описано всі бібліотечні засоби «останнього покоління». Завдяки наведеній інструкції спрощується доступ до бібліотечних засобів (один з них, з ім'ям **`cout`**, використовується в програмі). Проте компілятори попереднього покоління цієї інструкції «не розуміють», і для них писати її не можна.

Наведена програма складається з однієї функції з ім'ям **`main`**. Слова **`int main()`** у третьому рядку — це ***заголовок функції***. Дужки **`()`** після імені **`main`** вказують, що це ім'я саме функції. Ім'я **`int`** перед іменем функції є скороченням слова integer і означає, що функція має повертати ціле значення.

- Функція з ім'ям **`main`** називається ***головною функцією***. Вона має бути в кожній програмі, адже саме з неї починається виконання програми і, як правило, нею й закінчується. Ім'я **`main`** не є зарезервованим, але використовувати його з іншим призначенням не слід.

Вміст рядків 4-7 утворює ***тіло*** функції, яке починається символом **`{`** і закінчується **`}`**. У тілі функції задано дії у вигляді ***послідовності інструкцій***. Інструкція в п'ятому рядку задає виведення на екран повідомлення **`Hello, World!`**. Воно з'являється у вікні програми, яке має відкритися на екрані за її виконання та зникнути після її завершення. Текстове повідомлення, яке потрібно вивести на екран, записується в лапках **`""`**.

Інструкція в шостому рядку каже, що функція повинна повернути значення 0 (нуль)[4].

[4] Повернути можна й інше ціле значення. Значення, що повертається

Отже, запустимо цю програму на виконання. Чи встигнемо ми щось побачити? Відповідь залежить від системи програмування, що використовується. Деякі, але не всі, системи програмування дозволяють переглядати вікно програми після її завершення. З іншого боку, виконання програми можна затримати за допомогою бібліотечної функції **system**.

```
#include <iostream>
using namespace std;
int main () {
   cout<<"Hello, World!";
   cout<<endl;
   system("pause");
   return 0;
}
```

prog002.cpp

За інструкцією **system("pause");** виконання програми призупиняється й на екрані з'являється повідомлення, що треба натиснути будь-яку клавішу. Після натискання програма завершується. Завдяки попередній інструкції **cout<<endl;** повідомлення виводиться в новому рядку. Якби цієї інструкції не було, повідомлення з'являлося б відразу після слів **Hello, World!**. ◂

- Наведена програма складається з двох директив та головної функції. Узагалі, програма може містити багато функцій, директив та деяких інших елементів.

Вправа 2.1. Що буде виведено за виконання таких трьох програм? Поясніть, у чому різниця.

```
#include <iostream>
using namespace std;
int main() {
 cout<<"Microsoft";
 cout<<"Visual";
 cout<<"Studio";
 cout<<endl;
 system("pause");
 return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
 cout<<"Microsoft ";
 cout<<"Visual ";
 cout<<"Studio";
 cout<<endl;
 system("pause");
 return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
 cout<<"Microsoft";
 cout<<endl;
 cout<<"Visual";
 cout<<endl;
 cout<<"Studio";
 cout<<endl;
 system("pause");
 return 0;
}
```

2.3. Алфавіт і словник мови C++

Будь-яка мова має ***алфавіт*** — скінченну множину символів, дозволених для використання. Алфавіт для написання програм мовою C++ містить:

— ***літери*** (***букви***) — великі й малі латинські **A**, **B**, … **Z**, **a**, **b**, … **z**, а та-

функцією **main**, передається операційній системі як результат виконаної програми. За традицією 0 є ознакою вдалого виконання програми.

д) перевіряє послідовність на неспадання;
е) обчислює, скільки разів знаки чисел змінюються на протилежні;
ж) підраховує суму $a_1 \cdot a_2 + a_2 \cdot a_3 + \ldots + a_{n-1} \cdot a_n + a_n \cdot a_1$.

8.7. Написати програму, яка за введеним цілим числом та основою системи числення p:
а) обчислює середнє арифметичне значень цифр його p-кового запису;
б) обчислює середнє геометричне значень цифр його p-кового запису;
в) визначає найменшу цифру його p-кового запису та виводить її значення в десятковому запису;
г) перевіряє послідовність цифр його p-кового запису на монотонність;
д) обчислює скільки разів у p-ковому записі повторюються сусідні цифри.

8.8. **Планета Форт Нокс**. На планеті Форт-Нокс лежать золоті зливки у формі прямокутного паралелепіпеда. Космічний корабель має вивезти частину зливків у своєму трюмі, входом до якого є прямокутне вікно. До вікна зливки подає конвеєр. Зливок лежить на конвеєрі своєю нижньою гранню, а його бічні грані паралельні напряму руху. Конвеєр здатен виконувати операції трьох видів: повернути зливок так, щоб його верхня грань стала бічною, повернути так, щоб верхня грань стала передньою, та скинути паралелепіпед з конвеєра. Якщо зливок після будь-яких поворотів не може пройти віконце або його об'єм перевищує половину залишку об'єму трюму, то конвеєр скидає його. Написати програму, що моделює завантаження корабля та виводить протокол роботи конвеєра: розміри a, b, c кожного паралелепіпеда, послідовність операцій конвеєра з ним. У кінці роботи виводиться загальний об'єм завантажених та скинутих зливків. Початковий об'єм трюму та розміри зливків отримуються за допомогою генератора псевдовипадкових чисел.

Контрольні запитання

8.1. Чим відрізняються статичні змінні функції від звичайних її змінних?
8.2. За якою ознакою змінні поділяються на глобальні та локальні?
8.3. За якою ознакою змінні поділяються на статичні та автоматичні?
8.4. Чому один фрагмент коду не повинен мати кількох різних обов'язків?

Задачі

8.1. На контрольно-пропускному пункті (КПП) митниці працює одна бригада інспекторів. Автомобілі під'їздять, стають у чергу (якщо вона є) й проходять перевірку в порядку черги. Для кожного автомобіля відома тривалість його перевірки t: автомобіль виїздить з КПП через t одиниць часу після початку його перевірки. Моменти прибуття автомобілів задано відносно прибуття попереднього автомобіля. Треба визначити моменти виїзду автомобілів.
Момент прибуття та тривалість обробки автомобіля вводяться за допомогою клавіатури, тривалість 0 позначає кінець роботи. Після кожного введення виводиться момент виїзду автомобіля.

8.2. За умов попередньої задачі на КПП митниці працює не одна, а дві бригади інспекторів, і кожен автомобіль перевіряє одна з них. Якщо вільні обидві бригади, починати роботу може будь-яка з них. Визначити моменти виїзду автомобілів.

8.3. За умов попередньої задачі вхідні дані отримуються за допомогою генератора псевдовипадкових чисел, де момент прибуття може набувати значення з відрізка [0; 100], а тривалість перевірки — з відрізка [0; 200].

8.4. На вхід з клавіатури подається послідовність відрізків прямої, які задано координатами кінців. Знайти перетин відрізків. Робота закінчується, якщо перетин уже введених відрізків порожній або введено відрізок нульової довжини.

8.5. На вхід з клавіатури подається послідовність квадратів площини у вигляді координат центру та довжини сторін, паралельних осям координат. Знайти перетин квадратів. Робота закінчується, якщо перетин уже введених квадратів порожній або введено квадрат із нульовою стороною.

8.6. На вхід з клавіатури подається послідовність чисел a_1, a_2, …, яка закінчується повторним уведенням попереднього числа (другий раз воно в послідовність не входить). Кількість чисел нічим не обмежено. Написати програму, яка:
а) обчислює середнє арифметичне введених чисел;
б) обчислює середнє геометричне введених чисел;
в) визначає найменший елемент послідовності;
г) визначає, чи містить послідовність від'ємні числа;

кож символ підкреслення `_` та «долар» `$`;

— ***десяткові цифри*** `0`, `1`, `2`, …, `9`;

— ***знаки пунктуації*** `+ - * ~ / \ % ? ! = < > & | ^ # . , ; : ' " ( ) [ ] { }` ;

— деякі інші символи.

З символів алфавіту утворюються ***лексеми*** — «слова», тобто послідовності символів, які розглядаються як неподільні й самі по собі мають деякий зміст. Серед лексем мови C++ виділяють константи, імена, знаки операцій та інші.

- ***Константа*** (*constant*) — це позначення значення (числового або іншого). Константи також називають ***літералами*** (*literals*) .

Нам добре знайомі числові константи, наприклад, `273`, `3.1415926`, `2.71828` (ціла частина від дробової відокремлюється крапкою, а не комою). В апострофах записуються символьні константи, у лапках — рядкові, наприклад, `'3'`, `'A'`, `'\n'`, `"University"`, `"3.1415926"`, `""`. Докладніше константи описано нижче.

- ***Ім'я*** — це послідовність букв алфавіту й цифр, що починається з букви[5], наприклад, `A`, `x1`, `_1a$2`, `best_student`. Великі й малі букви в іменах ***відрізняються***: `Name`, `NAME`, `name` — це різні імена. Довжину імен ***не обмежено***[6].

Ім'я завжди ***іменує*** якийсь об'єкт; воно виділяє його з-поміж інших, тобто ***ідентифікує***, тому ім'я ще називають ***ідентифікатором***.

Деякі імена (це англійські слова або їх скорочення) мають спеціальний зміст, наприклад, `const`, `else`, `int`, `sizeof`. Вони називаються ***зарезервованими словами***. Використовувати їх за іншим призначенням не можна (цього просто не дозволить компілятор). Зарезервовані слова інколи вважаються окремим різновидом лексем.

- ***Знак операції***, або ***оператор*** (*operator*) — це позначення операції, виконання якої над числами та іншими значеннями породжує число або інше значення. Значення, до яких застосовується операція, називаються її ***операндами*** (*operand*), а породжуване значення — ***результатом*** (*result*).

Операції позначаються як окремими символами, наприклад `+` або `-`, так і іменами або послідовностями інших символів, наприклад `sizeof` або `<=`.

Лексеми часто відокремлюються так званими ***порожніми символами*** — вони не мають зображення й з'являються в тексті програми, якщо,

[5] Символ підкреслення `_` може бути першою літерою імені, але імена, що починаються з одного чи двох `_`, мають у C++ спеціальне призначення. Тому використовувати імена з `_` на початку не рекомендується.

[6] Формально довжину імен у C++ не обмежено, але, наприклад, компілятор Microsoft Visual C++ 2005 розрізняє лише перші 2048 символів імені.

створюючи його, натискати клавіші **Space** (пропуск), **Enter** («кінець рядка») і **Tab** (символ табуляції). Будь-які дві лексеми можна відокремити порожніми символами в довільній кількості. Компілятор, обробляючи текст програми з деякої поточної позиції, завжди виділяє найдовшу можливу лексему. Тому порожні символи між сусідніми лексемами обов'язкові лише тоді, коли, записані разом, ці лексеми утворюють лексему. Наприклад, записи **1+2**, **sizeof-2** та **1<=2** позначають застосування відповідних операцій і не є лексемами, а **sizeof2** є ім'ям (а також **однією** лексемою).

2.4. Поняття типу даних

Комп'ютер може обробляти дані різної природи. Так, рік народження людини ми зображаємо цілим числом, вага може бути не лише цілою, але й дробовою, а прізвище ми позначаємо послідовністю символів. Якщо числа можна додавати та віднімати, то послідовності символів можна дописувати одна до одної. Отже, різні за своєю природою дані (значення) мають різні способи зображення та допустимі операції, і за цими ознаками поділяються на типи.

2.4.1. Типи даних

- ***Тип даних*** — це множина значень разом із множиною застосовних до них операцій. Множина значень називається ***носієм*** типу, множина операцій — ***сигнатурою***.

У програмуванні використовуються типи чисел, символьних та логічних значень. Ці значення розглядаються як цілісні елементи, що не мають окремих складових частин, тому називаються ***скалярними*** на відміну від ***структурних***, складених з окремих компонентів. Типи скалярних даних називаються ***скалярними***. Кожна мова програмування забезпечує цілу сім'ю скалярних типів. Вони мають певні, означені наперед (***стандартні***) імена й називаються ***базовими скалярними типами*** мови.

Серед базових скалярних типів мови C++ є типи цілих і дійсних чисел з іменами **int** і **double**, а також тип символів **char** (*int* і *char* — скорочення від *integer* і *character*, *double* — «подвійний») та логічний тип **bool** (*bool* — скорочення від *boolean*). Крім того, часто користуються й типом беззнакових цілих чисел **unsigned int**, або просто **unsigned**.

2.4.2. Мінімальні відомості про базові типи

Типи **int**, **unsigned**, **char** та **bool** у мові C++ відносять до ***цілих*** (*integral*) типів. Числа типу **int** зображаються в знаковій формі й займають 4 байти, тому носій цього типу утворено цілими числами від -2147483648 до 2147483647 (див. с. 14). Числа типу **unsigned** (перекладається як *беззнаковий*) мають зображення без знака, тому їх діапазон — від 0 до 4294967295. Мінімальне та максимальне значення типу **int** позначено іменами **INT_MIN** та **INT_MAX**, максимальне значення типу **unsigned** — **UINT_MAX**.

значення **sort**	співвідношення	нове значення **sort**
CONSTANT	**prev>x**	**DOWN**
	prev<x	**UP**
UP	**prev>x**	**OTHER**
DOWN	**prev<x**	**OTHER**

Запишемо означення типів послідовності та функцію **process**.

```
const int CONSTANT=0, UP=1, DOWN=-1, OTHER=2;
int process(int &sort)
{ double prev, x;
  int state=OK;
  state=get(prev);
  if (state==EOS) return EMPTY;
  if (state==ERROR) return ERROR;
  sort=CONSTANT;
  while ((state=get(x))==OK){
    switch (sort) {
      case CONSTANT:
        if (prev>x) sort=DOWN;
        else if (prev<x) sort=UP;
        break;
      case UP:   if (prev>x) sort=OTHER; break;
      case DOWN: if (prev<x) sort=OTHER; break;
    }
    prev=x;
  } // state==EOS || state==ERROR
  if (state==ERROR) return ERROR;
  return OK;
}
```

Перевагою цього коду є те, що він ***цілком прозоро задає логіку перевірки послідовності***. Автори бачили багато помилкових програм перевірки послідовності на монотонність. Найгіршим у них було те, що спроби виправити помилки спричиняли появу інших помилок, код заплутувався, і вихід був лише один — почати все «з нуля».

«Родзинкою» наведено способу перевірки послідовності є те, що значення змінної **sort** фактично ***визначає стан процесу обробки послідовності та керує цим процесом***. Саме за станом вирішується, яке зі співвідношень між попереднім і наступним елементом слід перевіряти, а результат цієї перевірки визначає новий стан обробки.

- Підхід з використанням станів застосовний до великої кількості задач програмування.

Вправа 8.4. Написати головну функцію, яка для визначення типу послідовності викликає функцію **process** та за її результатами виводить на екран відповідне повідомлення.

дено максимальний елемент. Додамо до інших імен констант ім'я **EMPTY**.

```
const int OK=0, EOS=1, ERROR=2, EMPTY=1;
```

Згідно з наведеними міркуваннями змінимо функцію **process** та головну функцію.

```
int process(double &max)
{ double x;
  int state=OK;
  state=(get(max));
  if (state==EOS) return EMPTY;
  if (state==ERROR) return ERROR;
  // помилки немає, послідовність непорожня
  while ((state=get(x))==OK){
    if (x>max) max=x;
  } // state==EOS || state==ERROR
  if (state==ERROR) return ERROR;
  return OK;
}
int main()
{ double max;
  int state;
  state=process(max);
  if (state==ERROR)
     cout<<"An ERROR exists.\n";
  else if (state==EMPTY)
     cout<<"The sequence is EMPTY.\n";
  else
  // помилки немає, послідовність непорожня
     cout << "max=" << max << endl;
  system("pause"); return 0;
}
```

Приклад: монотонна послідовність. Необхідно визначити, чи є вхідна послідовність монотонною. Розв'яжемо загальнішу задачу: визначимо тип непорожньої послідовності — стаціонарна або нестаціонарна, а саме, неспадна, незростаюча або немонотонна. Ознакам цих типів дамо імена відповідно **CONSTANT**, **UP**, **DOWN** та **OTHER**. Одне з цих значень функція **process** має повернути за допомогою свого параметра-посилання **sort** (тип), якщо отримано хоча б один елемент послідовності.

Алгоритм обробки послідовності такий. Отримаємо перший елемент послідовності. Послідовність з одного елемента є стаціонарною, тому змінна **sort** має значення **CONSTANT**. Далі, нехай оброблений початок послідовності має тип **sort**, відомий попередній елемент послідовності **prev** та отримано новий елемент **x**. Залежно від співвідношення між значеннями **prev** та **x**, тип послідовності змінюється, як це вказано в таблиці. В інших ситуаціях тип послідовності не змінюється, тому ці ситуації не вказано.

Значення символьного типу **char** займають один байт; усього їх 256. Логічний тип **bool** має два значення, що позначаються іменами **false** і **true** (хибність та істина) та зображуються в комп'ютері числами **0** і **1** відповідно. Взагалі, значення «хибність» та «істина» називаються ***булевими*** на честь видатного англійського логіка й математика Джорджа Буля.

Дійсні числа типу **double** займають 8 байт, тому їх множина містить майже 2^{64} чисел (кілька з усіх можливих комбінацій нулів і одиниць не є зображеннями чисел). Ця ***обмежена скінченна*** множина чисел є симетричною відносно числа 0. Найбільше та найменше, відмінне від 0, додатні числа позначено іменами **DBL_MAX** та **DBL_MIN**. Ці числа дорівнюють приблизно 10^{-308} та 10^{308}. Зауважимо: всі цілі числа типу **int** мають зображення в типі **double**.

Цілі та дійсні типи разом називають ***арифметичними***.

2.5. Різновиди констант

У цьому розділі наведено мінімальні відомості про вигляд констант мови C++ та засоби їх виведення.

2.5.1. Символьні константи

Символьна константа — це символ в апострофах (**'A'**, **'1'**, **'.'**) або два символи в апострофах, першим з яких є **** (зворотна скісна, або *backslash*). Наприклад, константи **'\''**, **'\"'**, **'\\'** позначають символи, які називаються «апостроф», «лапки» та «зворотна скісна». Є кілька констант, наприклад, **'\n'** або **'\t'**, що містять малу латинську букву (***керуючі символи***) . Вони використовуються в спеціальний спосіб під час виведення на зовнішні носії даних. Зокрема, **'\n'** задає перехід на новий рядок екрана.

Символьні константи зображують значення типу **char**, які займають один байт. Цей байт як зображення числа без знаку дає ***код символу*** (число від 0 до 255) , а зі знаком — від -128 до 127. Відповідність між числами від 0 до 127 та символами зафіксовано в ***Американському стандартному коді для обміну інформацією*** (так звана ***таблиця ASCII***), решті кодів можуть відповідати різні набори символів. Наприклад, константа **' '** («пропуск») має код 32, а **'\0'** — код 0. Однією з властивостей таблиці ASCII є таке:

– символам **'0'**, **'1'**, …, **'9'** відповідають послідовні коди від 48 до 57;

– символам **'A'**, **'B'**, …, **'Z'** — від 65 до 90;

– символам **'a'**, **'b'**, …, **'z'** — від 97 до 122.

Інструкція **cout<<'Z';** виводить значення константи **'Z'** — на екрані з'являється **Z**. Аналогічно можна вивести інші символьні константи. Для того, щоб наступне повідомлення виводилося з нового рядка, можна скористатися інструкцією **cout<<endl;** або **cout<<'\n';** .

Вправа 2.2. Що буде виведено за виконання програми?

```
#include <iostream>
using namespace std;
int main() {
   cout<<'1';  cout<<'\n';
   cout<<'Y';  cout<<'e';
   cout<<'s';  cout<<endl;
   system("pause"); return 0;
}
```

Вправа 2.3. Написати програму, яка друкує символи `'`, `"`, `\`.

- За інструкцією вигляду **`cout<<`*`константа`*`;`** обробляється не константа, а зображене нею значення. Послідовність символів, утворена за значенням, може відрізнятися від константи.

2.5.2. Рядкові константи

Рядкові константи позначають послідовності символів і записуються в лапках, наприклад, **`"You are welcome!"`**. Символи `'`, `\`, `"` у рядковій константі записують як **`\'`**, **`\\`**, **`\"`** відповідно. До значення, заданого рядковою константою, у пам'яті додається символ **`'\0'`**, який позначає кінець послідовності символів. Окрім того, якщо в константі присутній символ **`'\0'`**, то значенням, яке вона задає, є послідовність символів перед **`'\0'`**. Наприклад, константі **`"\"A'BC\"\0zz"`** відповідає послідовність байтів з такими символами.

`"  A  '  B  C  "  '\0'  z  z  '\0'`

Значення утворено першими шістьма символами **`"A'BC"`**. Саме воно буде виведено за виконання інструкції **`cout<<"\"A'BC\"\0zz";`** .

За допомогою рядкових констант перепишемо програму з вправи 2.2.

```
#include <iostream>
using namespace std;
int main() {
   cout<<"1\nYes\n";
   system("pause"); return 0;
}
```

prog003.cpp

Вправа 2.4. Написати програму, що виводить ім'я дискового каталогу, в якому встановлено систему програмування C++.

Вправа 2.5. Поясніть різницю між значеннями **`"a"`** та **`'a'`**.

2.5.3. Цілі константи

Цілі константи позначають цілі числа й мають десяткову, вісімкову та шістнадцяткову форму запису (про системи числення див. додаток А.). У мові C++ цілі константи невід'ємні. ***Десяткова константа*** — це, як і в математиці, послідовність десяткових цифр, що не починається з **`0`**: **`273`**, **`1024`** тощо. Запис ***шістнадцяткових констант*** починається з символів **`0x`** або **`0X`**. Так, константа **`0x11`** позначає число 17, а **`0xf`** та **`0xF`** — число 15. Ціла константа, що починається з **`0`** й далі містить цифри від 0 до 7, є ***вісімковою***. Наприклад, константи **`0`** та **`010`** позначають числа 0 та 8 відповідно.

Наведена структура програми дозволяє за необхідності легко змінити джерело послідовності, умову її завершення або тип вхідних даних, оскільки для цього доведеться модифікувати тільки функцію **`get`**. Зміна ж дій, виконуваних з елементами послідовності, потребує змін тільки у функції **`process`**.

8.3.2. Зміна умови завершення послідовності

Розглянемо попередню задачу за умови: ознакою закінчення послідовності є число, що збігається з першим і до послідовності не входить. Наприклад, послідовність 1, 3, 4 задається входом **`1 3 4 1`**. Щоб визначити, чи досягнуто кінець послідовності, у функції **`get`** будемо зберігати перше введене значення (у змінній **`first`**). Оскільки перше значення обробляється не так, як наступні, необхідно знати, вводиться перший чи не перший елемент послідовності. Ознаку першого елемента виразимо логічною змінною **`isFirst`**. Змінні **`first`** та **`isFirst`** мають бути доступними виключно у функції **`get`** й зберігати своє значення між її викликами, тому зробимо їх ***статичними***. Отже, змінимо лише функцію **`get`**.

```
int get(double &x){
  static bool isFirst=true;
  static double first;
  double y;
  if (isFirst) cout<<"Enter real:\n";
  else cout<<"Enter real ("<<
             first<<" to finish):\n";
  if (!(cin>>y)) return ERROR;
  if (isFirst)
    { isFirst=false; first=y; x=y; return OK; }
  else
    { if (y==first) return EOS;
      x=y; return OK;
    }
}
```

8.3.3. Зміна дій, виконуваних з послідовністю

Розглянемо, як змінюється програма, якщо з послідовністю виконуються інші дії.

Приклад: максимальний елемент. Необхідно знайти максимальне число послідовності. Отримуємо перше число — спочатку воно є максимальним, тому збережемо його в змінній **`max`**. Далі в головному робочому циклі отримуємо наступні елементи, порівнюємо їх із значенням **`max`** та, за необхідності, змінюємо його.

У цій задачі є відмінність від попередньої: у порожній послідовності сума чисел нульова, але максимального елемента немає. Тому, якщо послідовність порожня, функція **`process`** має повернути ознаку цього факту — значення **`EMPTY`**. Отже, результатом обробки послідовності є одна з ознак: **`ERROR`** — виникла помилка, **`EMPTY`** — послідовність порожня, **`OK`** — знай-

EOS (скорочення від *end of sequence* — послідовність вичерпано) та **ERROR** (спроба отримати елемент неуспішна). Для того, щоб обробити помилку в завершенні, будемо зберігати статус виклику **get** у допоміжній змінній **state** (стан). Отже, основний робочий цикл буде таким.

```
while ((state=get(x))==OK)
  обробити x;
```

Продовжимо відокремлювати обчислення від повідомлень. Обчислення реалізуємо в окремій функції **process**, а виведення результатів — у головній функції. Функція **process** повертає ознаку успішності обробки послідовності (**OK**, якщо все гаразд, інакше **ERROR**) та обчислену сума за допомогою параметра-посилання. Головна функція викликає **process**, отримує від неї результати й виводить відповідні дані на екран.

Реалізуємо функцію **get** за умови, що послідовність дійсних чисел вводиться з клавіатури, а число 0 вважається ознакою кінця послідовності.

```
#include <iostream>
using namespace std;
const int OK=0, EOS=1, ERROR=2;
int get(double &x) // отримати наступний елемент
{ double y;
  cout<<"Enter one real (0 to finish):\n";
  if (!(cin>>y)) return ERROR;
  if (y==0.0) return EOS;
  x=y; return OK;
}
int process(double &sum)      // розв'язання задачі
{ double x;
  int state=OK; sum=0;        //ініціалізація
  while((state=get(x))==OK){ //є поточний елемент
    if(x>0.) sum+=x;          //обробити поточний елемент
  }
  // state == EOS || state == ERROR
  if (state == ERROR)
    return ERROR;
  else
    return OK;
}
int main()
{ double sum;
  int state;
  state = process(sum);
  // виведення результатів
  if (state==ERROR)
     cout<<"An ERROR exists.\n";
  else
     cout << "sum=" << sum << endl;
  system("pause"); return 0;
}
```

prog021.cpp

Якщо в програмі спеціально не вказано інше, значення цілих констант незалежно від форми їх запису ***виводяться в десятковому записі***. Так, за інструкцією **cout<<11;** на екран виводиться **11**, а за **cout<<0x11;** — **17**.

Вправа 2.6. Пояснити різницю між а) **0**, **'\0'** та **'0'**; б) **123** та **"123"**.

Вправа 2.7. Що виводиться на екран за виконання двох таких програм? Поясніть, у чому різниця.

```
#include <iostream>
using namespace std;
int main() {
 cout<<13;   cout<<'\n';
 cout<<013;  cout<<'\n';
 cout<<0x13; cout<<endl;
 system("pause");
 return 0;
}
```

```
#include <iostream>
using namespace std;
int main() {
 cout<<"13";cout<<'\n';
 cout<<"013";cout<<'\n';
 cout<<"0x13";cout<<endl;
 system("pause");
 return 0;
}
```

2.5.4. Дійсні константи

Розглянемо приклад. Число **12,34** можна позначити, наприклад, так: $123{,}4\cdot10^{-1}$. У цьому записі є ціла частина **123**, дробова частина **,4** і десятковий порядок **–1**. Запису відповідає константа **123.4E–1**, в якій **123** — ***ціла*** частина, **.4** — ***дробова***, а **E–1** — ***порядок***. Це саме число можна зобразити й іншими константами: **12.34**, **1234.e-03**, **1234e-03**, **.1234E2**, **0.01234E+3** та іншими. Отже, ціла частина — це непорожня послідовність цифр, дробова — послідовність цифр з ***крапкою*** на початку, порядок — латинська буква **E** або **e** з однією або декількома цифрами, можливо, зі знаком **+** або **–**.

- Константа, в якій ціла частина містить одну цифру від **1** до **9**, називається ***нормалізованою***, наприклад, **1.234E+1**, **9.81** або **1.0E2** (для числа 0 такою є **0.0**).

Дійсні константи записуються тільки як десяткові. Якщо константа має цілу частину, то за нею слідує дробова частина й порядок (можливо, одне з них). При цьому в дробовій частині, якщо вона є, цифр може не бути, але крапка обов'язкова. Якщо цілої частини немає, то константа починається крапкою, після якої має бути хоча б одна цифра. Порядок необов'язковий. Приклади наведено вище. Перед константою може бути знак **–**; тоді вона задає від'ємне число: **–1.2345E1**.

Вправа 2.8. Поясніть, у чому різниця між **314.** та **314**.

Вправа 2.9. Поясніть, у чому різниця між **12.151** та **"12.151"**.

Значення дійсних констант виводяться в різних формах (***форматах***) . Значення, зображені дійсними константами, виводяться або з ***фіксованою*** (*fixed*) ***крапкою***, або в ***нормалізованій*** («***науковій***», *scientific*) ***формі***, наприклад, для числа 12,34 це відповідно **12.34** та **1.234e+001**. Від чого залежить формат виведення та як ним керувати сказано у підрозділі 3.3.2 на с. 44.

2.6. Поняття змінної величини

2.6.1. Змінні величини

Поняття змінної числової величини з'явилося в роботах Декарта в XVII ст. Пізніше з'ясувалося, що змінна величина може бути не лише числовою. У найширшому розумінні ***змінна величина*** — це загальне поняття реального чи уявного об'єкта (або його окремої характеристики), який може перебувати в різних станах. Змінні величини позначають ***іменами***, наприклад, у другому законі Ньютона $a = F/m$ імена m, a, F позначають змінні — масу тіла, прискорення його руху та силу, що діє на нього. У програмуванні змінна є моделлю (зображенням) об'єкта в пам'яті комп'ютера.

- ***Змінна в машинній програмі*** — це ділянка пам'яті, яка може мати різні стани й зображувати ними різні стани об'єкта.

Змінну в програмі мовою високого рівня позначає ***ім'я***. Кожна ділянка пам'яті комп'ютера має ***адресу*** (номер першого байта ділянки), і цією адресою позначається в машинній програмі. Кажуть, що адреса ***вказує***, або ***посилається*** на ділянку. За трансляції програми ім'я змінної перетворюється на адресу деякої ділянки пам'яті. Отже, будемо вважати, що під час виконання програми ім'я змінної ***вказує*** на ділянку пам'яті (рис. 2.1).

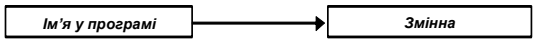


Рис. 2.1. Змінна та її ім'я

Станам об'єкта, зображуваного змінною, відповідають її ***значення*** (числові та інші), що належать певному типу. Тип змінної та її ім'я задаються в ***інструкції оголошення імені змінної***, що має такий вигляд.

```
ім'я-типу ім'я-змінної;
```

Після імені типу можна записати кілька імен змінних через кому. Інструкція оголошення імені змінної одночасно є й ***означенням змінної***, тобто задає ділянку пам'яті, на яку вказує ім'я змінної.

Приклади

1. Інструкції

```
int numberOfStudents;
double radius, circleLength;
```

задають утворення ділянки пам'яті, яка ідентифікується іменем **numberOfStudents** і може зберігати цілі числа, та двох ділянок з іменами **radius** і **circleLength**; їх значеннями можуть бути дійсні числа.

2. Квадратне рівняння $ax^2+bx+c = 0$ визначається трьома дійсними коефіцієнтами a, b, c. Щоб зобразити рівняння, означимо три змінні: **double a, b, c;**. ◂

Імена змінних та інших об'єктів обирайте так, щоб вони були пов'язані з об'єктами або характеристиками, які вони зображують, наприклад, **temperature**, **radius**, **circleLength** (температура, радіус, довжина кола). Завдяки таким ***змістовним*** іменам текст програми стає більш зрозумілим.

Вправа 8.3. Написати функцію, кожен виклик якої повертає наступне число Фібоначчі, тобто перший виклик — 0, другий — 1, наступні — 1, 2, 3, 5 тощо.

8.3. Обробка послідовностей

8.3.1. Загальна схема побудови програми

У прикладі на с. 81 розбиралася задача: визначити суму додатних елементів послідовності чисел, що вводиться з клавіатури й закінчується числом 0, яке до послідовності не входить. Для цієї задачі характерним є те, що:

– входом є послідовність чисел, кількість яких наперед невідома;

– вхід містить ознаку завершення послідовності (число 0).

Ця задача є типовою, тому побудуємо її розв'язання так, щоб його можна було пристосувати до інших аналогічних задач з мінімальними змінами. Сформулюємо алгоритм розв'язання в загальному вигляді.

```
Ініціалізація
Головний робочий цикл, а саме:
while (є наступний елемент)
  обробити наступний елемент
Завершення
```

Ініціалізація — це дії, підготовчі до отримання вхідної послідовності (наприклад, вивести запрошення для користувача) та до обробки введених елементів (наприклад, встановити в 0.0 суму введених додатних чисел). ***Головний робочий цикл*** — це повторювані дії з отримання та обробки елементів послідовності. ***Завершення*** — це заключні обчислення та, можливо, дії, потрібні, щоб коректно закінчити обробку послідовності.

Розглянемо головний робочий цикл детальніше. Кожен фрагмент коду повинен мати один обов'язок, тому отримання елементів послідовності слід, наскільки це можливо, відокремити від їх обробки. Отримання наступного елемента «сховаємо» у функцію. Джерелом, з якого надходить послідовність, може бути не тільки клавіатура, але за межами функції ***від джерела ніщо не залежатиме***.

Якою має бути ця функція? Вона має повертати наступний елемент послідовності, якщо він є, а якщо немає — якось про це повідомляти. Вона також має повідомляти про неуспішність спроби отримати наступний елемент, тобто про помилку. Звідси, функція має повертати ***статус останньої операції*** отримання елемента послідовності та сам цей елемент. Статус будемо повертати як значення функції, а елемент — за допомогою параметра-посилання. Домовимося: якщо отримати наступний елемент послідовності неможливо, функція не змінює значення параметра. Отже, прототип функції буде таким.

```
int get (double &x);
```

Розглянемо використання функції **get** у головному циклі. Значення, які може повернути функція **get**, позначимо іменами **OK** (елемент отримано),

Функція **time** звертається до вбудованого годинника й отримує від нього число, пов'язане з поточним днем та часом.

Вправа 8.1. Напишіть функцію, яка за кожного свого виклику повертає наступне псевдовипадкове дійсне число з відрізка [*a*; *b*].

Вправа 8.2. Напишіть функцію, яка за кожного свого виклику повертає наступне псевдовипадкове ціле число з проміжку [*a*; *b*]. *Вказівка*. Перетворити число *x*, наприклад, з проміжку від 0 до 100 на число від –5 до 4 можна за формулою *x*%(4–(–5)+1)+(–5).

8.2. Статична пам'ять

Нагадаємо (див. розділ 4.4): ім'я, оголошене зовні функцій, називається ***глобальним***. Змінні, оголошені зовні функцій, існують протягом усього часу виконання програми й зберігаються в окремій області пам'яті. Область пам'яті зі змінними, що існують протягом усього часу виконання програми, та самі ці змінні називаються ***статичними***. Статичними можуть бути як глобальні змінні, так і локальні.

Приклад. Розглянемо програму, в якій статичною є саме локальна змінна.

```
#include <iostream>
using namespace std;
int nCalls() {
  static int i=0;
  return ++i;
}
int main(){
  cout << nCalls() << endl;
  cout << nCalls() << endl;
  system("pause"); return 0;
}
```

prog020.cpp

Специфікатор **static** в означенні **static int i=0;** вказує, що:

– пам'ять для змінної **i** виділяється в статичній пам'яті програми ***на початку виконання програми*** й змінна відразу ініціалізується значенням **0**;

– у викликах функції **nCalls** для змінної **i** ***пам'ять не виділяється***;

– між викликами функції **nCalls** змінна **i** ***зберігає своє значення***.

Кожен виклик функції **nCalls** збільшує значення змінної **i** на **1** і повертає його. Отже, за першого виклику функція поверне 1, за другого — 2 тощо. Взагалі, функція **nCalls** повертає кількість своїх викликів.

Специфікатор **static** не впливає на область дії оголошення імені. Ім'я **i** оголошено в тілі функції **nCalls**, тому ім'я змінної **i** є локальним у функції, а за межами функції **nCalls** ця змінна недоступна. ◂

- Локальні статичні змінні варто використовувати, коли необхідно забезпечити недоторканність значень змінних функції між її викликами.

Ім'ям змінної не може бути зарезервоване слово. Імена, означені в бібліотеках системи програмування, використовувати як імена змінних можна, але краще цього не робити.

Надалі будемо дотримуватися таких угод. Ім'я змінної починаємо з малої букви; якщо воно утворено з кількох слів, то всі слова, окрім першого, записуємо з великої букви, наприклад, **circleLentgh**. Ця система запису сьогодні є найпоширенішою у світі й через зовнішній вигляд імен має назву ***«верблюжий запис»*** (*camel notation*) .

Якщо потрібна невелика й відома наперед кількість змінних, що позначають однотипні сутності, то в кінці імені можна дописати номер, наприклад, **radius1**, **radius2**.

Існує давня традиція, за якою імена **i**, **j**, **k**, **m**, **n** даються змінним цілих типів. Найчастіше ці змінні позначають індекси (номери) або кількості.

Узагалі, змінна може бути складовою частиною іншої змінної. Тоді вона позначається не ім'ям, а деяким складнішим виразом. Але поки що не будемо на цьому зосереджуватися.

Вправа 2.10. Написати означення змінних, які зображують: а) день, місяць та рік народження; б) точку площини; в) клітинку шахового поля; г) дріб; д) квадратне рівняння; е) прямокутник на площині.

Вправа 2.11. Пояснити різницю між: а) **a** і **'a'**; б) **a** і **"a"**.

2.6.2. Уведення значень у змінні та виведення їх на екран

Є два способи надати змінній значення — увести з зовнішнього носія або присвоїти. У найпростішому випадку ***інструкція введення*** значення в змінну має вигляд

cin >> ***ім'я-змінної*****;**

і задає введення з клавіатури. Ім'я **cin**, як і **cout**, оголошено у файлі **iostream**. Виконуючи операцію введення, комп'ютер зупиняється й очікує на значення для змінної. У відповідь слід на клавіатурі набрати деяку послідовність символів, що зображує значення (ці символи з'являться на екрані), та натиснути на клавішу **Enter**. Після цього за введеною послідовністю (вхідною константою) створюється відповідне значення й присвоюється змінній. За стандартних налаштувань запис числа сприймається як десятковий.

Якщо натиснути **Enter**, не набравши нічого, окрім пропусків, то комп'ютер і надалі чекатиме. Взагалі, перед вхідною константою можна набрати довільну кількість порожніх символів, що відповідають клавішам пропуску, табуляції та **Enter**. За стандартних налаштувань усі порожні символи буде пропущено.

В інструкції введення можна записати кілька імен змінних, кожне після «свого» знака **>>**. За виконання такої інструкції треба набрати на клавіатурі відповідну кількість вхідних констант, відокремивши їх одним або кількома порожніми символами.

Приклади

1. Нехай є змінна **`char c;`** і виконується інструкція **`cin >> c;`**. Щоб присвоїти змінній значення **`'A'`**, треба натиснути послідовно на клавіші **`A`** та **`Enter`**, а значення **`'7'`** — на клавіші **`7`** та **`Enter`**. Якщо перед клавішею **`A`** або **`7`** кілька разів натиснути на клавіші **`Enter`** або **`Space`**, результат буде той самий.

2. Нехай змінні **`double a, b, c;`** зображають коефіцієнти квадратного рівняння. Увести в них дійсні значення можна за допомогою трьох окремих інструкцій.

```
cin >> a; cin >> b; cin >> c;
```

Ті самі дії буде задано й однією інструкцією.

```
cin >> a >> b >> c;
```

В обох ситуаціях на клавіатурі слід набрати три вхідні константи, натискаючи між ними на клавіші пропуску, табуляції або **`Enter`**. Виконання інструкцій закінчиться, коли після третьої константи буде натиснуто на **`Enter`**. ◂

- Практично перед кожною інструкцією введення з клавіатури варто записати інструкцію виведення, яка запрошує до введення значень і вказує, скільки й яких типів.

Приклад. Уведення значень у дійсні змінні, що зображують коефіцієнти квадратного рівняння, можна оформити так.

```
cout << "Enter coefficients of quadratic equation ";
cout << "(three real or integer numbers):\n";
cin >> a >> b >> c;
```

Після появи цього запрошення курсор буде переведено в наступний рядок екрана, і з нього почнеться відображення символів, набраних на клавіатурі. ◂

Вивести значення змінної на екран можна за допомогою інструкції такого вигляду.

```
cout << ім'я-змінної;
```

Приклад. Після введення значень трьох дійсних змінних за інструкцією

```
cin >> a >> b >> c;
```

можна вивести їх на екран, відокремивши проміжками.

```
cout << a; cout << ' '; cout << b;
cout << ' '; cout << c;
```

Ті ж самі дії можна задати й однією інструкцією.

```
cout << a << ' ' << b << ' ' << c;
```

Якщо введено значення 1, –3, 2, то буде надруковано **`1 -3 2`**. ◂

Вправа 2.12. Що буде виведено на екран, якщо за виконання програми введено символи **`15 i`** та натиснуто на клавішу **`Enter`**?

```
#include <iostream>
using namespace std;
int main() {
   int i;  char a;
   cin >> i >> a;
```


Глава 8. Обробка та отримання послідовностей

8.1. Послідовності псевдовипадкових чисел

У багатьох задачах (від моделювання природних або соціальних процесів до розкладання карт) потрібні послідовності чисел, що належать певній множині, але більше ніяк не пов'язані одне з одним. Такі числа називаються випадковими. Послідовності випадкових чисел часто імітують, використовуючи ***генератор псевдовипадкових чисел*** — підпрограму, яка за певним алгоритмом утворює послідовність чисел, що виглядає випадковою. Щоб кожного разу отримувати різні послідовності, алгоритм ініціалізують числом, яке, як правило, визначає всю послідовність.

Мовою C++ послідовності псевдовипадкових чисел можна отримувати за допомогою двох функцій, оголошених у файлі **`cstdlib`**. Функція з прототипом **`void srand(unsigned int);`** встановлює початкове значення послідовності, використовуючи для цього аргумент у виклику (ним має бути невід'ємне ціле число). Функція з прототипом **`int rand(void);`** обчислює й повертає наступне псевдовипадкове значення. Усі значення є цілими числами в діапазоні від 0 до 32767. Число 32767 позначається константою **`RAND_MAX`**; її теж оголошено у файлі **`cstdlib`**.

Ім'я **`rand`** є скороченням слова *random* (випадковий), а **`srand`** — скороченням *seed random* (засіяти випадкове). Значення аргументу у виклику **`srand`** називається ***зерном***, оскільки «зерно» є головним значенням англійського слова *seed*.

Розглянемо приклад програми, яка отримує від користувача зерно, створює тисячу псевдовипадкових дійсних чисел у діапазоні [0; 1] та обчислює їх середнє арифметичне (воно має бути досить близьким до 0,5). Дійсні числа з [0; 1] утворюються шляхом ділення цілих псевдовипадкових чисел на **`RAND_MAX`**.

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main() {
  double sum;
  int k, seed;
  cout << "Enter int seed>"; cin >> seed;
  srand(seed);
  for(k=0,sum=0; k<1000; ++k)
    sum += rand()/double(RAND_MAX);
  cout <<  sum/1000 << endl;
  system("pause"); return 0;
}
```

prog019.cpp

Зерно можна зробити випадковим за допомогою функції **`time`**, яку оголошено у файлі **`time.h`**. Для цього потрібен такий виклик.

```
srand(unsigned(time(NULL)));
```

чин. Розглянемо, наприклад, обчислення **f(6)**. Очевидно, що виклик **f(4)** виконується двічі, **f(3)** — тричі, **f(2)** — п'ять разів, **f(1)** — вісім разів, **f(0)** — п'ять разів.

У загальному ж випадку за $n > 3$ обчислення F_n породжує два виклики **f(n-2)**, три виклики **f(n-3)**, п'ять викликів **f(n-4)**, …, F_{n-1} викликів **f(2)**, F_n викликів **f(1)** та F_{n-1} викликів **f(0)**. Щоб виконати всі ці дії, потрібно часу не менше ніж для перенесення ханойської вежі висотою n. Але, на відміну від задачі про вежі, ці дії не є обов'язковими, адже циклічний алгоритм на с. 86 (підрозділ 6.3.2) вимагає лише одного додавання та двох присвоювань на кожне з чисел F_2, F_3, …, F_n (плюс присвоювання, додавання та порівняння номерів).

Вправа 7.2. Напишіть функцію, яка за цілим значенням **n** повертає кількість вкладених викликів наведеної функції **f** за виклику **f(n)**.

Контрольні запитання

7.1. Чому пам'ять, утворену локальними змінними викликів функцій, називають програмним стеком?

7.2. За якої умови у функції можна використовувати імена, не оголошені в ній?

7.3. Яке ім'я називається у функції локальним, а яке — глобальним?

7.4. Чи можна використовувати локальні імена функції в інших функціях, записаних після неї?

7.5. Чи є глобальними змінні функції **main**?

Задачі

7.1. Арифметичні вирази з константами, знаками операцій (+, -, * або /) та дужками опишемо так:

а) дійсна константа є арифметичним виразом;

б) якщо E і F є арифметичними виразами, то (E+F), (E-F), (E*F), (E/F) також є виразами.

в) інших виразів, окрім утворених за правилами пп. а) та б), не існує.

Написати програму, яка за введеним з клавіатури арифметичним виразом виводить його значення.


```
    cout<<"i="<<i<<' '<<"a="<<a<<endl;
    system("pause"); return 0;
}
```

2.6.3. Присвоювання: операція, інструкція, вираз

Надати значення змінній можна за допомогою ***операції присвоювання значення змінній***, яка копіює значення деякого виразу в змінну. Знаком операції є **=**, її можна задати в ***інструкції присвоювання*** (*assignment statement*) такого вигляду.

ім'я-змінної = вираз;

Спочатку обчислюється вираз праворуч, потім його значення записується в змінну, вказану ліворуч. Найпростішими виразами є константи та імена змінних, складніші описано в наступних розділах і главах.

Приклади

1. Якщо є символьна змінна **char c;**, то інструкція **c='0';** надає їй значення **'0'**.

2. Якщо є ціла змінна **int i;**, то перша з інструкцій **i=13; cout<<i;** надає їй значення 13, а друга виводить символи **13** на екран.

3. Нехай дійсні змінні **double a, b, c;** зображують квадратне рівняння $ax^2+bx+c = 0$. Щоб задати рівняння $x^2-3x+2 = 0$, присвоїмо їм значення відповідно 1, -3, 2, а потім виведемо на екран.

```
a=1.0; b=-3.0; c=2.0;
cout << a << ' ' << b << ' ' << c;
```

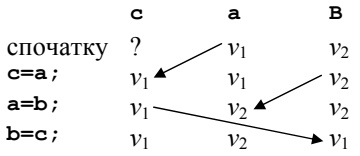
За виконання має бути надруковано **1 –3 2**. ◂

В інструкції присвоювання ***ім'я-змінної = вираз;*** ліворуч ім'я змінної позначає ділянку пам'яті, в яку значення записується, а у виразі праворуч — значення, яке добувається з ділянки. Добування значення змінної називається її ***розіменуванням***.

Приклади

1. Якщо змінні **a** та **b** однотипні, то інструкція **a = b;** копіює значення змінної **b** у змінну **a**.

2. Щоб обміняти місцями значення двох однотипних змінних **a** та **b**, скористаємося третьою змінною **c** того самого типу. Припустимо, що змінні **a** і **b** мають початкові значення v_1 і v_2, і праворуч вкажемо значення, яких набувають змінні після виконання інструкцій. Початкове значення змінної **c** вважається невизначеним і позначено знаком **?**.



◂

- Перед тим, як використовувати значення змінної, ***необхідно забезпечити***, щоб раніше ця змінна його отримала (за присвоювання або вве-

дення), інакше можливі непередбачувані наслідки. Як правило, компілятори лише попереджають, що змінна не отримувала значення перед використанням, тому за цим має стежити програміст.

Вправа 2.13. Написати інструкції оголошення дійсних змінних **a**, **b**, **c**, присвоювання їм початкових значень та обмін місцями цих значень «за колом»: значення **a** пересувається до **c**, **c** до **b**, **b** до **a**. *Порада*: почніть з проектування коду.

- Запис вигляду ***ім'я-змінної = вираз*** (без **;** в кінці) називається ***виразом присвоювання***. Його значенням є значення, присвоєне змінній.

Вираз присвоювання можна записати праворуч від знака **=**, наприклад, **a = b = 2**. Для наочності «внутрішнє» присвоювання можна записати в дужках: **a = (b = 2)** — дії будуть ті ж самі. Інструкція **a = b = 2;** задає в програмі ті ж самі дії, що й послідовність **b = 2; a = b;**. Довжину ланцюгів присвоювань не обмежено.

Інструкції та вирази присвоювання можуть бути частиною інструкцій інших різновидів (наприклад, інструкцій керування), описаних у наступних главах.

- Сукупність змінних, імена яких означено в програмі, називається ***пам'яттю програми***[7], а відповідність між іменами змінних та їх станами — ***станом пам'яті програми***. Присвоювання значення змінній означає ***зміну стану пам'яті програми***.
- Ім'я змінної є прикладом загальнішого поняття ***l-вираз***. Так називають вираз, який можна записати ліворуч (*left*) у виразі присвоювання. Він позначає змінну. Вираз, який позначає значення й записується праворуч (*right*) у виразах присвоювання, називається ***r-виразом***.

2.6.4. Ініціалізація змінних

Присвоїти початкове значення змінній можна в її означенні (***ініціалізувати змінну***), дописавши до імені змінної знак **=** і вираз. У виразі можуть бути константи та імена змінних (звичайно, якщо вони вже отримали значення).

Приклади

1. За виконання інструкції

```
int a = 44, b = a, c;
```

створені змінні **a** та **b** отримують значення **44**, а значення змінної **c** залишається невизначеним, тому використовувати її значення можна тільки після того, як вона його отримає.

2. За виконання інструкції

```
int i=j=k=0;
```

створюються змінні **i**, **j**, **k**, які відразу ж отримують значення **0**. ◂

[7] Далі ми побачимо, що у процесі виконання програми змінні можуть додаватися до її пам'яті або, навпаки, вилучатися.

7.3.2. Глибина рекурсії та загальна кількість рекурсивних викликів

З рекурсивними функціями пов'язано два важливих поняття — глибина рекурсії та загальна кількість викликів, породжених викликом рекурсивної підпрограми. Будемо відрізняти глибину рекурсії, на якій перебуває виклик, і глибину рекурсії, породжену викликом.

- ***Глибина рекурсії, на якій перебуває виклик підпрограми*** — це кількість рекурсивних викликів, розпочатих і незакінчених у момент початку цього виклику.
- ***Глибина рекурсії, породжена викликом*** — це максимальна кількість рекурсивних викликів, які розпочато й не закінчено після початку цього виклику.

Наприклад, у програмі «Ханойські вежі» виклик функції **tower** з висотою h перебуває на глибині рекурсії $n-h$ та породжує глибину $h-1$.

Коли виконується виклик функції, який перебуває на глибині рекурсії m, одночасно існує $m+1$ екземпляр локальної пам'яті функції. Кожен екземпляр займає ділянку певного розміру, тому збільшення глибини, як було сказано вище, може призвести до ***переповнення програмного стеку***.

- ***Загальна кількість рекурсивних викликів***, породжених викликом рекурсивної функції, — це кількість викликів, які виконано між його початком і завершенням.

Наприклад, у задачі «Ханойські вежі» кожен виклик задає одне перенесення диска та, якщо він не є найбільш заглибленим, породжує два рекурсивних виклики. Нам уже відомо, що перенесення вежі висотою n вимагає 2^n-1 перенесень дисків, тому між початком і закінченням виклику з аргументом n виконується 2^n-2 рекурсивних викликів.

Загальна кількість породжених викликів визначає тривалість виконання виклику. Як бачимо, кількість викликів у задачі «Ханойські вежі» за значень n уже порядку кількох десятків призводить до неприпустимо довгого виконання програми.

- Записуючи рекурсивну функцію, необхідно уміти оцінити можливу глибину рекурсії, розмір пам'яті виклику функції та загальну кількість рекурсивних викликів.

7.3.3. Приклад недоречного використання рекурсії

Повернемося до чисел Фібоначчі, які визначаються рекурентним співвідношенням $F_n = F_{n-1}+F_{n-2}$, $n\geq 2$, та початковими умовами $F_0 = 0$, $F_1 = 1$. За цим означенням дуже легко написати таку функцію обчислення числа Фібоначчі за його номером.

```
int f(int n) //pre: n>=0, інакше повертаємо -1
{  if (n<0) return -1;
  if (n==0) return 0;
  if (n==1) return 1;
  return f(n-2)+f(n-1);      // два виклики
}
```

ЦЕ ДУЖЕ ПОГАНА ФУНКЦІЯ!!! Її головний недолік: вона ***породжує абсолютно не потрібні багаторазові обчислення тих самих вели-***

стрижень 3, потім вежу з 2 на 3. При перенесенні вежі з 1 на 2 допоміжним буде стрижень 3, а при перенесенні з 2 на 3 — 1, причому ніяка інша послідовність дій неможлива. Отже, розв'язання задачі для вежі висотою *n* описано за допомогою розв'язання для вежі висотою *n*–1, тобто ***рекурсивно***.

Нехай **`diskMove(a,b)`** позначає перенос одного диска зі стрижня **`a`** на стрижень **`b`**, а **`tower(h,a,b,c)`** — перенесення вежі висотою **`h`** з **`a`** на **`b`** з використанням стрижня **`c`** як допоміжного. За **`h`** > 1 виконання **`tower(h,a,b,c)`** буде таким.

```
tower(h-1,a,c,b);
diskMove(a,b);
tower(h-1,c,b,a)
```

За **`h`** = 1 переносять тільки один диск, тобто виконують **`diskMove(a,b)`**. Отже, очевидною є така програма.

```
#include <iostream>
using namespace std;
void diskMove(int diskFrom, int diskTo)
{ cout << diskFrom << "->" << diskTo << ' ';
  return;
}
void tower(int height, int from,int to, int via)
//height - висота, from - з, to - на, via - через
{ if(height==1) diskMove(from,to);
  else
  { tower(height-1,from,via,to);
    diskMove(from,to);
    tower(height-1,via,to,from);
  }
}
int main()
{ int n;
  cout << "Hanoi towers\n";
  cout << "Enter a number of disks>";
  cin >> n; if (n<1) n=1;
  tower(n,1,3,2);
  cout<<endl;
  system("pause");return 0;
}
```

prog018.cpp

Визначимо кількість переносів дисків у вигляді функції $s(n)$, де n — висота вежі. Очевидно, що $s(1) = 1$ і $s(n) = 2 \cdot s(n-1)+1$. Як бачимо, $s(2) = 3$, $s(3) = 7$, $s(4) = 15$ тощо. Кожне наступне значення подвоюється з додаванням 1, тому неважко переконатися, що $s(n) = 2^n-1$. Значення $s(64)$ приблизно дорівнює 10^{19}. Якщо припустити, що ченці переносять один диск щосекунди, то для перенесення такої вежі потрібно більше 10^{12} років! У ченців ще багато роботи…

- Рекурсивна функція без жодного циклу може легко приховати величезні обсяги обчислень. Як і будь-який потужний засіб, рекурсія вимагає обережного використання.

Вправа 2.14. Коло на площині можна задати його центром та радіусом. Написати програму, в якій змінні, що зображують коло на площині, оголошені та ініціалізовані для зображення кола з центром у точці (1,5; 2,3) і радіусом 3,5. Програма має вивести координати центра кола та його радіус.

2.7. Поняття вхідного потоку

2.7.1. Поняття потоку

У мові C++ основним поняттям уведення та виведення даних є ***потік*** — послідовність символів або інших даних. У програмі потік зображує фізичний файл на зовнішньому носії даних (диску, клавіатурі або екрані монітора), тобто фізичний файл «видно в програмі» як потік даних. Операції обміну даних з файлом представлено в програмі як операції добування даних з потоку або дописування їх до нього

У програмуванні існує поняття ***стандартних файлів*** уведення та виведення — як правило, ними є клавіатура та екран. У C++-програмі їм відповідають ***стандартний потік уведення*** з ім'ям **`cin`** та ***стандартний потік виведення*** з ім'ям **`cout`**. Отже, імена **`cin`** і **`cout`**, означені у файлі **`iostream`**, насправді позначають не клавіатуру та екран, а потоки, що відповідають цим пристроям у програмі. На початку виконання програми обидва потоки **`cin`** і **`cout`** порожні.

З засобів обробки потоків розглянемо лише операції ***введення*** **`>>`**, або ***вставки*** (*insertion*), даних з вхідного потоку, та ***виведення*** **`<<`**, або ***вставки***, даних у вихідний потік. Розгляд інших засобів виходить за межі цієї книги.

2.7.2. Операція вставки з потоку

Вираз із операцією введення (вставки з потоку) **`>>`** має вигляд **`cin >> ім'я-змінної`**. Додавши в кінці виразу символ **`;`**, маємо інструкцію введення.

Виконуючи операцію введення, комп'ютер забирає з вхідного потоку послідовність непорожніх символів, за якою створює відповідне значення й присвоює змінній. Усі порожні символи пропускаються. Якщо у вхідному потоці немає непорожніх символів («потік порожній»), комп'ютер очікує на його поповнення. Потік поповнюється, коли людина набирає на клавіатурі деяку послідовність символів (вони з'являються на екрані) та натискає на клавішу **`Enter`**.

- Згідно з типами змінних, в які вводяться значення, операція **`>>`** розбиває вхідний потік на ***лексеми*** (послідовності непорожніх символів) та перетворює їх на значення певних типів (***інтерпретує лексеми***). Отже, операцію **`>>`** можна розглядати як ***інтерпретацію вхідного потоку символів***.

Значенням виразу введення є той потік, з якого «забрано» символи. До нього знову можна застосувати операцію **`>>`**, тобто можливі вирази вигляду

`cin >> ім'я-змінної_1 >> ім'я-змінної_2`
та аналогічні їм з більшою кількістю змінних. Наведений вираз еквівалентний виразу
`(cin >> ім'я-змінної_1) >> ім'я-змінної_2`
й задає послідовне введення значень двох змінних.

Приклад. Якщо є змінні `char c1, c2;` і за виконання `cin >> c1 >> c2;` натиснути клавіші **Space**, **W**, **Space**, **Space**, **3**, **Q**, **Enter**, то змінні отримають значення `'W'` та `'3'`, а **Q** залишиться у вхідному потоці. ◂

- За стандартних налаштувань операція **>>** пропускає всі порожні символи вхідного потоку.

Вправа 2.15. Нехай за виконання програми користувач послідовно натиснув на клавіші **5**, **Enter**, **7**, **Space**, **9**, **Enter**. Що він побачить на екрані?

```
#include <iostream>
using namespace std;
int main() {
   int i, j, k;
   cout<<"Enter 2 integers:"; cin>>i>>j;
   cout<<"Enter an integer:"; cin>>k;
   cout<<"Entered integers: "<<i<<", "<<j<<", "<<k<<endl;
   system("pause"); return 0;
}
```

Вправа 2.16. Чи може змінна `char c1` за виконання `cin>>c1` отримати значення `' '`?

За стандартних налаштувань, утворюючи за вхідною послідовністю числове значення, операція **>>** розглядає вхідну послідовність як ***десяткове*** зображення числа. Дійсні числа можна задавати в нормалізованій формі, а також без дробової частини. У дробовій частині використовується ***крапка***, а не кома. Для цілих чисел дробова частина та порядок не допускаються; задане число повинно належати діапазону можливих значень типу, який має змінна, інакше введення може мати непередбачувані наслідки.

Приклад. Якщо є змінна `int k;` і за виконання `cin >> k;` натиснути клавіші **0**, **1**, **1**, **Enter**, то значенням **k** буде **11**. Якщо є змінна `double x;` і за виконання `cin >> x;` натиснуто клавіші **8**, **Enter**, то **x** отримує значення **8.0**, а якщо **1**, **e**, **2**, **Enter**, — значення **100.0**. ◂

Під час уведення з клавіатури помилки найчастіше трапляються з числовими змінними. Якщо за виконання виразу введення сталася помилка, то вхідний потік переходить у ***стан помилки***, в якому ***виконання подальших операцій уведення неможливе***. За наявності помилки значення змінної, в яку мало відбутися введення, ***не змінюється***. Отже, якщо змінну не ініціалізовано й значення їй не присвоєно, то її значення залишається «випадковим сміттям»!


```
void badFunc(int x)
{ if(x==2) {cout << x; return;} // повертаємося
  badFunc(x-2);                 // заглиблюємося
}
```

За виконання виклику `badFunc(6)` відбуваються рекурсивні виклики з аргументами **4** та **2**. За `x==2` заглиблення в рекурсію немає, тому виводиться **2**, а потім послідовно закінчуються виклики з аргументами **2**, **4** та **6**. Проте виклик `badFunc(5)` призведе до рекурсивних викликів з аргументами **3**, **1**, **−1**, **−3**, …, в яких умова повернення `x==2` ніколи не стане істинною, тому виконання рекурсивних викликів ***переповнить програмний стек***.
◂

У цій главі розглядається тільки так звана ***пряма*** рекурсія, коли функція містить виклики самої себе. У програмуванні також використовується ***непряма*** рекурсія, коли функції містять взаємні виклики.

Вправа 7.1. Що буде виведено на екран, якщо за виконання програми ввести а) 10; б) 7; в) 5?

```
#include <iostream>
using namespace std;
void func (int n){
  if (n>7) func(n-1);
  cout<<n<<" ";
  return;
}
int main()
{ int n;
  cout <<"Enter one integer\n"; cin >> n;
  func(n);
  cout<<endl; system("pause");return 0;
}
```

7.3. «Підводні камені» рекурсії

7.3.1. Приклад: ханойські вежі

На дошці — три стрижні: 1, 2, 3. На першому розміщено вежу з n дисків; нижній диск має найбільший діаметр, а діаметр кожного наступного менший за діаметр попереднього. За один хід із будь-якого стрижня можна взяти верхній диск і перемістити на інший стрижень, але дозволено класти диск лише на дошку або на диск більшого діаметра. Треба перемістити усю вежу зі стрижня 1 на стрижень 3.

Ця гра називається «Ханойські вежі». За легендою, цю гру почали понад тисячу років тому ченці в одному монастирі поблизу Ханоя у В'єтнамі. У ченців було $n = 64$ диски. Коли вони закінчать гру, настане кінець Всесвіту. Розв'язком цієї гри-задачі є послідовність переносів дисків. Напишемо програму виведення позначень цих переносів.

Щоб перенести вежу з n дисків зі стрижня 1 на стрижень 3, необхідно перенести вежу з $n-1$ диска на стрижень 2, потім перенести нижній диск на

'0' до '9' та від 'A' до 'Z'. Для перетворення цифри в її символьне зображення використовується допоміжна функція **digit**.

```
#include <iostream>
using namespace std;
char digit(int v) // цифра за значенням v
//pre: 0<=v<=35
{ if(v<10) return char(int('0')+v);
  else return char(int('A')+v-10);
}
void outNP(int n, int p)  // виведення p-кових цифр числа n
//pre: n>=0 && p>=2
{ if(n>=p) outNP(n/p, p); // заглиблення
  cout << digit(n%p);     // виводиться остання цифра
  return;
}
int main() {
  int n, p;
  cout <<"Enter one nonnegative integer and \n"
       <<"one integer between 2 and 36 >";
  cin >> n >> p;
  if (n>=0 && 2<=p && p<=36) outNP(n,p);
  else cout<<"You enter wrong numbers";
  cout<<endl; system("pause");return 0;
}
```

prog017.cpp

Для перевірки програми задайте число 2147483647 (найбільше число типу **int**) та основи 2, 8, 16, 32. Відповідними результатами мають бути **11**…**1** (31 «**1**»), **17777777777**, **7FFFFFFF**, **1VVVVVV**. Перевірте також декілька чисел з максимальною основою 36, наприклад, 35, 71 та 1295 (результатами мають бути **Z**, **1Z** та **ZZ**). Також слід перевірити некоректні числа та основи системи числення.

- З кожним рекурсивним викликом зайнята частина програмного стека збільшується, а з закінченням виклику — зменшується. Розмір стеку обмежений, тому можлива ситуація (особливо за виконання рекурсивних функцій), коли пам'яті в стеку забракне й програма завершиться аварійно.
- У рекурсивній функції обов'язково має бути ***умова***, за істинності якої відбувається повернення з виклику (див. приклади вище). Ця умова визначає «***дно рекурсії***», яке під час виконання функції ***обов'язково має досягатися***, інакше виклики призведуть до переповнення програмного стека або інших непередбачуваних наслідків.

Приклади

1. Для функції **outNP** з попередньої програми кожен виклик зменшує значення параметра **n**, тому дно рекурсії в ній досягається.

2. Розглянемо функцію з ***невдалою умовою*** повернення з рекурсії.

Вправа 2.17. Запустіть на виконання програму, наберіть **100e777** та натисніть **Enter**. Що буде надруковано на екрані? Чи буде комп'ютер очікувати введення цілого числа? Поясніть побачені результати.

```
#include <iostream>
using namespace std;
int main() {
   int i; double x;
   cout<<"Enter one real:"; cin>>x;
   cout<<"Enter one integer:"; cin>>i;
   cout<<"Entered numbers: "<<x<<", "<<i<<endl;
   system("pause"); return 0;
}
```

Вправа 2.18. Що зміниться в попередній вправі, якщо замість інструкцій **int i; double x;** записати інструкції **int i=0; double x=0;**?

Контрольні запитання

2.1. Які обов'язкові частини повинна мати програма мовою C++?
2.2. З яких двох частин складається функція мови C++?
2.3. Які основні групи символів є в алфавіті мови C++?
2.4. Які види лексичних одиниць є в мові C++?
2.5. Що таке ім'я в мові C++?
2.6. Що таке змінна? Що таке змінна в програмуванні?
2.7. Що таке тип даних? Що таке носій та сигнатура типу?
2.8. Що таке константа? Які різновиди констант є в мові C++?
2.9. Що таке вираз та інструкція присвоювання?
2.10. У чому полягає семантика інструкції присвоювання?
2.11. Що таке пам'ять програми та стан пам'яті програми?
2.12. Що є семантикою послідовності інструкцій присвоювання?
2.13. Що таке потік значень?

Задачі

2.1. Напишіть програму, яка виводить таке повідомлення **Each algorithm is a list of well-defined instructions for completing a task. Starting from an initial state, the instructions describe a computation that proceeds through a well-defined series of successive states, eventually terminating in a final ending state.**
Зверніть увагу на розташування тексту по рядках. Щоб отримати охайне повідомлення, за необхідності починайте виведення з нового рядка.

2.2. Напишіть програму, яка друкує охайне повідомлення такого змісту **Цю програму створив студент** … **групи** … **у** … **році.** Замість … вставте особисті дані (прізвище, ім'я та по-батькові, номер групи та рік написання програми).

2.3. Підготуйте реферат на тему: «Способи запису символьних констант та символів у рядках у мові C++».

Глава 3. Прості математичні обчислення

3.1. Арифметичні операції

3.1.1. Операції, операнди, вирази

Обчислювальні дії в програмуванні називаються ***операціями***. Операції застосовуються до ***операндів***, тобто значень. Застосування операцій до значень описується у вигляді ***виразу*** (*expression*). Послідовність застосування операцій називається ***обчисленням*** виразу й має результатом ***значення виразу***. Операції у виразі позначаються знаками (***операторами***), а значення — константами та іменами змінних. У виразі також можуть бути дужки, які визначають порядок застосування операцій. Найпростішими виразами є вирази, що не містять операцій, тобто константи та імена змінних.

Приклад. Вираз **`2+2`** означає: додаються 2 й 2 і значенням виразу є 4; вираз **`2*radius`** — 2 множиться на значення змінної **`radius`** і значенням виразу є подвоєне значення цієї змінної; **`1+2*3`** — множаться 2 і 3, отриманий добуток 6 додається до 1 і значенням є 7; **`(1+2)*3`** — додаються 1 і 2, їх сума 3 множиться на 3 й значенням є 9. За двома останніми виразами видно, як дужки впливають на порядок операцій. ◂

- Вираз має подвійну семантику — ***послідовність операцій*** з операндами, а також ***значення***, що є результатом цієї послідовності.

Операція з одним або кількома значеннями, результатом якої є число, називається ***арифметичною***. Спочатку розглянемо тільки деякі з багатьох арифметичних операцій мови C++.

Операції ***додавання***, ***віднімання***, ***множення*** та ***ділення*** мають знаки відповідно **`+`**, **`-`**, **`*`**, **`/`**. Результатом операції з ***цілими*** числами є ***ціле*** число, з ***дійсними*** — ***дійсне***. Наприклад, значенням виразу **`4/2`** є ціле число **`2`**, а виразу **`4.0/2.0`** — дійсне **`2.0`**. Застосування операцій до різнотипних значень розглянуто нижче.

Знак **`-`** позначає як двомісну операцію віднімання, так і одномісну операцію «мінус»: **`-32768`**, **`-(2+3)`**. Знак **`+`** також може позначати одномісну операцію.

Результатом ділення **`/`** цілих чисел є ***ціла частка*** від ділення з остачею, наприклад, вираз **`7/3`** має значення **`2`**. ***Цілу остачу*** від ділення обчислює операція **`%`**: значенням виразу **`7%3`** є **`1`**. Зауважимо: знак остачі збігається зі знаком діленого, наприклад, обидва вирази **`-7%3`** та **`-7%-3`** мають значення **`-1`**, а вираз **`7%-3`** — значення **`1`**.

Результатом ділення дійсних чисел є число в його дійсному зображенні, наприклад, значенням виразу **`7.0/3.0`** є деяке наближення до числа **`2.33`**..., а виразу **`6.0/3.0`** — число **`2.0`**.

- Операція **`%`** до дійсних чисел ***незастосовна***.
- Виконання операції **`/`** або **`%`** з дільником 0 призводить до аварійного завершення програми.

кінець уведення. Задача: прочитати числа та видати їх у зворотному порядку (кінцевий 0 не виводити).

Перше число треба вивести останнім, друге — передостаннім, і так далі. Отже, для обробки входу потрібно прочитати перше число і, якщо це не 0, то ***в такий самий спосіб*** обробити решту входу й потім вивести перше число. А якщо прочитано 0, то обробку вхідних даних закінчено. Для цього в програмі використовується рекурсивна функція **`outReverse`**.

```
#include <iostream>
using namespace std;
void outReverse()
{ int n;
  cin >> n;
  if (n==0) return;  // уведено 0 – повернення
  outReverse();      // уведено не 0 – заглиблення
  // після повернення з рекурсивного виклику
  cout << n << " ";
  return;
}
int main(){
  cout << "Enter some integers, tha last should be 0\n";
  outReverse();
  cout<<endl;
  system("pause"); return 0;
}
```

prog016.cpp

Уведене число записується в ***локальну*** змінну **`n`** функції; кожен виклик додає до програмного стеку новий екземпляр **`n`**. Після введення 0 виклики закінчуються в порядку, зворотному до того, в якому починалися. Звідси й значення локальних змінних **`n`** виводяться у зворотному порядку. У головній функції потрібен лише виклик функції **`outReverse`**.

2. Прочитати невід'ємне число n типу **`int`** (у звичайному десятковому записі) та основу системи числення p, $2 \le p < 37$. Вивести p-ковий запис числа. Значення цифр 10, 11, 12, …, 35 представити цифрами **`A`**, **`B`**, **`C`**, …, **`Z`**.

Значення молодшої цифри в p-ковому записі числа є остачею від ділення числа на p. Далі, узявши замість числа частку від його ділення на p, ***так само*** отримаємо значення наступної цифри. Так можна діяти, доки не залишиться число менше p. Проте цифру, ***отриману першою***, треба ***вивести останньою***, тому запам'ятаємо значення молодшої цифри, далі рекурсивно виведемо старші цифри, а потім виведемо молодшу цифру.

Головна функція вводить два числа — число n та основу системи числення p, за їх коректності викликає функцію **`outNP`** виведення p-кових цифр числа n цифр, інакше повідомляє про помилку. У функції виведення цифр **`outNP(n, p)`** заглиблюємося в рекурсію, якщо $n \ge p$ (число n має не менше двох p-кових цифр), інакше виводимо останню цифру. Можливим значенням цифр від 0 до 35 мають відповідати цифри — символи від

Виконання виклику **f(g(a))** починається з обчислення виразу **g(a)** — аргументу для параметра **f.x**, тому починається виклик функції **g** з аргументом **a**. Цей аргумент підставляється за посиланням, тому імена **g.x** і **main.a** позначають одну й ту саму змінну. З виклику функції **g** повертається **6** і присвоюється параметру **f.x**. Пам'ять функції **g** звільняється. Тільки тепер виконуються інструкції функції **f**. Після їх закінчення в головну функцію повертається значення **7**. Пам'ять виклику функції **f** звільняється. У головній функції виводиться значення **7**, отримане з виклику, а потім значення змінної **main.a**.

◂

7.2. Знайомство з рекурсією

7.2.1. Поняття та приклади рекурсії

Означення називається ***рекурсивним***, якщо воно задає елементи певної множини за допомогою інших елементів цієї ж множини. Об'єкт, заданий рекурсивним означенням, також називається ***рекурсивним***, а використання таких означень — ***рекурсією***.

Приклади

1. Значення функції «факторіал» можна задати початковим елементом $0! = 1$ та рекурентним співвідношенням $n! = n\cdot(n-1)!$. Усі елементи цієї множини, крім першого, означаються рекурсивно. Взагалі, будь-яке рекурентне співвідношення разом з початковими умовами є прикладом рекурсивного означення.

2. Арифметичні вирази з константами, знаком операції **+** та дужками опишемо так:

а) константа є арифметичним виразом;

б) якщо E і F є арифметичними виразами, то $(E)+(F)$ також є виразом;

в) інших арифметичних виразів, крім утворених за пп. а) та б), не існує.

Описаними виразами є, наприклад, 1, 2, (1)+(2), ((1)+(2))+(1).

3. Інструкції мови С++, що задають розгалуження та цикли, самі містять інструкції. Отже, представники поняття «інструкція» є рекурсивними. ◂

У рекурсивному означенні не повинно бути «зачарованого кола», коли в означенні об'єкта використовується він сам або інші об'єкти, задані за його допомогою.

Приклад. Змінимо означення функції «факторіал»: $n! = n\cdot(n-1)!$ за $n > 0$, $0! = 1!$. Значення функції від 1 виражається через її ж значення від 0, яке, у свою чергу, — через значення від 1. За цим «означенням» не можна дізнатися, яким числом є 1!. ◂

7.2.2. Прості приклади рекурсивних функцій

Функції, що містять виклики самих себе, як і самі ці виклики, називаються ***рекурсивними***. Виклик рекурсивної функції виконується так само, як і виклик будь-якої функції.

Приклади

1. На клавіатурі набираються цілі числа, не рівні нулю. Поява 0 означає

Інструкція вигляду **cout << вираз;** обчислює та виводить значення виразу.

Вправи

3.1. Що буде виведено на екран за виконання інструкцій?

```
cout<<9/5<<' '<<-9/5<<' '<<9/-5<<' '<<-9/-5<<endl;
cout<<9%5<<' '<<-9%5<<' '<<9%-5<<' '<<-9%-5<<endl;
cout<<9./5.<<endl;
```

3.2. Що буде виведено на екран за виконання інструкцій?

```
cout<<7/3<<' '<<1/6<<endl;
cout<<7./3.<<' '<<1./6.<<endl;
```

3.3. Припустимо, що значення дійсної змінної **length** відповідає довжині будівлі в міліметрах. Написати вираз, який задає довжину будівлі в метрах.

3.4. Нехай значення цілої змінної **sizeOfFile** задає розмір файлу в байтах. Написати вираз, значенням якого є розмір файлу в Кбайтах.

3.5. Нехай **v** — ім'я цілої змінної з невід'ємним значенням. Написати вираз, який обчислює: а) значення молодшої десяткової цифри числа **v**; б) значення молодшої двійкової цифри числа **v**.

Одномісна операція **sizeof** обчислює цілу ***кількість байтів***, зайнятих її операндом (дані типу **char** займають 1 байт, типу **int** — 4 байти, **double** — 8). Отже, за виконання інструкції

```
cout<< sizeof 'A'<<' '<< sizeof 1 <<' '<< sizeof 0.0<<endl;
```

побачимо **1 4 8**.

3.1.2. Старшинство операторів та порядок виконання операцій

Мова С++, в основному, відповідає угодам математики про порядок застосування операцій у виразах. Це дозволяє не записувати зайві дужки, наприклад, **1-2*3** означає те ж, що й **1-(2*3)**. На порядок обчислення виразу за відсутності дужок впливає ***старшинство*** (*precedence*), або ***пріоритет***, операторів: якщо поруч із позначенням операнда записано два оператори, то спочатку виконується операція, що відповідає старшому оператору (з вищим пріоритетом). Наприклад, пріоритети ***** і **/** однакові й вищі за **+** і **-**. Одномісні оператори старші за двомісні, а двомісні оператори *****, **/**, **%**, **+**, **-** старші за всі інші двомісні, у тому числі присвоювання.

Окрім пріоритетів, оператори мають властивості право- або лівобічного зв'язування. У мові С++ усі двомісні оператори, окрім присвоювань, мають властивість ***лівобічного зв'язування***: якщо ліворуч і праворуч від позначення операнда записано знаки операцій з однаковим старшинством, то операнд зв'язується з оператором, вказаним ліворуч (ця операція застосовується спочатку). Докладніше про пріоритети операторів та властивості зв'язування див. у додатку Б.

Приклади

1. Значенням виразу **4+7/5** є **5**, оскільки спочатку обчислюється **7/5** з

результатом **1**, а потім **4+1** з результатом **5**. Значенням виразу **(4+7)/5** є **2**, оскільки спочатку обчислюється операція в дужках **(4+7)** — її значення 11, — а потім **11/5** з результатом **2**.

2. У виразі **sizeof 2.0+4** обчислюється **sizeof** з результатом **8**, потім додається **4**.

3. Значення виразу **4-3-2** дорівнює **-1**, оскільки спочатку обчислюється **4-3**, тобто **1**, а потім **1-2**; у виразі **2*7%8** спочатку обчислюється **2*7** (це **14**), потім **14%8**, тобто **6**.

4. Нехай дійсні змінні **a**, **b**, **c** зображують коефіцієнти квадратного рівняння $ax^2+bx+c=0$. Дискримінант рівняння визначається виразом **b*b-4*a*c**. Присвоїмо його дійсній змінній **d**: **d=b*b-4*a*c**.

- Пріоритети операторів дозволяють не записувати зайві дужки, але ***зловживати цим не слід***. Інколи зайва пара дужок значно підвищує зрозумілість запису. Так, у виразі **sizeof 2.0+4** пропуск між **sizeof** та **2.0** провокує людину спочатку (помилково) обчислити **2.0+4**, а потім **sizeof**. Проте **sizeof(2.0)+4** є очевидним.

Вправи

3.6. Що буде виведено на екран за виконання інструкцій?

```
cout<<1+4/2<<' '<<(1+4)/2<<endl;
cout<<2*4%7<<' '<<2*(4%7)<<endl;
cout<<51/6%7<<' '<<51/(6%7)<<endl;
cout<<2*(7%8)<<' '<<12/6%8<<' '<<5-3%2<<endl;
```

3.7. Що буде виведено на екран за виконання інструкцій?

```
cout<<4*6/8<<' '<<4/8*6<<endl;
```

3.8. Що буде виведено на екран за виконання інструкцій?

```
cout<<(-3+5)*(2%7/3+4*2)<<endl;
```

3.9. Що буде виведено за такими інструкціями?

а)
```
int a=2, b=3;
cout<<"a*b="<<a*b<<";\n***"<<endl;
```

б)
```
int a=3, b; cout<<a*a<<' '<<a+4<<'\n';
```

3.10. Нехай **a**, **b**, **c** — дійсні змінні, що позначають довжини сторін трикутника. Запишіть вираз, що задає периметр, та інструкцію, яка присвоює периметр змінній **p**.

3.1.3. Бібліотечні математичні функції та константи

Деякі операції з числами позначаються ***викликами функцій***, тобто у вигляді **f(**…**)**, де **f** позначає певне ім'я. Розглянемо дві функції, означені в усіх реалізаціях мови C++. Для використання цих функцій у програмі необхідно підключити модуль **cmath**.

```
#include <cmath>
```

Одномісна функція **sqrt** обчислює квадратний корінь свого невід'ємного ***дійсного*** операнда, тобто значенням виразу **sqrt(2.0)** є при-

Під час виконання викликів функцій ділянки автоматичної пам'яті виділяються й звільняються за принципом «останньою зайнято — першою звільнено». Якщо складати аркуші паперу в стос і брати їх тільки згори, то аркуш, що потрапив до стосу останнім, забирають першим. Англійською мовою стос називається *stack* (стек), а кладуть і беруть аркуші за принципом «*Last In — First Out*» (LIFO), тобто «останнім прийшов — першим пішов». Тому автоматичну пам'ять програми також називають ***програмним стеком***.

Аналогічно набій, заштовхнутий у магазин автомата останнім, вилітає першим, а заштовхнутий першим (він на дні магазина) вилітає останнім. Можна вистрілити набій з магазина й на його місце додати новий. Так само, коли послідовно виконуються два виклики функцій у тілі функції, то для другого виклику виділяється пам'ять, звільнена після першого.

Приклад. Розглянемо імітування такої програми.

```
#include <iostream>
using namespace std;
int f(int x)
{ return ++x; }
int g(int& x)
{ return x/=2;  }
int main(){
  int a=12;
  cout << f(g(a)) << ' ';
  cout << a << endl;
  system("pause"); return 0;
}
```

prog015.cpp

Імітуючи програму, до імен змінних, оголошених у функціях, приписуємо імена цих функцій, наприклад **main.a** або **f.x**. Параметр-посилання та ім'я змінної, що є відповідним аргументом у виклику, позначають одну й ту саму змінну (хоча, враховуючи технічні подробиці, це не зовсім так). Через **f.r** та **g.r** позначимо значення, що повертаються з функцій.

Що виконується	Змінні всіх функцій		
	main.a		
int a=12	12		
початок **out<<f(g(a))**	12		
виклик **f(g(a))**	12	**f.x**	
початок **f.x=g(a)**	12	?	
виклик **g(a)**	**g.x**	?	
g.x/=2	6	?	
g.r=g.x	6	?	
повернення з **g**	6	?	**g.r**=6
закінчення **f.x=g(a)**	6	6	
++f.x	6	7	
f.r=f.x	6	7	
повернення з **f**	6	7	**f.r**=7
виведення **f(g(a))**: **7**	6		
виведення **a**: **6**	6		

Глава 7. Рекурсія

Одним зі способів опису об'єктів є рекурсія. Рекурсивними можуть бути правила, що описують структуру виразів деякої мови, означення математичних функцій, алгоритми тощо. Рекурсія є одним з фундаментальних понять програмування й математики. Завдяки їй різноманітним об'єктам можна дати зрозумілий і компактний опис.

У цій главі представлено рекурсивні функції мови C++. Проте спочатку, щоб краще розуміти їх виконання, розглянемо, як взагалі виконуються виклики функцій.

7.1. Виконання виклику функції

7.1.1. Пам'ять виклику функції

Сукупність змінних, які утворюються під час виклику функції (підпрограми), має назву ***пам'ять виклику функції***, або не зовсім точно ***локальна пам'ять*** функції. Змінні в цій пам'яті називаються ***локальними*** й відповідають ***параметрам*** та ***іменам змінних***, означеним у тілі функції.

Локальна пам'ять функції містить ще один елемент — посилання на місце, з якого має виконуватися програма після закінчення виклику. Наприклад, головна функція на с. 74, яка містить три виклики функції **swap**, після цих викликів продовжується трьома різними інструкціями. Місце продовження називається ***точкою повернення з функції***, а посилання на неї зберігається під час виконання виклику функції.

7.1.2. Огляд процесу виконання виклику

1. Виділяється пам'ять для точки повернення та параметрів функції. Посилання на точку повернення з функції запам'ятовується.

2. Обчислюються значення аргументів для параметрів-значень, посилання на пам'ять аргументів для параметрів-посилань. Відбувається підстановка аргументів.

3. Виділяється пам'ять, відповідна локальним іменам змінних[14].

4. Виконуються інструкції тіла функції до інструкції повернення.

5. Якщо підпрограма не є **void**-функцією, то значення, що повертається з її виклику, копіюється до пам'яті функції, яка містила виклик.

6. Функція, яка містила виклик, продовжується з точки повернення.

Змінні в локальній пам'яті функції не відповідають іменам у функції, яка містила виклик, тобто ця пам'ять ***недоступна*** після того, як виклик закінчено. Вона ***вважається звільненою***; її можна використовувати для наступного виклику цієї ж або іншої функції.

- Відбувається так зване ***логічне звільнення***, тобто зміст локальної пам'яті не змінюється, але стає недоступним.

7.1.3. Автоматична пам'ять, або програмний стек

Ділянки пам'яті викликів функцій утворюються й звільняються в спеціальній області пам'яті процесу виконання програми — ***автоматичній пам'яті***. Називається вона так тому, що за виконання викликів функцій пам'ять виділяється й звільняється без явних вказівок у програмі, написаній мовою високого рівня, тобто «автоматично».

[14] Окрім локальних статичних змінних (див. розділ 8.2)

близно **1.41421**, а **sqrt(4.0)** — **2.0**. До цілих чисел функція ***незастосовна***.

Двомісна функція **pow** обчислює дійсний степінь, основою якого є перший операнд, показником — другий. Наприклад, значенням виразу **pow(2.0, 3)** є **8.0**, виразу **pow(2, 0.5)** — приблизно **1.41421**. Результатом функції **pow** завжди є дійсне значення.

Функція **log** обчислює натуральний логарифм свого додатного дійсного аргументу, функція **log10** — десятковий логарифм. Застосування функцій до цілих аргументів є помилковим.

Функція **fabs** обчислює дійсне значення $|x|$ за дійсним аргументом x. Функція **abs** із бібліотеки **cstdlib** обчислює ціле значення $|x|$ за цілим аргументом x; якщо аргумент дійсний, обчислене значення може відрізнятися від математичного.

Приклади

1. Корінь з невід'ємного дискримінанта квадратного рівняння з дійсними коефіцієнтами **a**, **b**, **c** можна обчислити виразом **sqrt(b*b-4*a*c)**, а дійсні корені рівняння — виразами **(-b-sqrt(b*b-4*a*c))/(2*a)** та **(-b+sqrt(b*b-4*a*c))/(2*a)**.
Дужки в знаменнику обов'язкові. Якщо їх не записати, відбудеться не ділення, а множення на **a**.

2. Вираз **pow(b*b-4*a*c,0.5)** означає обчислення квадратного кореня з **b*b-4*a*c**, вираз **pow(b,1.0/3.0)** — обчислення кубічного кореня з **b**, а обидва вирази **pow(2.0, 5)** та **pow(2, 5.0)** означають піднесення дійсного числа **2.0** у степінь **5**. Зверніть увагу: вираз **pow(2, 5)** з двома цілими аргументами є помилковим.

3. Значенням **log10(2.0)** є (наближено) 0.30103, значенням **log(1)** — дійсне 0.

4. Значенням **fabs(-2.0)** є дійсне **2.0**, значенням **abs(-2)** — ціле **2**. ◂

Вправи

3.11. Написати вираз мови C++, відповідний математичному виразу
а) $\sqrt{a^2+b^2}$; б) $(a+b)^{1/7}$; в) $\sqrt[3]{a^{12}+b^{12}}$; г) $\left|\sqrt{a}-\sqrt{b}\right|$.

3.12. У математиці виконується співвідношення $\log_x y = \ln y / \ln x$. Написати вираз мови C++, відповідний математичному виразу $\log_x y$.

3.13. Написати програму, яка виводить значення $\sqrt{2}$.

3.14. Написати програму, яка виводить значення $(2{,}5)^{1{,}6}$.

3.15. Написати послідовні інструкції присвоювання:
а) змінній **d** значення математичного виразу b^2-4ac, а **x1** та **x2** — вирази $\frac{-b-\sqrt{d}}{2a}$ та $\frac{-b+\sqrt{d}}{2a}$;

б) змінній **p** значення математичного виразу $(a+b+c)/2$, а змінній **s** — виразу $\sqrt{p(p-a)(p-b)(p-c)}$.

У стандарті мови C++ відсутні математичні константи, зокрема ті, що позначають числа $\pi = 3.141593\ldots$ та $e = 2.7182818\ldots$. Натомість у бібліотеці **cmath** означено константи[8] з іменами **M_PI** (число π), **M_PI_2** ($\pi/2$), **M_PI_4** ($\pi/4$), **M_1_PI** ($1/\pi$), **M_E** (число e), **M_LN2** (ln 2), **M_LN10** (ln 10) та деякі інші. Щоб користуватися ними, необхідно перед підключенням бібліотеки **cmath** записати директиву **#define _USE_MATH_DEFINES** (define — означити).

Приклад. Програма

```
#include <iostream>
#define _USE_MATH_DEFINES
#include <cmath>
using namespace std;
int main() {
   cout<<"pi="<<M_PI<<endl;
   cout<<"e="<<M_E<<endl;
   cout<<endl;system("pause"); return 0;
}
```

prog004.cpp

виводить значення математичних констант π та e. ◂

Вправа 3.16. Написати програму, яка виводить на екран суму чисел π та e.

Бібліотеки систем програмування мовою C++ містять різноманітні константи та численні підпрограми, що реалізують математичні та інші функції. Зауважимо: склад бібліотек у різних середовищах може бути різним, тому вичерпну інформацію про вміст бібліотек може дати лише довідка в конкретному середовищі або самі бібліотечні файли. Докладніше про зазначені та деякі інші функції сказано в Додатку В.

- ***Функція*** в мові ***C++*** — це частина програми, оформлена в спеціальний спосіб. Якщо програма описує дії з розв'язання деякої задачі, то функція описує дії з розв'язання деякої частини цієї задачі, тобто ***підзадачі***. Цей термін буде уточнено в главі 5.

3.1.4. Складені присвоювання

Вирази присвоювання дуже часто мають вигляд, наприклад, **a=a+b**, **a=a*b** тощо, тобто збільшують значення змінної **a** на величину **b**, множать на **b** тощо. Такі присвоювання можна задавати за допомогою спеціальних операторів у скороченій формі: **a+=b**, **a*=b**, які називають складеними (*compound*) присвоюваннями. В якості **b** можна записати будь-який вираз відповідного типу. Значенням виразу присвоювання є «нове» зна-

[8] У деяких версіях мови C++ ці константи замінено відповідними бібліотечними функціями.

6.6. Описати загальний вигляд та порядок виконання інструкції циклу **for**.

6.7. Скільки разів обчислюється початковий вираз за виконання циклу **for**?

6.8. Чи будь-який циклічний процес можна запрограмувати, з різних форм циклів використовуючи тільки одну?

6.9. Назвіть головні відмінності між префіксними та постфіксними операціями збільшення (зменшення).

6.10. Наведіть приклад співвідношення, яке не є рекурентним.

Задачі

6.1. Протабулювати (див. вправи 6.8, 6.10 на с. 83) функції $f(x)$, $S(x)$ та $S(x)$-$f(x)$ на відрізку $[a; b]$ з кроком h, де $a_0<a$, $b<b_0$ та

а) $a_0 = -0.9$, $b_0 = 0.9$, $f(x) = \operatorname{arctg} x$, $S(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n x^{2n+1}}{2n+1}$;

б) $a_0 = -0.9$, $b_0 = 0.9$, $f(x) = \operatorname{ch} x$, $S(x) = \sum_{n=0}^{+\infty} \frac{x^{2n}}{(2n)!}$;

в) $a_0 = -0.9$, $b_0 = 0.9$, $f(x) = \operatorname{sh} x$, $S(x) = \sum_{n=0}^{+\infty} \frac{x^{2n+1}}{(2n+1)!}$;

г) $a_0 = -0.9$, $b_0 = 0.9$, $f(x) = \frac{m!}{(1-x)^{m+1}}$, $S(x) = \sum_{n=0}^{+\infty} \frac{(n+m)!}{n!} x^n$;

д) $a_0 = -0.9$, $b_0 = 0.9$, $f(x) = \frac{2x}{(1-x)^3}$, $S(x) = \sum_{n=1}^{+\infty} n(n+1)x^n$;

е) $a_0 = -\pi/5$, $b_0 = \pi/5$, $f(x) = \cos x$, $S(x) = \sum_{n=0}^{+\infty} \frac{(-1)^n x^{2n}}{(2n)!}$;

ж) $a_0 = -1$, $b_0 = 1$, $f(x) = 2(\arcsin(\frac{x}{2}))^2$, $S(x) = \sum_{n=0}^{+\infty} \frac{(n!)^2}{(2n+2)!} x^{2n+2}$;

з) $a_0 = -1/3$, $b_0 = 1/3$, $f(x) = \frac{1}{2}(\arcsin(2x))$, $S(x) = \sum_{n=0}^{+\infty} \frac{(2n)!}{(n!)^2(2n+1)} x^{2n+1}$;

и) $a_0 = 1/10$, $b_0 = 1$, $f(x) = \ln x$, $S(x) = \sum_{n=1}^{+\infty} \frac{(-1)^{n-1}(x-1)^n}{n}$;

к) $a_0 = -1/5$, $b_0 = 1/5$, $f(x) = (1-4x)^{-\frac{3}{2}}$, $S(x) = \sum_{n=0}^{+\infty} \frac{(2n+1)!}{(n!)^2} x^n$;

л) $a_0 = -1/5$, $b_0 = 1/5$, $f(x) = -\sqrt{1-4x}$, $S(x) = \sum_{n=0}^{+\infty} \frac{(2n)!}{(n!)^2(2n-1)} x^n$;

м) $a_0 = -1/5$, $b_0 = 1/5$, $f(x) = \frac{1}{\sqrt{1-4x}}$, $S(x) = \sum_{n=0}^{+\infty} \frac{(2n)!}{(n!)^2} x^n$.

При обчисленні за значення $S(x)$ прийняти першу часткову суму S_n, за якої $|S_n-S_{n-1}| < \varepsilon$. Значення a, b, h та ε задає користувач.

```
  double a=x,     // x - перший доданок
         s=0,     // спочатку доданків немає
        x2=-x*x; // мінус квадрат аргументу
  int k=1;        // подвоєний номер доданка - 1
  while (fabs(a) >= eps)
  {  // точність не досягнуто
    s+=a;
    k+=2;
    a*=x2/(k*(k-1));
  }
     // |a| < eps - точність досягнуто
  return s+=a;   // останній доданок теж додаємо
}
```

Для перевірки функції треба забезпечити, щоб в її викликах були аргументи 0, значення, близькі до 1 та –1, а також проміжні значення, наприклад, 0.5, –0.5.

Додатково зауважимо: якщо значення x велике за модулем, то вказана в умові сума збігається повільно, і за великої кількості ітерацій навіть виникають числа, які не можна зобразити в типі **double**. Проте на відрізку $[-\pi; \pi]$ обчислення безпечні, а за x зовні цього відрізку слід скористатися тим, що $\sin(x+2k\pi) = \sin x$ за будь-якого цілого k.

Вправа 6.23. Написати функцію з цілим параметром n, яка повертає значення a_n, обчислене за системою рекурентних співвідношень

$a_1 = 1$, $b_1 = 1$, $a_k = b_{k-1}\cdot a_{k-1} + 2\cdot a_{k-1}$, $b_k = a_{k-1} \% b_{k-1} + 2\cdot a_{k-1}$ за $k > 1$.

Вправа 6.24. Написати функцію, що за заданим n обчислює значення виразу $\sum_{i=1}^{n} \frac{\sum_{j=1}^{i} \cos j}{\sum_{j=1}^{i} \sin j}$. Тригонометричні функції для кожного аргументу мають обчислюватися тільки по одному разу.

Вправа 6.25. Обчислити $\sum_{k=1}^{n} \frac{x^k}{k!!}$, де $-1 < x < 1$, $k!! = 1\cdot 3\cdot \ldots \cdot k$ за непарного k, $k!! = 2\cdot 4\cdot \ldots \cdot k$ за парного k.

Контрольні запитання

6.1. Описати загальний вигляд та порядок виконання інструкції циклу з передумовою.

6.2. Описати загальний вигляд та порядок виконання інструкції циклу з післяумовою.

6.3. За яких обставин тіло циклу з передумовою не виконується жодного разу?

6.4. Якими можуть бути причини зациклювання програми?

6.5. У чому полягає структурність інструкцій? Які інструкції її можуть порушити?

чення змінної в його лівій частині. Так, значенням виразу **a+=b** є «нове» значення змінної **a**.

В описаний спосіб до змінної можна застосувати будь-яку з операцій **+**, **-**, *****, **/**, **%**, а також деякі інші. Наприклад, замість **number=number%10** можна написати **number%=10**, замість **degree=degree+pow(2,n)** — **degree+=pow(2,n)**.

Вправа 3.17. Записати кілька варіантів виразу, що подвоює значення цілої змінної.

Вправа 3.18. Записати кілька варіантів виразу, що збільшує значення цілої змінної на 2.

3.1.5. Особливості цілих типів

Кожен цілий тип «зациклено», тобто «наступним» після максимального числа в типі є мінімальне (див. с. 13). Взагалі, нехай у цілому типі є 2^n значень (n — кількість біт у зображенні даних цього типу) з відповідним максимальним, яке позначимо **MAX**. Якщо сума або різниця двох чисел цілого типу більше **MAX**, то вона зображується в типі числом, яке на 2^n менше справжньої суми чи різниці. Якщо справжня сума або різниця менше мінімального числа цього типу, то результат на 2^n більше її. Аналогічно, результатом множення двох чисел деякого типу є деяке число цього ж типу, різниця між яким і справжнім добутком кратна 2^n.

Приклад. У типі **int** значення 12!=479 001 600 зображується правильно, а 13!=6 227 020 800 — як **1 932 053 504**, тобто 13! – 2^{32}. ◂

Вправа 3.19. Що буде виведено на екран за виконання програми?

```
#include <iostream>
using namespace std;
int main() {
   int iMax=INT_MAX, iMin=INT_MIN;
   cout<<iMax+1<<' '<<iMin-1<<endl;
   system("pause"); return 0;
}
```

3.2. Сумісність та перетворення типів

3.2.1. Сумісність типів за присвоюванням та перетворення типів

- ***Сумісність типів за присвоюванням*** — це можливість присвоювати змінним одного типу значення іншого.

Усі базові типи мови C++ сумісні за присвоюванням один з одним[9]. У виразі присвоювання змінна отримує значення, перетворене до її типу.

- Утворення значення одного типу за значенням іншого типу називається ***перетворенням типу***.

[9] У мовах програмування не всі типи сумісні між собою у присвоюваннях. Наприклад, є мови, що не дозволяють дійсне чи символьне значення присвоїти цілій змінній, а ціле значення — символьній. Проте більшість мов дозволяють присвоювати цілочислові значення дійсним змінним.

При перетворенні цілого значення до дійсного типу відповідне число зображується в цьому типі. За цим правилом кожне значення типу **int** може бути представлено в типі **double**. При перетворенні дійсного значення до типу цілих у ньому відкидається дробова частина. Якщо ціла частина, що залишилася, не належить діапазону чисел типу **int**, то результат перетворення буде помилковим.

Приклади

1. Якщо змінній **double x** присвоюється **1**, її значенням стає дійсне **1.0**.

2. Якщо змінній **int a** присвоюється **1.9**, її значенням стає **1**, а якщо **-1.9**, то **-1**.

3. Нехай є такі змінні й присвоювання.

```
double dd=3.5e38; int id=d;
```

Змінні отримають значення: **dd**=**3.5e38**, **id**=**-2147483648**. ◂

- Присвоюючи вираз одного типу змінній іншого типу, контролюйте, щоб значення виразу могло бути зображено в типі змінної без втрат. Уникайте присвоювань дійсних виразів змінним цілих типів, оскільки це може мати ***непередбачувані наслідки***.

За виконання інструкції **int i=UINT_MAX;** змінна **i** отримає значення **-1**. Не наводячи повного переліку правил перетворень типів, зазначимо лише, що константа **UINT_MAX** задає найбільше беззнакове ціле число 4294967295. А воно не може бути зображено в типі **int**.

- Арифметичний тип вважається більшим за інший арифметичний тип, якщо для зображення його значень потрібно більше байтів або він має більше максимальне значення.
- Уникайте присвоювань значень більшого цілого типу змінним меншого цілого типу, оскільки це може мати ***непередбачувані наслідки***. Зокрема, уникайте перетворення значень беззнакових цілих типів на значення знакових типів.

При перетворенні значення типу **char** до типу **int** однобайтове зображення символу розглядається як зображення[10] числа. При зворотному перетворенні числа від 0 до 127 (а за стандартних налаштувань компілятора й чисел від -128 до -1) до типу **char** результатом є символ, зображений в одному байті так само, як і це число. Наприклад, числу 32 відповідає **' '** (пропуск), числу 48 — символ **'0'**, 49 — **'1'** тощо, числу 65 — символ **'A'**, 66 — **'B'** тощо, числу 97 — **'a'**, 98 — **'b'** тощо.

При перетворенні до типу **int** логічному значенню **false** відповідає **0**, а **true** — **1**. При перетворенні числових значень на логічні нульове значення перетворюється на **false**, а ***усі*** ненульові значення — на **true**.

[10] Залежно від налаштувань компілятора воно може бути знаковим (за стандартних налаштувань) або беззнаковим.

У першому наближенні алгоритм обчислень має такий вигляд (ім'ям **eps** позначено ε).

```
a=x; s=0; k=1;
while (fabs(a)>=eps)
{ s+=a;  // додавання ak
  k++;   // наступний номер
  обчислити a=ak;
}
s+=a;
// |a| < eps, тому |s-sin(x)| < eps
```

Формула наступного члена ряду a_k проста, але, якщо застосувати її безпосередньо, то для кожного члена ряду доведеться обчислювати степінь і факторіал. Проте послідовність доданків теж неважко описати рекурентно:

$a_1 = x$ і $a_k = a_{k-1}\dfrac{-x^2}{(2k-2)(2k-1)}$ за $i > 1$. Звідси, щоб отримати наступне значення a_k, треба лише попереднє a_{k-1} помножити на $-x^2$ і поділити на $(2k-1)(2k-2)$.

Отже, обчислювані величини описуються такою системою рекурентних співвідношень.

$$a_1 = x \text{ і } a_k = a_{k-1}\frac{-x^2}{(2k-2)(2k-1)} \text{ за } k > 1,$$

$S_0 = 0$, $S_k = S_{k-1} + a_k$ за $k > 0$.

За цієї системи тіло циклу можна уточнити так.

```
s+=a; // спочатку додаємо останній доданок
k++;  // потім переходимо до наступного
a*=(-x*x)/((2*k-2)*(2*k-1));
```

Покращимо наведений алгоритм. Помітимо: щоразу відбувається множення на **-x*x**, але **x** у циклі не змінюється! Тому можна перед циклом запам'ятати **-x*x** в допоміжній змінній: **x2=-x*x**. Так само на кожній ітерації двічі обчислюється вираз **2*k**. Уведемо поняття «подвоєний номер доданка **- 1**» і в цьому значенні вживатимемо змінну **k**, перед циклом присвоївши їй **1**. Отже, оптимізуємо цикл так.

```
s+=a; // спочатку додаємо останній доданок
k+=2; // подвоєний номер збільшується на 2
a*=x2/(k*(k-1));
```

- Якщо в циклі обчислюється деякий вираз з незмінним значенням, краще присвоїти його допоміжній змінній перед циклом і використовувати її в циклі.

Оформимо алгоритм обчислень як функцію з параметром **x** (**x** ≤ 1). Точність **eps** = 10^{-6} задамо перед функцією як константу.

```
#define eps 1e-6
```

Імена змінних відповідають уже введеним позначенням.

```
double sum(double x)
{ // обчислення функції sin(x) з точністю eps
```

$n > 1$. Це дозволяє в якості умови продовження взяти $|x_n - x_{n-1}| > d$ за деякого $d > 0$. У цій умові вказано два сусідні члени, тому потрібні дві змінні, які повинні мати різні значення перед кожною перевіркою умови продовження. Щоб забезпечити це, треба перед циклом і в його тілі спочатку запам'ятати останнє обчислене значення, а потім обчислити нове й зберегти в іншій змінній. Номери членів послідовності нас не цікавлять, тому алгоритм має такий вигляд.

```
double mySqrt(double a)
{ x=(a+1)/2;      // x – останнє значення
  y=0.5*(x+a/x); // y – нове значення
  double d=1e-6; // d – можлива похибка
  while (fabs(x-y)>d)
    { x=y; y=0.5*(x+a/x); }
  // |x-y|<=d, тому й |y-sqrt(a)|<d
  return y;
}
```

◂

Вправа 6.21. Написати функцію обчислення $\sqrt{a}$, де $a \geq 0$, за алгоритмом Герона та умовою продовження, яка відповідає нерівності $|x_n^2 - a| > d$.

Вправа 6.22. Послідовність (x_n), задана співвідношеннями

$x_1 = (a+m-1)/m$, $x_i = ((m-1)x_{i-1} + a/x_{i-1}^{m-1})/m$ при $i > 1$,

збігається до $a^{1/m}$. Запрограмувати обчислення $a^{1/m}$ за довільного додатного дійсного a. Як потрібне число беремо перше x_n, для якого $|x_n - x_{n-1}| < \varepsilon$ (ε — додатна константа).

6.3.3. Системи рекурентних співвідношень

Розглянемо задачу наближеного обчислення нескінченної суми

$$S(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!} + \ldots .$$

Відомо, що послідовність часткових сум (S_k), де

$$S_k = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \ldots + \frac{(-1)^{k-1} x^{2k-1}}{(2k-1)!} = \sum_{n=1}^{k} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!},$$

за $-1 < x < 1$ «достатньо швидко» збігається до $S(x)=\sin x$, тобто функцію $\sin x$ на інтервалі $x \in (-1\,;\,1)$ можна зобразити сумою степеневого ряду $\sin x = \sum_{n=1}^{\infty} \frac{(-1)^{n-1} x^{2n-1}}{(2n-1)!}$. Значення $\sin x$ за $x \in (-1\,;\,1)$ можна обчислити наближено як перше S_n, за якого $|a_n| = |S_n - S_{n-1}| < \varepsilon$ (ε — деяка додатна константа). Це ε назвемо «точністю» обчислення.

Нехай $a_k = \frac{(-1)^{k-1} x^{2k-1}}{(2k-1)!}$. Тоді $S_k = a_1 + \ldots + a_k$, послідовність сум (S_k) задовольняє співвідношенню $S_k = S_{k-1} + a_k$ за $k > 0$, а $S_0 = 0$.

Вправа 3.20. Яке значення отримає змінна **c** за виконання інструкції

```
char c=0;
```

?

3.2.2. Сумісність та правила перетворення типів у виразах

- ***Сумісністю*** відмінних між собою типів у виразах називається можливість сумісного використання операндів цих типів.

Типи дійсних і цілих чисел є сумісними, тобто у виразах можна записувати операнди цих різних типів, наприклад, **1+2.0**. Якщо один з операндів цілий, а другий дійсний, то за цілим значенням утворюється відповідне дійсне, і операція застосовується до дійсних значень.

Отже, за виконання **1+2.0** спочатку ціле **1** перетворюється на дійсне **1.0**, а потім додаються дійсні числа.

Вправа 3.21. Що буде виведено на екран за виконання програми? Зверніть увагу на відмінність ділення цілих та дійсних чисел.

```
#include <iostream>
using namespace std;
int main() {
   cout<<(3/5)*20+32.<<' '<<(3/5.)*20+32<<endl;
   system("pause"); return 0;
}
```

Усі цілі (*integral*) типи також є сумісними у виразах. Зокрема, при обчисленні виразів значення типів **char** та **bool** спочатку завжди перетворюються на значення типу **int**. Тому вирази **'5'-'1'** та **'F'-'A'** є синтаксично правильними й мають значення 4 та 5 відповідно.

Вправа 3.22. Яке значення отримає змінна **c** за виконання інструкції

```
char c='f'-'b';
```

?

3.2.3. Явне перетворення типів

Перетворення типів значень, описані вище, відбуваються без явних на це вказівок, тобто неявно. Перетворення можна задати ***явно*** у вигляді ***тип*(*вираз*)** або **(*тип*)*вираз***. Пріоритет перетворення такий самий як і в інших префіксних одномісних операторів.

Приклади

1. Значенням виразу **int(2.8)** є **2**, виразу **double(-2)/3** — **-0.666**..., а виразу **double(-2/3)** — **0.0**.

2. Значенням виразу **char(48)** є символ **'0'**, а виразу **char('0'+7)** — символ **'7'**.

3. У виразах часто зустрічаються дійсні значення дробів з цілими чисельником і знаменником. Якщо не перетворити один з цих операндів до дійсного типу, значення виразу відрізнятиметься від потрібного. Наприклад, за будь-якого додатного значення цілої змінної **n** дріб **n/(n+1)** має значення **0**; щоб отримати потрібне дійсне значення, необхідно явно перетворити тип: **(double)n/(n+1)**, **double(n)/(n+1)** або **n/double(n+1)**. ◂

3.3. Виведення значень виразів

3.3.1. Операція вставки у вихідний потік

Вираз із операцією виведення (вставки) у вихідний потік **<<** має вигляд **cout << *вираз***. Додавши в його кінці символ **;**, отримаємо інструкцію виведення.

Операція **<<** виконується так. Обчислюється значення виразу праворуч, за ним утворюється послідовність символів, яка зображує це значення, і виводиться в потік (за **cout** — на екран). Результатом операції є потік, і до нього можна знову застосувати операцію **<<**. Це уможливлює вирази вигляду

```
cout << вираз_1 << вираз_2
```

та аналогічні з більшою кількістю виразів. Коли послідовно виводяться два вирази, пропуски між ними не додаються. За стандартних налаштувань числа виводяться в десятковому записі.

Приклад. Після виконання інструкцій

```
int a=44; double b=1.2e-2;
cout << "a=" << a << ";\n[" << "b=" << b << ']';
```

на екран виводяться такі рядки.

```
a=44;
[b=0.012]
```

З константи **";\n["** виводиться символ **;**, керуючий символ **\n** задає перехід курсора на новий рядок, в якому першим є символ **[**. ◂

- Пріоритет оператора **<<** нижчий за пріоритет **+** та **–** і вищий за пріоритет порівнянь (див. додаток Б).
- Порядок обчислення виразів-операндів **<<** залежить від компілятора. Більшість компіляторів будують машинний код, в якому вирази обчислюються у зворотному порядку, тобто ***від останнього до першого***. Це важливо, коли вирази змінюють значення змінних.

Приклад. Після виконання інструкцій

```
int k=3; cout << k*11 << ' ' << (k=7) << '\n';
```

на екран виводяться символи **77 7**, оскільки спочатку значенням **k** стає **7**, а потім, уже за нового значення **k**, обчислюється **k*11**. ◂

Виразів виведення, які можуть змінювати значення змінних, краще ***уникати***. Так, інструкцію виведення з наведеного прикладу краще замінити такими.

```
k=7; cout << k*11 << ' ' << k << '\n';
```

Якщо ж спочатку треба надрукувати «старе» значення змінної **k**, множене на **11**, а потім її «нове» значення, то слід писати так.

```
cout << k*11 << ' '; k=7; cout << k << '\n';
```

3.3.2. Елементи форматування вихідного потоку

Бібліотека **iostream** містить багато засобів, які дозволяють керувати процесами введення й виведення. Розглянемо деякі з цих засобів (докладніше див. додаток Г).

Маємо рекурентне співвідношення для послідовності C_n^0, C_n^1, …, C_n^n біномних коефіцієнтів:

$$C_n^k = C_n^{k-1} \cdot \frac{n-k+1}{k}.$$

Початкові умови такі: $C_n^0 = 1$.

Усі елементи послідовності цілі, тому обчислення можна задати так.

```
int i, d;
for (d=1,i=1; i<=k; i++)
   d=d*(n-i+1)/i;
```

Врахувавши тотожність $C_n^k = C_n^{n-k}$, кількість повторень циклу за $k \le n/2$ можна зробити рівною k, інакше — n–k. Не забудемо також, що $C_n^k = 0$ за $k > n$, $k < 0$ або $n < 0$. Оформимо обчислення у вигляді функції з цілими параметрами **n** і **k**, яка повертає значення типу **int**.

```
int Cnk(int n, int k)
{
  if (k>n || k<0 || n<0) return 0;
  int d, i;
  if (k>n/2) k=n-k; //C(n,k)=C(n,n-k)
  for(d=1,i=1; i<=k; i++)
    d=d*(n-i+1)/i;
  return d;
}
```

Для перевірки, чи правильно запрограмовано обчислення, треба задати три пари значень **n** і **k**, за яких результатом має бути 0, а також пари, за яких цикл не виконується жодного разу, виконується один і кілька разів. Особливої уваги потребують випадки, коли **k** = **n/2** і **k** = **n/2+1**, оскільки за таких значень **k** при фіксованому **n** біномні коефіцієнти максимальні. Варто також дослідити граничні пари значень **n** і **k**, за яких правильний результат вміщається в типі **int**. ◂

Вправа 6.19. Не користуючися функцією обчислення біномного коефіцієнта та вкладеними циклами, запрограмувати виведення послідовності

а) C_n^0, C_n^1, …, C_n^n; б) C_{n+0}^n, C_{n+1}^n, …, C_{n+n}^n; в) C_{n+k}^k, C_{n+k+1}^{k+1}, …, C_{n+k+n}^{k+n}.

Вправа 6.20. Написати функцію, яка за цілими n і m повертає кількість цілих невід'ємних чисел, які менше 10^n і мають суму цифр m.

Приклад. Античні греки вміли приблизно обчислювати $\sqrt{a}$, де $a > 0$, за допомогою рекурентної послідовності чисел, яка збігається до $\sqrt{a}$. За алгоритмом Герона, послідовність утворюється шляхом застосування рекурентного співвідношення $x_n = \frac{1}{2}(x_{n-1} + \frac{a}{x_{n-1}})$, починаючи з будь-якого додатного x_1, наприклад, з $x_1 = (a + 1)/2$.

Однією з властивостей послідовності є те, що $|x_n - \sqrt{a}| < |x_n - x_{n-1}|$ за

Розглянемо приклади, в яких рекурентні співвідношення не вказано явно.

Приклад. За натуральним n обчислити значення виразу $\sqrt{2+\sqrt{2+\ldots+\sqrt{2}}}$ (n знаків кореня).

Помітимо: результат можна обчислити як n-й елемент послідовності $\sqrt{2}$, $\sqrt{2+\sqrt{2}}$, $\sqrt{2+\sqrt{2+\sqrt{2}}}$, …. Нехай $a_n = \sqrt{2+\sqrt{2+\ldots+\sqrt{2}}}$ (n знаків кореня).

Щоб розв'язати цю задачу, знайдемо для послідовності (a_n) рекурентне співвідношення та початкові умови. На кожному кроці до попереднього результату додається 2 й береться квадратний корінь з цієї суми, отже, $a_n = \sqrt{2+a_{n-1}}$ (це наше рекурентне співвідношення). Помітимо, що $a_1 = \sqrt{2} = \sqrt{2+0}$. Тому для систематичності покладемо $a_0 = 0$ (це наші початкові умови). Отже, щоб обчислити a_n, достатньо, починаючи з a_0, n разів виконати обчислення за рекурентним співвідношенням. Можна програмувати.

```
double res;
for(res=0, int i=0; i<n; i++){
  res=sqrt(2+res);
}
```
◂

Вправа 6.17. Написати функцію, яка за заданим n обчислює значення виразу $\sqrt{3+\sqrt{6+\ldots+\sqrt{3(n-1)+\sqrt{3n}}}}$ (n коренів).

Вправа 6.18. Написати функцію, яка за заданим n визначає число, що утворюється оберненням десяткового запису n. Наприклад, оберненням 123 є 321, 340 — 43 (незначущі нулі відкидаються).

Приклад. Формула ***бінома Ньютона*** $(a+b)^n = \sum_{k=0}^{n} C_n^k a^k b^{n-k}$ — одна з найвідоміших у математиці. Коефіцієнти $C_n^k = \dfrac{n!}{k!(n-k)!}$ називаються ***біномними***. Обчислимо біномний коефіцієнт за заданими n і k.

Можна написати функцію для обчислення факторіалів і тричі її викликати, але це дуже нераціонально. По-перше, одні й ті самі значення, що виникають у процесі обчислення факторіала, обчислюються тричі. По-друге, функція $m!$ за збільшення m зростає дуже швидко, тому існує чимало значень n і k, за яких значення C_n^k можна зобразити в цілих типах, а $n!$ і $k!$ — ні.

За умови $k \ge 1$ перетворимо формулу біноміального коефіцієнта

$$C_n^k = \frac{n!}{k!(n-k)!} = \frac{n!\cdot(n-k+1)}{k(k-1)!(n-k)!(n-k+1)} = \frac{n!}{(k-1)!(n-k+1)!}\cdot\frac{n-k+1}{k} = C_n^{k-1}\cdot\frac{n-k+1}{k}.$$

Формати дійсних значень. Інструкції

```
cout<<1.0; cout<<' '<<0.1; cout<<' '<<0.00000001;
```

виведуть на екран `1 0.1 1e-008`. Перше число виглядає як ціле, друге має дробову частину, третє — від'ємний порядок. Чому ***формати*** (або форми) виведення чисел такі різні й як ними керувати?

Дійсні значення виводяться або з ***фіксованою*** (*fixed*) ***крапкою*** , або в ***нормалізованій*** («***науковій***», *scientific*) ***формі***. Після інструкції `cout<<scientific;` дійсні числа виводяться в нормалізованій формі, а після `cout<<fixed;` — з фіксованою крапкою. Для обох форматів інструкція вигляду `cout.precision(2);` задає кількість знаків дробової частини (тут їх 2). Якщо спосіб виведення явно не встановлено, то засоби виведення обирають його самі.

Приклад. Виконання програми

```
#include <iostream>
using namespace std;
int main() {
   cout<<scientific; cout.precision(1);
   cout<<1.14; cout<<"  "; cout<<0.16; cout<<'\n';
   cout<<fixed; cout.precision(1);
   cout<<1.14; cout<<"  "; cout<<0.16; cout<<'\n';
   system("pause"); return 0;
}
```

prog005.cpp

дасть такий вихід.

```
1.1e+000  1.6e-001
1.1  0.2
```

Зверніть увагу: відбулося ***округлення*** дійсних чисел до вказаної кількості дробових знаків. ◂

Вправи

3.23. Що буде виведено за виконання інструкцій?

```
cout<<fixed; cout.precision(2);
cout<<-123.321<<' '<<345.2<<' '<<11.777<<' '<<13.<<' '<<7;
```

3.24. Що буде надруковано за виконання інструкцій?

```
cout<<scientific; cout.precision(2);
cout<<12.151; cout<<" 12.151"; cout<<'\n';
cout<<fixed; cout.precision(2);
cout<<12.151; cout<<" 12.151"; cout<<'\n';
```

3.25. Написати програму, яка виводить значення $\sqrt{2}$ з вісьмома знаками після десяткової крапки.

3.26. Написати програму, яка виводить значення $(2{,}5)^{1{,}6}$ з чотирма знаками після десяткової крапки.

Інтерпретація символів. Операція `<<` перетворює значення певного типу в послідовність символів, яка зображує це значення, і виводить її у вихідний потік. При цьому вивідні символи ***інтерпретуються***, тобто виконуються особливі дії, задані деякими символами, наприклад, за появи

керуючого символу `'\n'` виведення продовжується в новому рядку. Саме інтерпретація символів дозволяє виводити одні й ті самі значення в різних форматах, як, наприклад, дійсні числа з різною кількістю десяткових знаків.

Кінець рядка. Кінець рядка можна позначити іменем `endl` (скорочення від *end of line* — кінець рядка): вираз `cout<<endl`, як і вираз `cout<<'\n'`, задає виведення символу `'\n'` у вихідний потік. [11]

Ширина поля та притискання до його краю. Виклик функції `width` дозволяє задати певну ***ширину поля***, яке займуть символи значення, що буде виводитися наступним. Наприклад, інструкція `cout.width(10);` встановлює ширину поля 10. Якщо для наступного значення потрібно більше символів, ніж задана ширина, то друкуються всі символи, а якщо менше, то спочатку виводяться пропуски (доповнюючи кількість символів значення до заданої ширини), а потім символи значення. Встановлена ширина стосується лише одного елемента виведення.

Приклад. За виконання інструкцій

```
cout<<'#'; cout.width(4); cout<<12<<'#';
cout<<'#'; cout.width(4); cout<<123456<<'#';
```

на екрані з'явиться текст `#  12##123456#`. Тут виведено два пропуски перед `12` і всі цифри `123456`. На символи `'#'`, тобто елементи, наступні за числовими константами, встановлена ширина поля не впливає. ◂

За встановленої ширини поля вивідні символи «притискаються» до правого краю поля. Притискання до лівого краю можна задати інструкцією `cout<<left;` . Її дія поширюється на всі подальші виведення, в яких встановлено ширину поля, поки не буде задано притискання до правого краю, тобто `cout<<right;` .

Керування форматом виведення. Керувати можна не тільки форматами дійсних значень, шириною поля та способом притискання до його країв, але й багатьма іншими параметрами виведення; докладніше див. додаток Г.

Вправа 3.27. Написати програму, яка запитує в користувача координати точки дійсної площини, а потім виводить отриману точку. Зверніть увагу на зовнішній вигляд друкованого результату.

Вправа 3.28. Написати програму, яка запитує в користувача коефіцієнти рівняння $ax^2+bx+c=0$, а потім виводить отримане рівняння (x^2 можна вивести у вигляді `x^2`). Зверніть увагу на зовнішній вигляд друкованого рівняння.

[11] Насправді, він означає дещо більше, але розгляд цього виходить за рамки книги.

Увага! Під час «зсуву віконця» першою нове значення має отримувати його «найстаріша» змінна, тобто спочатку `f2`, а потім `f1`. Присвоювання `f1=f; f2=f1;` були б дуже грубою помилкою.

Для перевірки цієї функції потрібно, наприклад, у головній функції програми задати аргументи в її виклику –1, 0, 1, 2 та деякі інші так, щоб було перевірено обробку некоректних параметрів, обчислення початкових елементів, одно- та багаторазове виконання тілу циклу.

Зауважимо: послідовність чисел Фібоначчі зростає доволі швидко, наприклад, 50-е число «не вміщається» в типі `int`.

◂

Змінна `i` використовується тільки в тілі циклу. Мова C++ дозволяє такі змінні оголошувати в початковій дії циклу `for`. Це оголошення буде діяти до кінця циклу. Відповідний фрагмент коду такий.

```
for (int i=1; i<n;i++) {
  f=f1+f2;
  f2=f1;f1=f;//зсув віконця
} //оголошення імені i вже не діє
```

Зверніть увагу: щоб знайти *n*-й елемент послідовності, заданої рекурентним співвідношенням, необхідно послідовно обчислювати 1-й, 2-й, …, *n*-й елементи.

Вправи

6.13. Яким буде значення виразу `fibo(5)`, якщо зсув віконця записати в іншій послідовності, тобто так `f1=f; f2=f1;` ?

6.14. Написати функцію, яка за цілим $n \ge 0$ повертає *n*-й елемент послідовності, заданої рекурентним співвідношенням $a_n = 3a_{n-1} - a_{n-2}$, $n \ge 3$, $a_1 = 3$, $a_2 = -2$.

6.15. Написати функцію, яка за заданим цілим числом *n* знаходить найменше число Фібоначчі, яке більше *n*. (Вказівка: обчислення елементів послідовності Фібоначчі має закінчитися, коли останній обчислений елемент буде більше *n*.)

6.16. Визначити найбільше число Фібоначчі в межах типу `int` та його номер.

Рекурентні співвідношення використовують у програмуванні обчислень, пов'язаних з послідовностями. Співвідношення дозволяють формально виразити залежності між величинами, які обчислюються послідовно, а на основі цих виразів легко запрограмувати цикл. Для розв'язання задачі треба:

а) зрозуміти, що величини утворюють послідовність;

б) записати відповідне рекурентне співвідношення;

в) визначити перші члени послідовності, які обчислюються без співвідношення (за початковими умовами);

г) сформулювати умову продовження, за якої застосовується співвідношення.

Після цього згадані вище «деталі конструктора» й порядок їх розташування в алгоритмі стають очевидними.

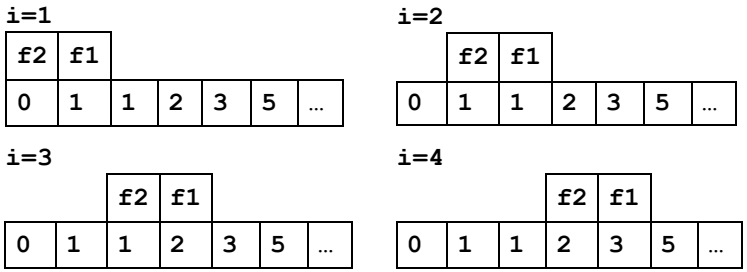
Далі безпосередньо за ними неважко запрограмувати циклічні обчислення.

Приклад. За цілим числом $n \geq 0$ знайти число Фібоначчі F_n.

Для розв'язання цієї задачі запрограмуємо обчислення за рекурентним співвідношенням $F_n = F_{n-1}+F_{n-2}$, $n \geq 2$, та початковими умовами $F_0 = 0$, $F_1 = 1$.

Якщо $n = 0$, то результатом є 0; якщо $n = 1$, то результатом є 1. За $n \geq 2$ будемо обчислювати елементи послідовності один за одним, доки не отримаємо числа з заданим номером. За F_0 та F_1 обчислимо F_2, за F_1 та F_2 — F_3, і так далі. При цьому, щоб обчислити наступний елемент, потрібно знати два останні елементи, що йому передують. Щоб вчасно зупинитися, потрібно також зберігати й щоразу збільшувати номер елемента, обчисленого останнім.

Отже, виділимо поняття: ***передостаннє число***, ***останнє*** та ***наступне***, а також ***номер*** числа, обчисленого останнім. Представимо їх змінними, відповідно, **f2**, **f1**, **f** та **i**. Змінні **f2**, **f1** можна розуміти як «віконця», що пересуваються послідовністю Фібоначчі.



Отже, зі збільшенням **i** віконце пересувається «на одну позицію праворуч». Для цього обчислюється наступний елемент і зберігається в змінній **f**, значення **f1** стає значенням **f2**, а **f** — значенням **f1**.

було	f2	f1	f
			f2+f1
стало		f2	f1

Реалізуємо наведені міркування у функції.

```
int fibo(int n) {
//pre: n>=0, інакше повертаємо -1
  int f2=0, f1=1, f;
  if (n<0)return -1;
  if (n==0) return f2;
  if (n==1) return f1;
  int i;
  for (i=1; i<n;i++) {
    f=f1+f2;
    f2=f1; f1=f;     //зсув «віконця»
  }
  return f;
}
```


3.4. Приклад програми та поняття якості коду

3.4.1. Коментарі та якість коду

Напишемо програму, яка вводить температуру за Цельсієм (ціле число) і виводить температуру за Кельвіном, більшу на 273 градуси (273 — константа Кельвіна). Ось початковий (і не найкращий!) варіант.

```
/* Програма вводить температуру за Цельсієм і
   виводить температуру за Кельвіном */
#include <iostream>
using namespace std;
int main()
{
  int tC,          // температура за Цельсієм
      tK;          // температура за Кельвіном
  cout << "Celsium to Kelvin.\n";
  cout << "Enter temperature by Celsium (int>=-273):";
  cin >> tC;
  tK = tC+273;
  cout << "temperature by Kelvin: " << tK << endl;
  system("pause"); return 0;
}
```

Порівняно з попередніми прикладами, ця програма має нові елементи. У першому й другому рядках записано ***коментар*** — послідовність символів між парами символів **/*** та ***/**, яка не містить ***/**. Такий коментар може займати кілька рядків та бути записаним між будь-якими двома лексемами. Інші коментарі є послідовностями символів, що починаються парою символів **//** і закінчуються в кінці рядка, який їх містить. Під час компіляції коментарі пропускаються.

Програми можна оцінювати за їх якістю, причому з різних позицій. На прикладі обчислення температури за Кельвіном розглянемо якість програми з точки зору легкості її розуміння, використання та модифікації. Розглянемо ще один варіант програми.

```
#include <iostream>
using namespace std;
int main()
{  int tC, tK;
   cin >> tC;
   tK = tC+273;
   cout << "temperature by Kelvin: " << tK << endl;
   system("pause"); return 0;
}
```

Обидві програми розв'язують одну й ту саму задачу, але перша краще тим, що зрозуміти її набагато легше. Отже, ***код повинен читатися та сприйматися як розмовна мова***. Мартін Фаулер (Martin Fowler) сформулював це так.

- «Кожен дурень може написати код, зрозумілий комп'ютеру. Добрі програмісти пишуть код, зрозумілий людям.»

Чому код ***повинен*** бути зрозумілим людині? Справа в тому, що код, який у сучасних програмах вимірюється мільйонами (!) рядків, необхідно ***обслуговувати*** — налагоджувати, виправляти помилки та модифікувати в разі змін вимог до нього. І робити все це має не комп'ютер, а людина.

- Щоб полегшити людині роботу з кодом, можна використати коментарі. Але ними не слід зловживати. Коментар біля кожної інструкції програми аніскільки не зробить код зрозумілішим для людини.Високоякісний код повинен містити коментарі тільки там, де вони дійсно ***необхідні***.

Зміст коментарів може бути довільним, але зазвичай у коментарях зазначають:

– що повинна виконувати програма, функція чи фрагмент коду;

– яку сутність моделює або для чого використовується змінна;

– умови, які мають справджуватися ***перед*** виконанням певних інструкцій, щоб забезпечити їх коректність (так звані ***pre-умови***, або ***передумови***);

– умови, які справджуються ***після*** виконання певних інструкцій (так звані ***post-умови***, або ***післяумови***);

– поведінку програми за некоректних даних.

3.4.2. Змістовні імена

Отже, одним із засобів прояснити призначення коду та полегшити його сприйняття є коментарі. Проте для ефективного обслуговування коду самих їх замало. Якщо код необхідно виправити або модифікувати, все рівно доведеться розібратися в ньому до дрібниць.

- Код повинен бути зрозумілим ***навіть без додаткових коментарів***. (Звісно, про складні математичні алгоритми тут не йдеться.)

На практиці, якщо код стає дуже заплутаним та незрозумілим, то для виправлення навіть однієї помилки в його логіці буває простіше переписати все «з чистого аркуша», ніж внести зміни в існуючий код. І коментарі тут не врятують.

Щоб уникнути зайвих коментарів, варто обирати імена, які відображають призначення відповідних змінних та функцій. У прикладі є коментарі щодо призначення змінних **`tC`** та **`tK`**. Програма коротенька, і побачити коментарі легко. Проте за більшої програми довелося б шукати відповідні пояснення серед тисяч рядків коду. Отже, краще не коментувати зміст змінних, а дати їм ***змістовні імена***, що відображають їх призначення. Замінимо імена **`tC`** та **`tK`** іменами **`tCelsium`** та **`tKelvin`** відповідно, де літера **`t`** є скороченням від **`temperature`** (температура). Зауважимо: у бібліотеках сучасних систем програмування використовуються саме змістовні імена (**`cin`**, **`cout`**, **`pow`**, **`sqrt`** тощо), і це істотно полегшує сприйняття коду.

3.4.3. Іменування констант

Ще один недолік нашої програми — використання константи **`273`**. Такі «магічні» константи можуть з'являтися в різних місцях програми, але людина може помилитися, і десь набрати не **`273`**, а, скажімо, **`237`**. Синтакси-

6.3. Рекурентні співвідношення в циклічних алгоритмах

6.3.1. Рекурентні співвідношення

Повернемося до задачі обчислення факторіала невід'ємного цілого числа n. Для розв'язання цієї задачі серед інших варіантів було побудовано такий код.

```
fact=1;
for (k=1; k<=n; k++) fact*=k;
```

Його можна прочитати й так:

```
факторіал числа 0 дорівнює 1;
поки не знайдемо факторіал числа n
  обчислювати факторіал кожного наступного числа як
              факторіал попереднього, множеного на це число;
//останнє знайдене значення факторіалу є шуканим
```

Суттєвим у цьому алгоритмі є те, що шукається елемент послідовності факторіалів 0!, 1!, 2!, … з певним номером. При цьому сама послідовність визначається двома законами: один задає перший елемент послідовності, інший каже, як обчислити елемент за його номером та попереднім елементом. Отже, є початкові умови та рекурентне співвідношення, які разом задають послідовність. Для послідовності факторіалів початкові умови — це $0! = 1$, а рекурентне співвідношення — $n! = (n-1)!\cdot n$ за $n > 0$.

Розглянемо загальнішу ситуацію. Припустимо, що перші k елементів послідовності $a_0, a_1, \ldots, a_n, \ldots$ задано явно (це ***початкові умови*** $a_0, a_1, \ldots, a_{k-1}$), і є закон F, який дозволяє за номером елемента та/або деякими попередніми елементами знайти ***всі інші*** елементи:

$a_i = F(i, a_{i-1}, \ldots, a_0)$ за всіх $i \ge k$.

Останнє співвідношення називають ***рекурентним співвідношенням***. Кожне рекурентне співвідношення разом з початковими умовами визначає в точності одну послідовність.

Приклади

1. Рекурентне співвідношення $a_n = a_{n-1}+d$, $n \ge 1$, з початковими умовами $a_0 = a$ задає арифметичну прогресію з початковим елементом a та кроком d. Співвідношення $a_n = q\cdot a_{n-1}$, $n \ge 1$, та умови $a_0 = a$ визначають геометричну прогресію з початковим елементом a та коефіцієнтом q.

2. Рекурентне співвідношення $S_n = S_{n-1}+\sin(n)$, $n \ge 1$, з початковими умовами $S_0 = 0$ задає суму $\sum_{k=1}^{n} \sin(k)$.

3. Рекурентне співвідношення $F_n = F_{n-1}+F_{n-2}$, $n \ge 2$, з початковими умовами $F_0 = 0$, $F_1 = 1$ визначає рекурентну послідовність, що має назву ***послідовність Фібоначчі***. Вона має такий початок: 0, 1, 1, 2, 3, 5, 8, 13, … . ◂

6.3.2. Програмування циклічних обчислень за рекурентними співвідношеннями

За допомогою рекурентних співвідношень можна розв'язати чимало задач. Головне — знайти рекурентне співвідношення та початкові умови.

```
початкова дія;
while (умова)
{
   основна дія;
   перехідна дія;
}
```

Інструкція **break** у тілі циклу **for** завершує його виконання, а інструкція **continue** завершує виконання лише тіла циклу; відразу після неї виконується перехідна дія.

Приклади

1. Дуже часто інструкція циклу **for** зустрічається у вигляді

```
for (k=0; k<n; ++k) інструкція
```

й задає виконання ***інструкції*** за значень $k = 0, 1, 2, \ldots, n-1$, або у вигляді

```
for (k=1; k<=n; ++k) інструкція
```

й задає виконання ***інструкції*** за значень $k = 1, 2, \ldots, n$, або у вигляді

```
for (k=n; k>0; --k) інструкція
```

й задає виконання ***інструкції*** за значень $k = n, n-1, \ldots, 2, 1$. Змінну **k** у цих ситуаціях інколи називають ***лічильником циклу***.

2. Функція **printFraction** друкує перші k дробових знаків десяткового запису дійсного числа v від 0 до 1 (див. алгоритм у додатку А).

```
//pre: 0<=v<1 && k>0, інакше функція нічого не виводить
void printFraction(double v, int k){
   int d; //поточна цифра
   if (!(0<=v)&&(v<1)&&(k>0)) return;
   for(;k>0;k--){
     v*=10; d=int(v); v-=d;
     cout<<d;
   }
}
```

Операція слідування. У прикладі з обчислення факторіала спочатку треба присвоїти значення не тільки лічильнику **k**, але й змінній **fact**. На відміну від багатьох мов програмування, C++ дозволяє поєднати ці дві дії в одному виразі — за допомогою операції слідування.

Операція слідування, або ***«кома»***, зі знаком «**,**» позначає послідовне обчислення виразів, записаних через кому. Ця послідовність виразів розглядається як один вираз; його значенням є значення останнього виразу.

З її використанням цикл обчислення факторіала запишемо так.

```
for (fact=1,k=1; k<=n; k++) fact*=k;
```

- Операція слідування дозволяє на місці одного виразу записати кілька присвоювань.

Вправа 6.11. За $n \geq 0$ значення $n!!$ («подвійний» факторіал) задається так: $0!!=1$, $1!!=1$, $n!!=n\cdot(n-2)!!$, якщо $n \geq 2$. Написати функцію, яка за цілим числом повертає його «подвійний» факторіал.

Вправа 6.12. Написати функцію, яка за заданими дійсними a, h та цілим m друкує значення функції $\operatorname{arctg}(\sin x)$ у точках $a, a+h, \ldots, a+mh$.

чно програма залишиться правильною, а семантично — ні. Ще одна проблема з «магічними» константами виникає, коли в процесі створення програми таку константу потрібно змінити. Адже в усій програмі вона має змінитися ***однаково***! Одним із засобів уникнути «магічних» констант є директива **#define** (означити).

Директива препроцесора **#define** в найпростішому випадку записується так.

```
#define ім'я значення
```

Після такої директиви усі ***лексеми*** (див. с. 23) вхідної програми, що збігаються з указаним іменем, препроцесор замінює на вказане значення. Значенням може бути довільна послідовність символів. Наприклад, за наявності директиви

```
#define ZERO 0
```

інструкція **int i=ZERO;** перетвориться препроцесором на **int i=0;**, а інструкції **int ZERO1;** та **cout<<"ZERO";** залишаться без змін, оскільки лексемами в них є **ZERO1** та **"ZERO"**, а не **ZERO**.

Якщо в директиві **#define** вказано тільки ім'я, а значення відсутнє, то препроцесор вважає, що імені відповідає порожня послідовність символів.

- Можна вважати, що директива **#define** дає константі ім'я, тобто ***іменує константу***.

За наявності директиви **#define KELVIN 273** замість «магічної» константи 273 можна використовувати іменовану константу з ім'ям **KELVIN** і значенням **273**. Отже, програма стане такою.

```
/* Програма вводить температуру за Цельсієм і */
/*   виводить температуру за Кельвіном        */
#include <iostream>
using namespace std;
#define KELVIN 273
int main()
{ int tCelsium,tKelvin;
  cout << "Celsium to Kelvin.\n";
  cout << "Enter temperature by Celsium (int>=-"
       <<  KELVIN << "):";
  cin >> tCelsium;
  tKelvin = tCelsium+KELVIN;
  cout << "temperature by Kelvin: " << tKelvin << endl;
  system("pause"); return 0;
}
```

prog006.cpp

3.4.4. Константи як змінні з незмінним значенням

Директива **#define** має кілька недоліків, з яких розглянемо два. По-перше, у директиві в якості значення можна записати будь-яку послідовність символів, тому через необережний запис директиви препроцесор може змінити лексичну структуру програми. По-друге, тип значення не зада-

но явно, тому його визначає компілятор, а це не завжди бажано. Уникнути цих недоліків можна, використовуючи так звані ***змінні з незмінним значенням***.

Зарезервоване слово **`const`** (його називають ***специфікатором типу***) перед іменем типу в інструкції оголошення імені змінної означає, що значення цієї змінної не можна модифікувати. Змінна ініціалізується в оголошенні значенням будь-якого виразу відповідного типу (якщо вираз містить ім'я іншої змінної, їй раніше має бути присвоєно значення). Запис її імені ліворуч в інструкції присвоювання є помилкою.

Приклади

1. У програмі про температуру замість директиви **`#define KELVIN 273`** можна записати зовнішнє, тобто розташоване ***зовні головної функції***, оголошення імені.

```
const int KELVIN=273;
```

Оголошення задає ім'я цілої змінної зі значенням **`273`**. Її можна використовувати у функціях, записаних нижче в програмі. Решта програми залишається без змін.

2. Розглянемо такі інструкції оголошення.

```
int good=8, bad;
const int good4=4*good, bad4=4*bad;
```

Змінна **`good4`** отримає значення **`32`**, яке в подальшому не можна змінити, а значення **`bad4`** непередбачуване, оскільки перед її ініціалізацією змінна **`bad`** значення не отримала. ◂

- Змінні, оголошені зі словом **`const`**, часто використовують для іменування констант. В іменах констант прийнято (хоча й не обов'язково) використовувати тільки великі літери.
- Корисно іменувати вирази з постійним значенням, особливо, якщо вони потрібні в багатьох місцях програми. У цих місцях замість виразу достатньо записати лише його ім'я. Окрім того, за необхідності змінити вираз, достатньо зробити це лише в іменуванні, а якщо вираз не іменовано, доведеться змінювати його всюди, де він зустрічається.

Контрольні запитання

3.1. Що є семантикою арифметичного виразу?
3.2. Яка математична операція, оператор якої відсутній у мові C++, має властивість правобічного зв'язування?
3.3. Який вигляд має виклик функції?
3.4. Що таке сумісність типів?
3.5. Що таке перетворення типу?
3.6. Що таке сумісність типів за присвоюванням?
3.7. Як виглядає явне перетворення типу?
3.8. Чи кожне дійсне число можна точно представити в типі double?
3.9. З яких основних структурних частин складається програма, записана мовою C++?

конати ще одне обчислення, і за згоди користувача виконувала його. Відповідь користувача промоделювати символьною змінною: значення **`Y`**, **`y`** відповідають необхідності подальших обчислень обчислення.

6.8. **Табулювання функції**. Нехай задано дійсні числа a, b, причому $a \le b$ ($[a; b]$ — відрізок, на якому виконується табулювання), та додатне дійсне число h — крок табулювання. Необхідно вивести на екран два стовпчики — перший задає послідовні значення a, $a+h$, $a+2h$, $a+3h$, …, b аргументу функції з відрізку $[a; b]$, а другий — значення функції $\sin x$ у відповідних точках. Незалежно від того, чи ділиться довжина відрізка націло на h, останній рядок повинен містити числа b та $\sin b$. Розв'язком має бути **`void`**-функція з параметрами **`a`**, **`b`**, **`h`**. У коментарі вказати передумови виклику функції та описати її поведінку за некоректних вхідних даних.

6.9. Написати функцію, яка на відрізку $[a; b]$ з кроком h табулює обидві функції $\sin x$ і $\cos x$.

6.10. **Табулювання функції сторінками**. Написати функцію, яка на відрізку $[a; b]$ табулює з кроком h функцію $\sin x$, але після виведення кожних m рядків виводиться запит, чи продовжувати друкування. Робота завершується після відповіді «0».

6.2. Інструкція циклу for

Приклад. Нагадаємо: значення факторіала $n!$ невід'ємного цілого числа n визначається формулою $n! = 1 \cdot 2 \cdot \ldots \cdot (n-1) \cdot n$ при $n > 0$, $0! = 1$. Звідси, щоб обчислити цей добуток, можна взяти початкове значення 1 та послідовно помножити його на кожне число від 1 до n.

Цим діям відповідає послідовність інструкцій, в якій інструкції та вирази відіграють чітко визначені ролі.

```
fact=1;           // початкове значення добутку
k=1;              // початок перебору множників
while (k<=n) {    // перевірка умови продовження
  fact*=k;        // основна дія
  ++k;            // перехідна дія перед наступним кроком
} // після закінчення повторень k=n+1, fact=n!
```

Ці елементи виконуються в тому самому порядку, якщо записати їх так.

```
fact=1;
for (k=1; k<=n; ++k) fact*=k;
```

◂

Інструкція циклу **`for`**, або **`for`**-інструкція, має такий загальний вигляд.

```
for (початкова дія; умова; перехідна дія)
   основна дія
```

Слово **`for`** є зарезервованим, дужки та два знаки **`;`** всередині дужок є обов'язковими. Початкова дія, умова та перехідна дії є ***виразами*** (кожен з них може бути порожнім), основна дія — ***інструкцією***. Тілом циклу **`for`** називають його основну дію. Інструкція **`for`** виконується так само, як і інструкції такого вигляду.

```
  cout << "sum=" << sum << endl;
  system("pause"); return 0;
}
```
prog014.cpp

Зазначимо, що використання інструкції **`continue`** в цій програмі є дуже штучним. Ще одним недоліком є те, що в кінці не повідомляється, чи були помилки під час уведення. Проте обробка помилок у вхідних даних виходить за межі цієї книжки.

◂

За виконання послідовності інструкцій у програмі, після того, як виконано інструкцію, виконується ***наступна за нею в тексті програми***. Про таку програму кажуть, що вона є ***структурованою***. Інструкції **`break`** і **`continue`** ***порушують*** цей порядок обчислень, заплутуючи текст програми. Тому, користуючися ними, програміст має ретельно відслідковувати точку програми, якою продовжуються обчислення. Інколи ці інструкції дійсно скорочують запис розгалужень у циклі, проте в більшості випадків ті ж самі дії без втрати прозорості коду ***можна записати*** без них. Отже, не зловживайте **`break`** і **`continue`**.

Вправи

6.2. Модифікуйте програму prog014.cpp (с. 82), щоб позбутися **`break`** і **`continue`**.

6.3. Написати функцію, яка за цілим числом визначає:
а) кількість цифр його десяткового запису;
б) чи зустрічається в його десятковому запису задана цифра;
в) кількість входжень заданої цифри в його десятковий запис;
г) старшу цифру його десяткового запису;
д) мінімальну (максимальну) цифру його десяткового запису.

6.4. Написати функцію, яка за цілим a обчислює та повертає найбільше n, за якого $n! \le a$.

6.5. Написати функцію, яка за цілими n та m обчислює та повертає $[\log_n m]$. (У передумові до функції зверніть увагу на значення функції у випадку некоректних вхідних даних.)

6.6. На вхід програми дається послідовність символів **`(`** та **`)`**. Послідовність вважається правильною, якщо вона містить однакові кількості символів **`(`** і **`)`** та в довільному її початковому відрізку символів **`(`** не менше, ніж **`)`**. Ознакою завершення послідовності є введення будь-якого непорожнього символу, відмінного від **`(`** та **`)`**. Програма має визначити, чи є вхідна послідовність правильною.

6.7. Програма в прикладі на с. 63 за введеним з клавіатури дійсним числом, знаком операції (**`+`**, **`-`**, **`*`** або **`/`**) та ще одним числом друкувала результат застосування операції до цих чисел. Модифікувати її, щоб після кожного обчислення вона запитувала користувача, чи треба ви-

3.10. Чи можна створити змінну або константу з іменем int, const?
3.11. Чи можна створити змінну або константу з іменем sqrt, main?
3.12. Чим відрізняється іменування константи від ініціалізації змінної?
3.13. Для чого в програмах використовуються коментарі?
3.14. Чому для змінних варто обирати змістовні імена?
3.15. Які існують формати виведення дійсних чисел?

Задачі

3.1. Написати програму, яка запитує в користувача два дійсних числа, а потім виводить їх суму та різницю.

3.2. Математичний вираз $\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$ задає обчислення відстані між точками площини з координатами (x_1, y_1) та (x_2, y_2). Написати програму, яка запитує в користувача координати двох точок дійсної площини, а потім виводить задані точки та відстань між ними (див. вправу 3.27 на с. 46).

3.3. Підготувати реферат на тему: «Правила перетворення цілих (*integral*) типів у мові C++». Написати програми, які демонструють результат відповідних перетворень.

3.4. Вказати повний перелік прапорців (див. додаток Г), які можна використовувати для потоків уведення та виведення. Описати, що вони задають. Написати приклади використання прапорців, що демонструють їх вплив на форматування. Які прапорці встановлено для потоків **`cin`** та **`cout`** за замовчування?

Глава 4. Найпростіші засоби керування порядком обчислень

4.1. Умови

4.1.1. Операції порівняння

Двомісні операції ***порівняння*** мають знаки **==**, **!=**, **>**, **<**, **>=**, **<=** («дорівнює», «не дорівнює», «більше», «менше», «більше або дорівнює», «менше або дорівнює»). Результатом порівняння є **false** («хибність») або **true** («істина») — значення логічного типу **bool**. Нагадаємо, що при перетворенні логічних значень до типу цілих **false** стає **0**, а **true** — **1**. При зворотному перетворенні ненульові значення дають **true**, а значення 0 (нуль) — **false**.

Приклади

1. Вирази **1==2** і **'A'=='a'** мають значення **false**, вирази **1!=2** і **1>=1** — значення **true**. Значенням виразу **sizeof(b*b-4*a*c)>0** є **true**.

2. Значенням виразу **b*b-4*a*c>0** за **a==1**, **b==5**, **c==4** є **true**, за **a==1**, **b==2**, **c==1** та **a==1**, **b==1**, **c==1** — **false**. ◂

Порівнювати одне з одним можна не лише числові, але й символьні та логічні значення. При цьому символьні та логічні значення перетворюються до цілих.

Приклад. Значенням виразу **'a'<='A'** є **false**, а виразів **'4'<'9'** та **'A'<'Z'** — **true** (значеннями **int('a')**, **int('A')**, **int('4')**, **int('9')**, **int('Z')** є числа 97, 65, 52, 57, 90 відповідно). Значенням виразу **false==true** є **false**, а виразу **false<true** — **true**. ◂

Вправа 4.1. У чому відмінність виразів **a=3** та **a==3**?

Вправа 4.2. Обчислити значення виразів а) **1.5<7**; б) **'3'<'a'**; в) **'Z'<'a'**; г) **false<true**; д) **false<-1**? Які неявні перетворення типів виконуються під час їх обчислення?

4.1.2. Логічні операції

Двомісні операції «і», «або» та одномісна «не» (відповідно, булеве множення, або кон'юнкція, булеве додавання, або диз'юнкція, та заперечення) в логіці та математиці називаються ***булевими*** або ***логічними*** й застосовуються до булевих значень. У мові C++ вони відповідно мають знаки **&&**, **||** та **!**, застосовуються до значень логічного типу та породжують значення **false** або **true**. Їх означення наведено в таблиці.

Таблиця. Логічні операції

A	B	A && B	A || B	!A
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

Приклад. Вирази **(2<3) && (2<4)**, **(0>1) || (1>0)** та **!('a'=='x')** мають значення **true**, вирази **(2<-2) && true** та **!(5==5)** — значення **false**. ◂

Інструкції циклу з післяумовою є в багатьох мовах програмування, але в деяких їх умова розуміється як умова завершення (а не продовження) циклу. Тому при «перекладі» таких програм на інші мови програмування ***можуть виникати непорозуміння***.

- Кожен цикл з післяумовою ***можна замінити циклом з передумовою***, який в усіх мовах програмування розуміється однаково.

Наприклад, код з прикладу 1 (с. 80) набуде такого вигляду.

```
k=1;   // щоб ініціювати введення, присвоїмо
       // змінній k "неправильне" значення
while (!(10<=k && k<=99)){
    cout << "Enter one integer in [10,99]>";
    cin >> k;
}
// у кінці циклу так само 10<=k && k<=99
```

6.1.5. Переривання та продовження циклу

Вперше інструкцію **break** було представлено в розділі 4.5, де за її допомогою закінчувалося виконання інструкції-перемикача. Виконання цієї інструкції всередині циклу будь-якого різновиду переривает й завершує цикл; далі виконуються дії, наступні за цим циклом.

Якщо **break** записано в інструкції циклу, яку вкладено в іншу інструкцію циклу, то виконання **break** завершує вкладений цикл, а зовнішній цикл продовжується. Інструкція **continue** всередині циклу задає перехід на кінець тіла циклу. В інструкціях циклу з перед- та післяумовою слідом за **continue** обчислюється умова продовження циклу.

Приклад. За допомогою клавіатури вводиться послідовність дійсних чисел. Потрібно підрахувати суму її додатних елементів, а за появи 0 видати накопичену суму й завершити роботу.

Запрограмуємо цикл, в якому вводиться та обробляється послідовність чисел. Уведені числа зберігаємо в змінній **x**, а суму додатних елементів — у змінній **sum**. Якщо за введення трапилася помилка, то подальші дії з уведення не виконуються, а змінна **x** зберігає своє останнє значення (див. с. 34). Тому умовою продовження циклу буде саме відсутність помилок уведення. (Інакше можна отримати цикл, який ніколи не завершиться!) Цю умову задає значення виразу введення **cin>>x**, перетворене до логічного типу.

```
#include <iostream>
using namespace std;
int main()
{ double x;
  double sum=0;
  cout<<"Enter reals:\n";
  while (cin>>x){
    if (x==0.) break;     //виходимо з циклу
    if (x<0.) continue;   //пропускаємо від'ємні
    sum+=x;
  }
```

ву обчислюється умова. На відміну від інструкції циклу з передумовою, цикл ***починається діями в тілі*** й закінчується обчисленням умови.

Циклу з післяумовою відповідає блок-схема на рис. 6.2.

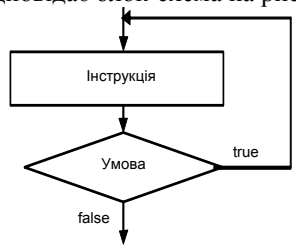


Рис. 6.2. Блок-схема інструкції циклу з післяумовою

- Умова перевіряється після виконання тіла циклу, тому її називають ***післяумовою***. Тіло циклу, заданого **do**-інструкцією, виконується обов'язково ***хоча б один раз*** (на відміну від **while**-інструкції).

Інструкцію циклу з післяумовою використовують, коли потрібно спочатку виконати тіло циклу, і лише потім перевіряти умову продовження.

Приклади

1. Потрібно з клавіатури ввести ціле число від 10 до 99. Якщо користувач набрав число за межами цього діапазону, слід ***повторити спробу***. Отже, спочатку треба вводити число, а потім перевіряти умову того, що число є двозначним.

```
do {
    cout << "Enter one integer in [10,99]>";
    cin >> k;
} while (!(10<=k && k<=99));
// 10<=k && k<=99
```

2. За двома натуральними числами n і m визначити, чи можна n представити як суму двох натуральних доданків, а m — як суму їх квадратів. Наприклад, $10 = 7+3$, $58 = 7^2+3^2$.

Доданків два, але для їх визначення достатньо одного циклу за можливими значеннями першого доданка x, а другий доданок визначається за умовою як $y = n-x$. Значення x перебираються, поки не знайдено розв'язок і x не більше y. Якщо після виходу з циклу x не більше y, то розв'язок знайдено, інакше розв'язків немає. Оформимо обчислення у вигляді функції, яка повертає ознаку успішності пошуку та надає значення двом параметрам-посиланням (нулі, якщо пошук неуспішний).

```
bool twoItems(int n, int m, int & x, int & y){
  x = 0;
  do {
    x++; y = n-x;
  } while (x<=y && x*x+y*y!=m);
  if (x<=y) return true;
  else { x=0; y=0; return false; }
}
```

◂

- Пріоритет порівнянь нижчий за пріоритет **+**, **–**, **<<** та вищий за пріоритет **&&**, **||**. Оператори **>**, **>=**, **<**, **<=** мають вищий пріоритет ніж **==** та **!=**, а **&&** — вищий ніж **||**. Докладніше пріоритети операторів див. у додатку Б.

Приклади

1. Враховуючи пріоритети операцій, вираз **(k>0)&&(d>0)** можна записати як **k>0&&d>0**, але перша форма запису сприймається людиною краще. Узагалі, намагайтеся записувати вирази так, щоб їх «зміст» не приховувався відсутніми дужками.

2. Математичну умову $x<y<z$ можна записати мовою C++ як **(x<y)&&(y<z)**. Запис **x<y<z** виражає зовсім іншу умову, а саме **(x<y)<z**. Обчислимо вираз **(-3<-2)<-1**. Значенням виразу **(-3<-2)** є **true**. Далі обчислюється **true<-1**, яке починається з приведення аргументів порівняння до типу **int**, тобто в кінці обчислюється **1<-1** і його значенням є **false**. Проте ***з математичної точки*** зору запис -3<-2<-1 істинний.

3. Математичну умову $x=y=z$ того, що числа x, y, z попарно рівні між собою можна записати мовою C++ як **(x==y) && (y==z)**. Для дійсних змінних **x**, **y**, **z** вираз **x=y=z** еквівалентний виразу **x=(y=z)** і задає присвоювання значення змінної **z** змінним **y** та **x**; його результатом буде значення змінної **z** — деяке дійсне число. ◂

- Записуючи вирази, не зловживайте пріоритетами операцій! Час, що економиться на дужках у виразах, потім витрачається на розшифрування логіки виразів та пошук помилок.

Вправа 4.3. Обчислити значення виразу: а) **!true==0**; б) **!false||(false==1)**; в) **(2>1)&&!true**.

Вправа 4.4. Обчислити значення виразу **6<3&&7<5||3==5**.

Компілятори C++ забезпечують так зване ***ледаче***[12], або ***скорочене***, обчислення булевих операцій **&&** та **||**. Спочатку обчислюється їх перший операнд. Якщо для операції **&&** це **false**, то другий операнд обчислювати не треба, адже результатом все одно буде **false**. Аналогічно, якщо перший операнд операції **||** має значення **true**, то другий операнд не потрібен.

Приклад. За обчислення виразу **(2*2==5)&&(323345%2209==37)** буде обчислено лише **2*2==5** (хибність), а виразу **(2*2==4)||(323345%2209==37)** — лише **2*2==4** (істина). ◂

Вправа 4.5. Що буде виведено на екран за виконання такої послідовності інструкцій?

```
int a=3; bool b=(2<3)||(a=-1); cout<<a<<endl;
```

[12] Принцип ледачих обчислень: «Обчислюю тільки те, що треба». Девізом же енергійних обчислень є: «Обчислюю все, що можу».

4.1.3. Умовні вирази

Вирази, які можуть бути або істинними, або хибними, називаються ***умовними***, або ***умовами***. У програмуванні вони відіграють особливу роль, оскільки є основою для прийняття рішень з вибору подальшого шляху обчислень. Значення умовного виразу належить логічному типу (або може бути приведеним до нього).

Наприклад, умовою того, що рівняння $ax+b=0$ з коефіцієнтами a та b має єдиний розв'язок, є нерівність $a \neq 0$. Залежно від значення умови **true** або **false** можна обчислити розв'язок рівняння або з'ясувати, чи має воно нескінченно багато розв'язків, чи не має жодного (для цього може знадобитися умова $b = 0$).

Умову часто використовують як ***ознаку*** деякої властивості. Ознака істинна, якщо властивість має місце, інакше хибна. Наприклад, умова $a \neq 0$ є ознакою того, що рівняння $ax+b=0$ має один розв'язок.

Найпростіші умови дуже часто мають вигляд порівнянь. Наприклад, умовою того, що значення цілої змінної **a** парне (ділиться на 2 без остачі), є те, що воно дає остачу 0 від ділення на 2: **a%2==0**.

Вправи

4.6. Виразити умову того, що значенням змінної **a** символьного типу є: а) цифра **1**; б) буква **h**.

4.7. Виразити умову того, що значення цілої змінної **b** ділиться на 3 без остачі.

4.8. Виразити умову того, що значення цілих змінних **a** і **b** мають однакову парність.

4.9. Дійсні змінні **a**, **b** задають кінці відрізку дійсної прямої. Написати вираз, який задає ознаку того, що довжина цього відрізку менше або дорівнює 0,001.

Складніші умови формулюють як системи або сукупності, складені з простіших умов. ***Системи умов*** записують мовою C++ за допомогою операції **&&**, ***сукупності*** — за допомогою **||**.

Приклади

1. Умовою того, що число x належить проміжку $[a; b]$, є математичний вираз $a \le x \le b$. Виразимо її так: **(a<=x) && (x<=b)**. Аналогічно, умову того, що значення змінної **char c** є десятковою цифрою, можна записати як **('0'<=c)&&(c<='9')**.

2. Припустимо, що значення числових змінних **a**, **b**, **c** представляють довжини відрізків. Умовою того, що з відрізків можна утворити трикутник, є система нерівностей:

$a > 0$, $b > 0$, $c > 0$, $a+b > c$, $a+c > b$, $b+c > a$.

Цій системі відповідає такий вираз (дужки не обов'язкові, але додають наочності).

```
(a>0) && (b>0) && (c>0) && (a+b>c) && (a+c>b) && (b+c>a)
```


Приклади

1. Інструкція **y=x++;** рівносильна **y=x; x=x+1;**, а інструкція **y=++x;** — **x=x+1; y=x;**. Якщо змінна **x** мала значення **1**, то після **y=x++** значенням **y** буде **1**, а після **y=++x** — **2**. В обох ситуаціях значенням **x** стане **2**.

2. Цикл **while (k<=n){fact*=k; k=k+1;}** за допомогою оператора **++** можна записати в будь-якій з таких форм.

```
while (k<=n){ fact*=k; k++;}
while (k<=n){ fact*=k; ++k;}
```

◀

Операції **++** та **--** виконуються швидше ніж відповідні присвоювання вигляду **x=x+1** та **x=x-1**, тому рекомендується використовувати саме їх. Операції **++** та **--** застосовні до змінних будь-якого з базових типів, хоча найчастіше їх використовують з цілими змінними.

Скрізь, де немає необхідності використовувати старе значення змінної, рекомендується з виразів вигляду **n++** та **++n** вибирати **++n**, оскільки він виконується швидше й простіше.

Спосіб і порядок обчислення виразу залежить від компілятора, тому краще записувати операції збільшення або зменшення в окремих виразах або інструкціях, а не в складі інших виразів. Наприклад, значення виразів **(n++)*(n++)** та **(++n)*(++n)** у різних системах програмування навіть можуть відрізнятися. Гарантовано лише те, що до значення змінної **n** двічі додається 1.

Вправа 6.1. Що буде надруковано за виконання програми? Пояснити зв'язок між значеннями змінних **i** та **x** і **y**:

```
#include <iostream>
using namespace std;
int main(){
 int i=1, x=1, y=2;
 while (x<y){
  i++; x*=i; y*=2;
  cout<< i <<" "<< x <<" "<<
y <<endl;
  }
  system("pause"); return 0;
}
```

а)

```
#include <iostream>
using namespace std;
int main(){
 int i=1, x=1, y=2;
 while (i<=10){
  i++; x*=i; y*=2;
  cout << i <<" "<< x <<" "<<
y <<endl;
 }
 system("pause"); return 0;
}
```

б)

6.1.4. Інструкція циклу з післяумовою

Інструкція циклу з ***післяумовою***, або **do**-інструкція, має такий загальний вигляд.

```
do інструкція while (умова);
```

Слово **do** («виконувати») є ключовим. Інструкція циклу з післяумовою виконується так. Спочатку виконується тіло циклу, потім обчислюється умова. Якщо вона хибна, цикл завершується, інакше повторюється тіло й зно-

Умову в інструкції циклу називають ***умовою продовження***, оскільки, якщо вона істинна, виконання інструкції циклу продовжується. Цикл починається обчисленням умови, тому її ще називають ***передумовою***.

- Інструкції циклу з передумовою застосовують, як правило, коли кількість повторень циклу наперед ***невідома***.

Приклади

1. За цілим $a \geq 0$ обчислити найменше n, за якого $n! > a$.

Будемо в циклі обчислювати послідовні значення $n!$, поки вони не стануть більше a. Коли цю умову буде порушено, n буде на 1 більше потрібного значення.

```
int minArgFact(int a)
{
  int n=1, fact=1;
  while (fact<=a) { fact*=n; n=n+1;}
  // fact=(n-1)!, fact > a
  return n;
}
```

Зауважимо, що за $a \geq 12!$ (це число порядку 500 мільйонів) замість 13! обчислюється помилкове значення, оскільки 13! не є числом типу **int**.

2. Обчислити суму цифр заданого натурального числа.

Щоб знайти суму цифр заданого числа n, можна діяти так. Спочатку сума дорівнює 0. Далі беремо молодшу цифру числа, додаємо її до суми та викреслюємо з числа. Повторюємо ці дії, поки в числі є цифри. Молодша цифра числа n є значенням виразу **n%10**, її «викреслення» можна представити як **n/=10**, наприклад, **123%10** = **3**, **123/10** = **12**. Оформимо розв'язання функцією з параметром **n**, яка повертає обчислену суму цифр.

```
int digitsSum(int n)
{ int sum = 0; // початкова сума цифр
  while (n>0)
    { sum+=n%10; n/=10; }
  // n=0, у sum накопичено суму цифр
  return sum;
}
```

6.1.3. Збільшення та зменшення

У циклічних обчисленнях дуже часто використовуються присвоювання вигляду **x=x+1** та **x=x-1**. Їх можна задати в скороченій формі за допомогою одномісних операторів ***збільшення*** (інкременту) **++** та ***зменшення*** (декременту) **--**. Ці оператори (та відповідні операції) мають ***префіксну*** (**++x**, **--x**) та ***постфіксну*** (**x++**, **x--**) ***форми***.

Вираз із постфіксним оператором **x++** або **x--** змінює значення змінної **x** на **1**, але значенням самого виразу є значення **x** ***перед зміною***. Вираз із префіксним оператором **++x** або **--x** теж змінює **x** на **1**, але значенням виразу є значення **x**, отримане ***після зміни***. Ці відмінності виявляються, коли оператори **++** та **--** застосовуються всередині виразів.

3. Той факт, що цілі ненульові числа a, b, c утворюють геометричну прогресію, математично можна виразити як $\frac{b}{a}=\frac{c}{b}$. Результатом ділення цілих у C++ є ціле, тому вираз **(b/a)==(c/b)** взагалі ***не відповідає*** математичній умові. Вираз **(double(b)/a)==(double(c)/b)** вирішує проблему з діленням цілих, але теж ***не є прийнятним***. Справа в тому, що тип **double** може містити лише наближення справжніх числових значень $\frac{b}{a}$ та $\frac{c}{b}$, тому результат порівняння значень типу **double** може відрізнятися від математичного порівняння. Щоб уникнути цього, запишемо вираз у математично еквівалентній формі: $bb = ac$. Отже, умова того, що числа a, b, c утворюють геометричну прогресію, у C++ має вигляд **b*b==a*c**.

- Перш ніж записувати математичну умову мовою програмування, спробуйте провести математичні перетворення.

Вправи

4.10. Виразити умову того, що значенням змінної **c** символьного типу є:
а) велика латинська літера (від **'A'** до **'Z'**); б) шістнадцяткова цифра.

4.11. Виразити умову того, що значення дійсних змінних **x**, **y**, **z** попарно різні.

4.12. Виразити умову того, що ціле **a** ділиться на ціле **b** без остачі. (Зверніть увагу на випадок, коли значення **b** дорівнює **0**.)

4.13. Написати умову того, що відрізки $[a, b]$ та $[c, d]$ осі Ox:
а) не мають спільних точок; б) мають хоча б одну спільну точку.

4.14. Написати умову того, що точки з координатами (**x1**, **y1**) та (**x2**, **y2**): а) збігаються; б) не збігаються.

4.15. Написати умову того, що точки з координатами (**x1**, **y1**), (**x2**, **y2**) та (**x3**, **y3**) лежать на одній прямій.

4.16. Пряму на площині можна задати дійсними коефіцієнтами **a**, **b**, **c** рівняння $ax+by+c = 0$. Написати умову того, що дійсні числа **a**, **b**, **c** задають:
а) пряму; б) вертикальну пряму; в) горизонтальну пряму;
г) пряму, яка проходить через початок координат.

4.17. Пряму на площині задано дійсними коефіцієнтами **a**, **b**, **c** рівняння $ax+by+c = 0$. Написати умову того, що точки з координатами (**x1**, **y1**) та (**x2**, **y2**) лежать у різних півплощинах відносно заданої прямої.

4.18. Написати умову того, що дві прямі, задані трійками коефіцієнтів рівняння $ax+by+c = 0$: а) збігаються; б) паралельні; в) паралельні й не збігаються; г) перетинаються; д) перпендикулярні.

4.19. Дійсні змінні **a**, **b** задають коефіцієнти рівняння $ax+b = 0$. Виразити умову того, що воно:
а) має єдиний розв'язок; б) не має жодного розв'язку; в) має нескінченно багато розв'язків.

4.20. Дійсні змінні **a**, **b**, **c** задають коефіцієнти рівняння $ax^2+bx+c=0$. Виразити умову того, що воно:

а) має рівно два дійсні корені; б) має єдиний дійсний корінь;

в) не має жодного розв'язку; г) має нескінченно багато розв'язків.

4.21. Написати умову того, що прямокутну цеглину з довжинами ребер **a**, **b**, **c** можна просунути в прямокутне вікно **x** на **y** так, щоб її грані були паралельні сторонам вікна.

4.1.4. Операція розгалуження

Мова C++ має операцію **?:**, яка називається ***операцією розгалуження***, або ***умовною***, й використовує умови. Вираз розгалуження виглядає так.

```
вираз_1 ? вираз_2 : вираз_3
```

За його виконання обчислюється значення першого виразу та приводиться до логічного типу. Якщо це **true**, то значенням виразу розгалуження буде значення другого виразу, інакше — третього виразу. В обох випадках значення приводиться до типу, який є більшим з типів другого та третього виразів.

Пріоритет операції розгалуження нижче ніж у логічних операцій, але вище ніж у присвоювання.

Приклади

1. Значенням виразу **(x>=0?x:-x)** є модуль числового значення змінної **x**.

2. Значенням виразу **(a<b?a:b)** є мінімальне зі значень числових змінних **a** та **b**. ◂

Вправа 4.22. Написати вираз, значенням якого є максимальне зі значень числових змінних **a** та **b**.

4.2. Інструкції розгалуження

Рівняння $ax+b=0$, залежно від конкретних значень a, b, може розв'язуватися одним з трьох шляхів. Шлях обирається після визначення, чи справджується умова $a \neq 0$; якщо це не так, то чи дійсна умова $b = 0$. Розглянемо засоби, які дозволяють указати вибір шляху обчислень залежно від тих або інших умов, і скористаємося ними в розв'язанні рівняння.

Вибір одного з двох можливих шляхів обчислення можна задати за допомогою ***інструкції розгалуження*** (***умовної інструкції***). Інструкція розгалуження в ***повній формі*** має такий вигляд.

```
if (умова) інструкція1 else інструкція2
```

Слова **if** і **else** є зарезервованими, дужки навколо умови обов'язкові. Цій інструкції відповідає блок-схема на рис. 4.1, *а*. Інструкція виконується так. Обчислюється значення умови. Якщо це **true**, виконується ***інструкція1*** і виконання закінчується. Якщо ж це **false**, то виконується ***інструкція2***, записана після **else**. Обидві ці інструкції довільні (присвоювання, розгалуження або інші, представлені нижче).

Скорочена форма інструкції розгалуження має такий вигляд.

```
if (умова) інструкція
```


Глава 6. Циклічні алгоритми

6.1. Прості інструкції повторення обчислень

6.1.1. Циклічне обчислення факторіала

Функція факторіала $n!$ невід'ємного цілого числа n визначається формулою $n! = (n-1)!\cdot n$ за $n > 0$, $0! = 1$. Звідси $n! = 1\cdot 2\cdot\ldots\cdot(n-1)\cdot n$. Щоб обчислити цей добуток за $n \geq 0$, можна почати з добутку 1 і далі послідовно отримувати добутки $1\cdot 1$, $1\cdot 1\cdot 2$, $1\cdot 1\cdot 2\cdot 3$, …, $1\cdot 1\cdot 2\cdot\ldots\cdot(n-1)\cdot n$, щоразу збільшуючи останній множник.

У цих міркуваннях присутні поняття ***добуток*** і ***останній множник***. Представимо їх цілими змінними **fact** і **k**. За означенням, за **k** = 0 маємо **fact** = 1. Візьмемо наступне значення **k** = 1. Отже, починаємо зі значень **fact=1**, **k=1**, а далі на кожному кроці множимо **fact** на **k**, зберігаємо добуток у **fact** і збільшуємо **k** на 1. Кроки повторюємо, поки **k** ≤ *n*.

```
fact=1; k=1;
поки (k<=n) повторювати { fact*=k; k=k+1;}
```

Мовою C++ це виглядає так.

```
fact=1; k=1;            // початок обчислень
while (k<=n)            // повторення кроків
  { fact*=k; k=k+1; }
// після закінчення повторень k=n+1, fact=n!
```

6.1.2. Інструкція циклу з передумовою

У наведеному прикладі циклічні, тобто повторювані обчислення задано за допомогою ***інструкції циклу з передумовою*** (**while**-інструкції) . Вона має такий загальний вигляд.

```
while (умова) інструкція
```

Слово **while** є зарезервованим, дужки обов'язкові, **while (*умова*)** — це ***заголовок циклу***, а ***інструкція — тіло***.

Інструкція циклу виконується так. Спочатку обчислюється умова в заголовку. Якщо вона істинна, виконується тіло циклу й знов обчислюється умова. Якщо вона істинна, все повторюється. Виконання інструкції циклу закінчується, коли обчислено значення умови **false**, тобто хибність. Отже, в останньому циклі тільки обчислюється умова, а тіло не виконується. Якщо при першому обчисленні умова хибна, то тіло циклу не виконується жодного разу. Перевірку умови циклу та виконання після неї тіла циклу інколи називають ***ітерацією циклу***.

Інструкції циклу з передумовою відповідає блок-схема на рис. 6.1.

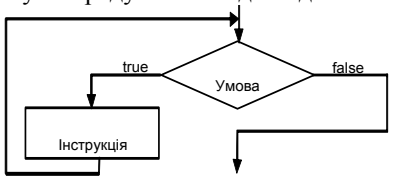


Рис. 6.1. Блок-схема інструкції циклу з передумовою

Однойменні функції повинні мати різні кількості параметрів або різні послідовності типів значень, що повертаються, та параметрів. При цьому відмінність таких функцій лише типом значень, що повертаються, є помилкою.

```
int f(double, double);
double f(double, double); // помилка!
```

У викликах функції типи аргументів можуть відрізнятися від типів параметрів. Для виконання виклику компілятор обирає ту з однойменних функцій, кількість і типи параметрів якої збігаються з кількістю та типами аргументів у виклику. Якщо такої немає, то обирається функція, прототип якої «досягається» шляхом автоматичних перетворень типів аргументів у виклику. Якщо цей вибір не є однозначним, компілятор повідомляє про помилку. Наприклад, за наявності таких прототипів

```
int f(double, int);
int f(int, double);
```

виклик **`f(1,2)`** не дозволяє компілятору визначити функцію для виконання виклику, і для компілятора це ***помилка***. Отже, подібних ситуацій слід уникати.

Контрольні запитання

5.1. Що таке підпрограма й що таке функція в мові C++?
5.2. Що таке виклик функції?
5.3. Що таке прототип функції?
5.4. Які елементи заголовку функції можуть бути відсутні в її прототипі?
5.5. Що таке формальні та фактичні параметри (параметри та аргументи)?
5.6. Назвіть особливості в записі та викликах функцій, що не повертають значень.
5.7. Чим відрізняється підстановка аргументів на місце параметрів-значень та на місце параметрів-посилань?
5.8. Чи можна використовувати ім'я, оголошене у функції, в інших функціях, записаних після неї?
5.9. В яких ситуаціях параметр функції має бути параметром-посиланням?
5.10. Що таке переозначення імені?
5.11. Чи можуть заголовки переозначених функцій відрізнятися лише типом значень, що повертаються?

Задачі

5.1. Написати програму, яка за трьома точками площини, визначає, чи утворюють вони гостро-, прямо- або тупокутний трикутник. (Вказівка: можна використати елементи задачі 4.2 на с. 65).
5.2. Модифікувати програму задачі 4.3 (с. 65) так, щоб для введення рівняння, знаходження його розв'язків та повідомлення результату використовувалися функції.
5.3. Написати програму, яка для прямої $ax+by+c=0$ визначає, чи лежать точки з координатами (**`x1`**, **`y1`**) та (**`x2`**, **`y2`**) у різних півплощинах відносно неї.
5.4. Написати програму, яка визначає, чи мають дві прямі на площині спільні точки.

Якщо обчислення умови дає значення **`false`**, то виконання інструкції розгалуження закінчується (див. блок-схему на рис. 4.1, *б*), інакше виконується ***`інструкція`***.

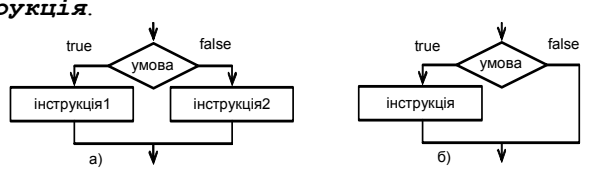


Рис. 4.1. Блок-схеми двох форм інструкції розгалуження

Приклад. Розглянемо такі інструкції.

```
int n, z;
cin >> n;
if (n%2==0) z=1; else z=-1;
```

Якщо введено невід'ємне значення **`n`**, то обчислення виразу **`n%2==0`** (перевірка умови) дає значення **`true`**, і змінна **`z`** отримує значення **`1`**. Якщо ж уведене значення від'ємне, то вираз **`n%2==0`** має значення **`false`**, і значенням **`z`** стає **`-1`**.

Замість інструкції

```
if (n%2==0) z=1; else z=-1;
```

можна записати такі.

```
z=-1; if (n%2==0) z=1;
```

Якщо значення **`n`** непарне, то значенням **`z`** залишиться **`-1`**.

Зазначимо, що наведений фрагмент коду обчислює значення математичного виразу $(-1)^n$ та присвоює його змінній **`z`**. ◂

Щоб програма легше сприймалася, інструкцію розгалуження часто записують так.

```
if (умова)                    або      if (умова)
    інструкція1                            інструкція1
else інструкція2                       else
                                           інструкція2
```

Приклад. Напишемо програму, яка розв'язує рівняння $ax+b=0$.

Уточнення постановки задачі. Визначимо вхідні та вихідні дані програми. *Вхід*: коефіцієнти рівняння — два дійсних числа *a* та *b*. *Вихід*: кількість розв'язків та, якщо розв'язок один, то цей розв'язок.

Математичний аналіз задачі. За умови $a \neq 0$ рівняння має один розв'язок — $-b/a$; за умови $a = 0$, якщо $b = 0$, то рівняння має нескінченно багато розв'язків, інакше — жодного.

Отже, всі вхідні дані є коректними, тому обробка помилок не потрібна.

Проектування програми. Найчастіше, алгоритм створюють, поступово уточнюючи поняття, пов'язані з задачею, та потрібні дії. Тоді кажуть, що розробку ведуть ***згори донизу***. У найбільш загальному вигляді алгоритм такий.

1. Отримати вхідні дані.
2. Обробити вхідні дані.
3. Вивести результат обробки.

Уточнимо кожен з кроків алгоритму.

«Отримати вхідні дані».

1.1. Вивести запрошення на введення даних.

1.2. Увести коефіцієнти рівняння в дійсні змінні **a** та **b**.

«Обробити вхідні дані». На основі математичного аналізу задачі, якщо $a \neq 0$, то кількість розв'язків дорівнює 1, а розв'язком є $-b/a$. Інакше, якщо $b = 0$, то кількість розв'язків нескінченна, інакше дорівнює 0. Кількість розв'язків присвоїмо цілій змінній **n**, при цьому значення **-1** зображатиме «нескінченно багато розв'язків». Розв'язок запам'ятаємо в дійсній змінній **x**.

«Вивести результат обробки».

3.1. Вивести рівняння, уведене користувачем.

3.2. За допомогою значень змінних **n** і **x** вивести кількість розв'язків рівняння та вивести розв'язок, якщо він один.

Нарешті, можна кодувати.

```
//програма, що розв'язує рівняння ax+b=0
#include <iostream>
using namespace std;
int main(){
  double a=0, b=0;
  int n;      //кількість розв'язків рівняння ax+b=0;
              // -1 позначає "нескінченно багато"
  double x;  // розв'язок рівняння
// отримати вхідні дані
  cout<<"Enter coefficients a and b of equation ax+b=0 (2
reals)\n";
  cin>>a>>b;
// обробити введені дані
  if (a!=0) n=1;
  else if (b==0) n=-1;
       else n=0;
  if (n==1) x=(-b)/a;
// повідомити результат
  cout<<"Equation "<<a<<"x";
  if(b>=0) cout<<"+"<<b;
  else cout<<b;
  cout<<"=0 ";
  if (n==1) cout<<"has one solution "<<x<<endl;
  else if (n==0) cout<<"has no solution\n";
  else cout<<"has each real as a solution)\n";
  system("pause"); return 0;
}
```

prog007.cpp ◀

У цій програмі кожен фрагмент коду задає певні дії для отримання необхідного результату, тобто має своє призначення, або свій ***обов'язок***. На перший погляд, програму можна зробити коротшою: якби обчислювати та відразу виводити кількість розв'язків, то перевірки умов скоротилися б удвічі. Але тоді код обробки даних був би ***перевантажений обов'язками***, тобто відповідав за кілька різних функцій (тут це обчислення та виведення на екран).


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

prog012.cpp

При виконанні першого та третього викликів **swap** її параметри **x** і **y** позначають змінні **a** й **b**, при виконанні другого — **b** й **c**.

5.3.5. Що вибирати?

Наведемо міркування, які дозволяють визначати потрібний різновид параметра функції — параметр-значення чи параметр-посилання.

- Якщо після виклику підпрограми використовується ***старе значення*** аргументу, або аргументом може бути довільний вираз, йому має відповідати ***параметр-значення***.
- Якщо після виклику підпрограми має використовуватися ***нове значення*** аргументу, отримане саме під час виконання виклику, то параметр має бути ***параметром-посиланням***.

Вправа 5.13. Написати функцію, яка вводить коефіцієнти квадратного рівняння $ax^2+bx+c = 0$. (Зверніть увагу: має виконуватись умова $a \neq 0$.)

Вправа 5.14. Написати функцію, яка вводить коефіцієнти a, b, c рівняння прямої $ax + by + c = 0$. (Коефіцієнти a й b не повинні одночасно бути нульовими.)

5.4. Переозначення функцій

Приклад. Припустимо, у програмі потрібні дві функції: одна має обчислювати максимум значень двох цілих параметрів, інша — трьох. Можна написати функції з іменами, наприклад, **max2** і **max3**. Проте мова C++ дозволяє дати їм одне й те саме ім'я **max**, що виглядає природніше. Розглянемо ці функції у такій програмі.

```
#include <iostream>
using namespace std;
int max(int, int);      // два різних прототипи
int max(int, int, int); // однойменних функцій
int main()
{ int a, b, c;
  cout << "Enter three integers>";
  cin >> a >> b >> c;
  cout << max(a,b) << " " << max(a,b,c) << endl;
  system("pause"); return 0;
}
int max(int x, int y) // два параметри
{return x>y ? x : y;}
int max(int x, int y, int z) // три параметри
{int t=max(x,y); return t>z ? t : z;}
```

prog013.cpp ◀

- Оголошення різних об'єктів програми з тим самим ім'ям називається ***переозначенням імені***.
- У мові C++ дозволені переозначення функцій та заборонені переозначення змінних.

Вправа 5.11. Що буде надруковано за виконання програми?

```
#include <iostream>
using namespace std;
int f(int a) { a++; return a; }
int g(int &x) { x=2; return x; }
int main(){
  int a=3, b=44;
  cout << f(a);
  cout<<' '<< g(b);
  cout<< ' ' << a << ' '<< b;
  system("pause"); return 0;
}
```

а)

```
#include <iostream>
using namespace std;
int f(int& a) {a+=4; return a; }
int g(int x) { x=3; return x; }
int main(){
  int a=2, b=1;
  cout << f(a);
  cout << ' ' << g(b);
  cout << ' ' << a << ' '<< b;
  system("pause"); return 0;
}
```

б)

Вправа 5.12. Що буде надруковано за виконання програми?

```
#include <iostream>
using namespace std;
int a=1, b=1, c;
int f(int x, int &y)
{ x++; y+=2; c=a+b; return x*y; }
int main(){
  cout<< f(a,b);
  cout<<' '<<a<<' '<<b<<' '<<c;
  system("pause"); return 0;
}
```

а)

```
#include <iostream>
using namespace std;
int a=2,b=2,c;
int f(int & x, int y)
{ x++; y+=2; c=a+b; return x+y; }
int main(){
  cout<< f(a,b);
  cout<<' '<<a<<' '<<b<<' '<<c;
  system("pause"); return 0;
}
```

б)

5.3.4. Ще один простий приклад

Розглянемо задачу: увести три цілих числа та вивести їх, упорядкувавши за неспаданням.

Уведемо числа в змінні **a**, **b**, **c**, за потреби обміняємо місцями їх значення так, щоб справджувалися нерівності $a \le b$ та $b \le c$, й виведемо ці значення. Уточнимо дії з упорядкування значень.

```
if(a>b) обміняти місцями значення a і b; // a ≤ b
if(b>c) обміняти місцями значення b і c;
// b≤c, a≤c, тобто значення c - найбільше,
// але a≤b може стати хибним, тому:
if(a>b) обміняти місцями значення a і b. // a ≤ b
```

У цьому алгоритмі явною є підзадача «обміняти місцями значення двох змінних», яку треба розв'язати тричі, лише з різними парами змінних. Для цієї задачі напишемо функцію **swap** з двома параметрами-посиланнями.

```
#include <iostream>
using namespace std;
void swap(int&, int&); // прототип функції обміну
int main(){
  int a, b, c;
  cout << "Enter three integers: ";
  cin>>a>>b>>c;
  if(a>b) swap(a,b);
  if(b>c) swap(b,c);
  if(a>b) swap(a,b);
  cout<<a<<' '<<b<<' '<<c<<'\n';
  system("pause"); return 0;
}
```


- Якщо кожен фрагмент коду має своє, «персональне», призначення, то це, по-перше, робить загальну структуру програми прозорішою і, по-друге, полегшує модифікацію окремих частин програми.

У наведеному прикладі можна забажати змінити вихідне текстове повідомлення, і це не вплине на алгоритм обчислення результату. Отже, відокремлення обробки від виведення результатів є цілком обґрунтованим.

Приклад. Написати фрагмент коду, який за дійсним x обчислює значення $f(x)$ та присвоює його дійсній змінній y.

$$f(x)=\begin{cases} x, \text{ якщо } 3 \le x \le 7, \\ x+2, \text{ якщо } x > 7, \\ 2x, \text{ якщо } -5 < x < 3, \\ 0, \text{ якщо } x \le -5. \end{cases}$$

По-перше, перепишемо формулу обчислення $f(x)$ у еквівалентному вигляді.

$$f(x)=\begin{cases} 0, \text{ якщо } x \le -5, \\ 2x, \text{ якщо } -5 < x < 3, \\ x, \text{ якщо } 3 \le x \le 7, \\ x+2, \text{ якщо } 7 < x. \end{cases}$$

Безпосередньо за формулою запишемо такий фрагмент коду.

```
if (x<=-5) y=0;
if (-5<x && x<3) y=2*x;
if (3<=x && x<=7) y=x;
if (7<x) y=x+2;
```

Цей код є правильним, але ***неоптимальним*** за кількістю виконуваних операцій. Якщо значенням змінної **x** є **-9.**, то обчислюються всі чотири умови, хоча з математичної точки зору зрозуміло, що за істинності умови **x<=-5** решта умов хибні. Отже, модифікуємо фрагмент коду.

```
if (x<=-5) y=0;
else if (-5<x && x<3) y=2*x;
else if (3<=x && x<=7) y=x;
else if (7<x) y=x+2;
```

Тепер за значення **-9** обчислюється тільки перша умова. Нехай значенням змінної **x** є **0.1**. Тоді вираз **x<=-5** є хибним, і обчислюється вираз **-5<x && x<3**. Але **x<=-5** є хибним, тому **-5<x** є істинним! Отже значенням виразу **-5<x && x<3** є значення **x<3**. Міркуючи так само далі, отримуємо ще один варіант.

```
if (x<=-5) y=0;
else if (x<3) y=2*x;  //якщо ми тут, то -5<x є істинним
else if (x<=7) y=x;   //якщо ми тут, то 3<=x є істинним
else y=x+2;           //якщо ми тут, то 7<x є істинним
```

Зауважимо: такий фрагмент коду для нашої задачі є ***помилковим***.

```
if (x<=-5) y=0;
if (x<3) y=2*x;
if (x<=7) y=x;
else y=x+2;
```

Якщо значенням **x** є **1.0**, то умова **x<3** істинна, і спочатку виконується присвоювання **y=2*x**. Проте потім перевіряється умова **x<=7**, виявляється істинною, і виконується **y=x**, що, вочевидь, є помилковим. ◂

Дуже часто інструкції розгалуження є частиною інших розгалужень, тому їх записують «східцями», зсуваючи вкладену інструкцію праворуч, наприклад, ось так.

```
if (умова1)
   if (умова2)
      інструкція1
   else
      інструкція2
else ...
```

Інколи виникають довгі ланцюги розгалужень, в яких за словами **else** слідують наступні розгалуження з **if** на початку. Краще записувати їх у такому вигляді.

```
if (умова)
  інструкція
else if (умова)
  інструкція
else if (умова)
  ...
```

Вправа 4.23. Що виводить програма, якщо введено: а) 1; б) 2; в) 3; г) 4.

```
#include <iostream>
using namespace std;
int main(){
  int x,y;
  cout<<"Enter one integer:"; cin >> x;
  if (x==1) y=16;
  else if (x==2) y=256;
  else if (x==3) y=4096;
  else y=10000;
  cout << y <<endl;
  system("pause"); return 0;
}
```

Вправа 4.24. Касиру потрібно видати деяку цілу суму грошей *n* монетами номіналом по 5 і 2 (яких є необмежений запас) та витратити якомога менше монет. Написати програму, яка обчислює та виводить кількості монет номіналом 5 і 2, які має видати касир. Якщо суму видати неможливо, вивести відповідне повідомлення.

4.3. Блок

Щоб написати кілька інструкцій там, де за правилами мови має бути одна, наприклад, в якості гілки в умовній інструкції, використовують ***блок*** — послідовність інструкцій у дужках **{}**. Він має такий загальний вигляд.

```
{
  інструкція
  ...
  інструкція
}
```

Виконання блоку полягає в послідовному виконанні інструкцій, записаних у ньому.

відбуваються в пам'яті, що відповідає аргументу, тобто ***є змінами аргументу***. Отже, у другому варіанті функції **f** під час виконання її виклику параметр **res** позначає змінну **c** головної функції, завдяки чому вона й отримує нове значення. У подібних випадках кажуть, що ***значення повертається за допомогою параметра-посилання***.

Приклад. Різницю між параметрами-значеннями та параметрами-посиланнями функції **f** схематично проілюстровано на рис. 5.1.

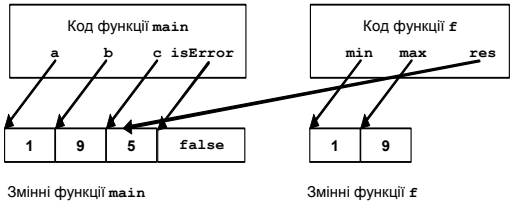


Рис. 5.1. Параметри-значення та параметри-посилання

Проілюструємо суттєві моменти виконання наведеної вище програми, за умови введення числа **33**. Стовпчики таблиці зображують змінні, тобто ділянки пам'яті, послідовні рядки — зміни стану пам'яті. Позначимо змінні, відповідні параметрам функції, додавши її ім'я: **f.min**, **f.max**, **f.res**. Оскільки **f.res** позначає ділянку пам'яті змінної **c**, то для спрощення в таблиці імені **c** та параметру **f.res** відповідає один і той самий стовпчик.

Що виконується	***Усі наявні змінні***					
Утворення змінних програми	**a**	**b**	**c**	**isERROR**		
a=1; b=9; isERROR=false;	**1**	**9**	**?**	**false**		
Виклик **f(a,b,c)**	**1**	**9**	**?**	**false**		
Утворення нових змінних	**1**	**9**	**f.res**	**false**	**f.min**	**f.max**
Присвоювання **min** і **max**	**1**	**9**	**?**	**false**	**1**	**9**
cin>>res;	**1**	**9**	**33**	**false**	**1**	**9**
if(...)...	**1**	**9**	**1**	**false**	**1**	**9**
Закінчення виклику **f**	**1**	**9**	**1**	**false**		
isERROR=f(a,b,c)	**1**	**9**	**1**	**true**		

Зауважимо: якби в тілі процедури було змінено **min** або **max**, це б не вплинуло на значення **a** або **b**, оскільки іменам **min** і **max** відповідають власні змінні.

Якщо ознака помилки, яку повертає виклик **f(a,b,c)**, далі не потрібна, інструкцію присвоювання з викликом **isERROR=f(a,b,c);** можна замінити інструкцією виклику функції **f(a,b,c);**. Значення, яке повертає функція, далі просто ігнорується. ◂

```
  res=min;
  return true;
}
```
Зазначимо, що після виконання інструкції **return false;** подальші інструкції з тексту функції вже не виконуються, тому писати для них секцію **else** не потрібно. Це «відсікання» оброблених варіантів дуже часто дозволяє уникати великої вкладеності інструкцій розгалуження **if**.

Отже, у функції **f** параметри **min** і **max** є параметрами-значеннями, **res** — параметром-посиланням. Наведемо фрагмент головної функції з викликом функції **f**.

```
int main()
{ int a, b, c;
  bool isERROR;
  a=1; b=9; isERROR=false;
  isERROR=f(a,b,c);
  ...
  return 0;
}
```

5.3.3. Підстановка аргументів на місце параметрів

У цьому підрозділі розглянемо, що відбувається з параметрами-значеннями та параметрами-посиланнями під час виконання виклику функції.

Параметр-значення. На початку виконання виклику утворюється нова змінна, відповідна параметру-значенню, тобто під час виконання виклику ***параметр-значення позначає власну змінну***. Аргументом у виклику, відповідним параметру-значенню, може бути ***довільний вираз***. Значення аргументу обчислюється й присвоюється цій змінній.

- Описаний спосіб передачі даних у функцію називається підстановкою аргументу на місце параметра ***за значенням***.

Коли виконуються інструкції з тіла функції, усі зміни, що стосуються параметра-значення, відбуваються в його власній змінній і не впливають на пам'ять, пов'язану з аргументом у виклику. Отже, у функції **f** з прикладу параметри **min** і **max** позначають власні ділянки пам'яті, ніяк не пов'язані зі змінними **a** і **b** головної функції.

Параметр-посилання. У виклику аргумент, відповідний параметру-посиланню, повинен задавати ***змінну того ж типу, що й у параметра***. У найпростішому випадку це ім'я змінної, тип якої збігається з типом параметра. На початку виконання виклику визначається посилання на цю змінну (фактично, її адреса) і передається до функції, тому під час виконання виклику функції параметр-посилання позначає змінну, яка «належить» аргументу і всі ***виконувані над параметром-посиланням дії виконуються над аргументом***.

- Описаний спосіб передачі даних у підпрограму називається підстановкою аргументу на місце параметра ***за посиланням***.

Коли виконуються інструкції з тіла функції, зміни параметра-посилання

Приклад. У прикладі зі с. 57, в обробці введених даних, кількість розв'язків рівняння та розв'язок можна обчислити разом в одному блоці (і це відповідає математичній розробці алгоритму розв'язання).

```
if (a!=0) {n=1; x=(-b)/a;}
else if (b==0) n=-1;
else n=0;
```
◂

4.4. Область дії оголошення імені

Тіло головної, як і будь-якої іншої функції, є блоком. Блок містить послідовність інструкцій, які у свою чергу можуть містити блоки, і так далі. У кожному блоці можна оголошувати імена змінних та деяких інших об'єктів. Виникає питання про те, в яких місцях програми «видно» те чи інше оголошення, а в яких ні.

- ***Область дії оголошення імені*** — це сукупність місць у програмі, в яких це ім'я позначає саме те (змінну або інший об'єкт), що описано в оголошенні.

Область дії оголошення визначається таким правилом.

- Оголошення діє від місця запису до кінця блоку, в якому записано. Проте, якщо всередині цього блоку є ще один блок (***вкладений***), і в ньому оголошено це саме ім'я, то це «внутрішнє» оголошення діє до кінця вкладеного блоку. Іншими словами, оголошення імені в блоці «перекриває» оголошення цього ж імені за межами блоку.
- ***Зовнішнє оголошення*** (розташоване за межами будь-якого блоку) діє за цими самими правилами від місця запису до кінця файлу, в якому його записано.

Правило, що визначає область дії оголошення імені, дозволяє в різних частинах програми (точніше, у різних блоках) давати різним змінним однакові імена. Але жодне ім'я змінної не може бути оголошено в блоці більше одного разу.

Приклад. Розглянемо таку програму.

```
#include <iostream>
using namespace std;
int a=99;                // «зовнішня» змінна
int main(){
  cout << a << ' ';      // вихід: 99
  int a=1;               // змінна в блоці функції
  {                      // вкладений блок: початок
    cout << a << ' ';    // вихід: 1
    int a=2;             // ще одна змінна
    cout << a << ' ';    // вихід: 2
  }                      // вкладений блок: кінець
  cout << a << endl;     // вихід: 1
  system("pause"); return 0;
}
```
prog008.cpp

За її виконання буде виведено **99 1 2 1.** ◂

- Змінна, визначена зовнішнім оголошенням, є ***глобальною*** та ***статичною***. З точки зору C++, змінні, оголошені за межами блоків, є ***глобальними*** (в межах файлу). ***Статичними*** вважаються змінні, які створюються один раз на початку виконання програми й знищуються в кінці, тобто існують протягом усього виконання програми.

Не варто усі змінні «головної» функції програми робити глобальними (хоча це й можливо). Кожна частина програми повинна мати доступ тільки до тих змінних, які їй необхідні. Це дозволяє уникати їх неправильного використання (наприклад, коли змінні, що зображають певні сутності, «раптом» стають допоміжними в обчисленнях). Глобальні змінні найчастіше призначаються для збереження загальних налаштувань програми або її окремих частин. У сучасному програмуванні рекомендується за можливістю ***уникати глобальних змінних***, а для обміну даними між частинами програми застосовувати інші засоби.

Вправа 4.25. Що буде надруковано за виконання програми, якщо ввести: а) 0; б) 1?

```
#include <iostream>
using namespace std;
int main(){
  int a, b=99, c=777;
  cin >> a;
  if(a)
    { int b=1; cout << b << ' ' << c << ' '; }
  else
    { c=0; cout << b << ' ' << c << ' '; }
  cout << b << ' ' << c << '\n';
  system("pause"); return 0;
}
```

Вправа 4.26. Що буде надруковано за виконання програми, якщо ввести: а) 0; б) 1?

```
#include <iostream>
using namespace std;
int b=99;
int main(){
  int a, c=777;
  cin >> a;
  if(a)
    { int b=22; cout << b << ' ' << c << ' '; }
  else
    { c=0; cout << b << ' ' << c << ' '; }
  cout << a << ' ' << c << '\n';
  system("pause"); return 0;
}
```

4.5. Вибір з кількох варіантів

У ситуації, коли варіант шляху обчислень визначається одним з кількох значень цілого або символьного типу, обчислення можна описати за допомогою ***інструкції вибору варіанту***, або ***перемикача***. Розглянемо її на прикладі.

Головна функція спочатку виводить значення координат, якими їх проініціалізовано. Потім, за виконання виклику функції **inputPoint**, координати мали б отримати значення. Мали б, але ***не отримують*** — про це свідчать ті самі значення координат після виклику. За введення чисел **1**, **1** текст у вікні програми буде таким.

```
"Old value": (0, 0)
Enter point (real coordinates x and y): 1 1
"New value": (0, 0)
```

А тепер — ***увага!*** У заголовку функції перед іменем параметрів **x** та **y** додамо символ **&**.

```
void inputPoint(double &x, double &y)
```

Після цього, якщо ввести ті самі числа, нова програма дає такий вихід.

```
"Old value": (0, 0)
Enter point (real coordinates x and y): 1 1
"New value": (1, 1)
```

Як бачимо, за виконання виклику функції **inputPoint** змінні **x** і **y** отримали нове значення. Причина в тому, що зі знаком **&** параметри функції стали параметрами ***іншого різновиду***.

- Параметри, оголошені зі знаком **&** перед іменем, називаються ***параметрами-посиланнями***, а без нього — ***параметрами-значеннями***.
- Знак **&** можна записувати як окремо від імен навколо нього (**double & x**), так і разом з ними, наприклад, **double& x** або **double &x**.

Отже, у новій функції **inputPoint** параметри **x** і **y** є параметрами-посиланням.

- Якщо є вибір, використовувати параметри-посилання чи глобальні змінні, віддавайте перевагу параметрам-посиланням.

5.3.2. Складніший приклад

Розглянемо функцію, яка має ввести та повернути ціле число з заданого діапазону [*min*; *max*]. Якщо користувач уведе значення за межами діапазону, його потрібно замінити значенням лівої межі діапазону. На перший погляд здається, що цю функцію можна написати з таким прототипом.

```
int f(int min, int max);
```

Проте, коли введене значення заміняється на **min**, про це ***корисно повідомити***. Враховуючи це, змінимо прототип функції на такий (необхідні коментарі до нього напишіть самостійно).

```
bool f(int min, int max, int &res);
```

Отримане число повертається через параметр-посилання **res**, а функція повертає ознаку того, що були помилки за введення числа (**true**, якщо користувач задав число поза діапазоном, інакше **false**).

```
bool f(int min, int max, int &res)
{ cout<<"Enter integer in ["<<min<<", "<<max<<"]: ";
  cin >> res;
  if(min<=res && res<=max) return false;
```

Виклик **void**-функції записується в окремій ***інструкції виклику функції***, а не як операнд у виразі. Наприклад, виклик функції **writePoint** міг би бути таким.

```
...; writePoint(1.5, 2.3); ...
```

- У **void**-функції не може бути інструкцій повернення значення виразу, але можуть бути інструкції повернення без виразу, що мають вигляд **return;**.

Вправа 5.8. Написати функцію, яка за дійсними коефіцієнтами *a*, *b* рівняння $ax+b = 0$ виводить рівняння на екран.

Вправа 5.9. Написати функцію, яка за кількістю розв'язків *n* та самим розв'язком *x* рівняння $ax+b = 0$ виводить повідомлення про розв'язки рівняння на екран (див. приклад на с. 57).

Вправа 5.10. Написати функцію, яка виводить логічне значення у вигляді **Yes** (для **true**) або **No** (для **false**).

5.3. Параметри-значення та параметри-посилання

5.3.1. Простий приклад

Інструкція **return** здатна повернути з функції ***одне значення***. А що робити, коли у функції треба змінити ***кілька значень***? Наприклад, у задачі визначення за трьома точками, чи утворюють вони трикутник, потрібно ввести три точки. Ці повторювані дії доцільно оформити функцією, яка має вводити, а отже, й змінювати значення двох координат точки.

Для ілюстрації розглянемо простішу програму з викликом функції, яка має ввести координати точки й присвоїти їх двом своїм дійсним параметрам.

```
#include <iostream>
using namespace std;
void inputPoint(double x, double y)
{ cout<<"Enter point (real coordinates x and y): ";
  cin >> x >> y;
}
void writePoint(double x, double y)
{ cout << "(" << x << ", " << y << ")"; }
int main(){
  double x=0, y=0;
  cout << "\"Old value\": "; writePoint(x,y); cout<<endl;
  inputPoint(x,y);
  cout << "\"New value\": "; writePoint(x,y); cout<<endl;
  return 0;
}
```

prog011.cpp

Але якщо далі виникне потреба вивести ще пропуск після дужки, то запис **'('** призведе до непередбачуваних наслідків під час виконання, а запис **"("** — ні. Отже, обрано варіант, безпечніший для програміста.

Приклад. Розглянемо задачу: увести з клавіатури дійсне число, знак операції (**+**, **-**, ***** або **/**), ще одне число й надрукувати результат застосування операції до цих чисел. Наприклад, після введення **2**, **+**, **3** виводиться **5**, а після **2**, **/**, **3** — приблизно **0.667**.

Для спрощення припустимо, що користувач програми правильно вводить числа, але може набрати недопустимий знак операції. Ще однією помилкою може бути операнд **0** після знака **/**, тобто спроба поділити на 0.

Операнди представимо дійсними змінними **op1** та **op2**, знак операції — символьною **sign**. Результат операції збережемо в дійсній змінній **res**. Нехай ціла змінна **state** зображує стан процесу обчислення:

– за коректних вхідних даних її значенням є **0** (відсутність помилки);

– якщо введено недопустимий знак операції, її значенням стає **1**;

– якщо задано ділення на 0, її значенням стає **2**.

Спочатку, коли вхідних даних ще немає, то немає й помилки (не будемо наперед звинувачувати користувача ☺), тому початковим значенням **state** є **0**. Уведемо операнди та знак. Потім за значенням **sign** виберемо одну з операцій: додавання, віднімання, множення або ділення, та виконаємо обчислення, одночасно визначаючи стан процесу обчислення. У кінці повідомимо результати обробки. Вибір варіанту дій за знаком операції спочатку опишемо ланцюжком розгалужень.

```
// програма обчислення результату операції
// за її знаком (+,-,*,/) та операндами
#include <iostream>
using namespace std;
int main(){
   double op1, op2, res;
   char sign;
   int state=0;
// уведення
   cout << "Enter double, operator (+,-,*,/), and double\n";
   cin >> op1 >> sign >> op2;
// обчислення
   if (sign=='+') res=op1+op2;
   else if (sign=='-') res=op1-op2;
   else if (sign=='*') res=op1*op2;
   else if (sign=='/')
      if (op2!=0) res=op1/op2;
      else state=1;
   else state=2;
// повідомлення результату
   if (state==0)
      cout << "===\n" << res << endl;
   else if (state==1)
      cout << "Division by zero\n";
   else cout << "Wrong operator\n";
   system("pause"); return 0;
}
```

prog009.cpp

Розглянемо інший спосіб описати вибір варіанту обчислення. Замість інструкції розгалуження за значеннями **sign** можна написати таку.

```
switch(sign){
  case '+': res=op1+op2; break;
  case '-': res=op1-op2; break;
  case '*': res=op1*op2; break;
  case '/': if (op2!=0) res=op1/op2; else state=1; break;
  default: state=2;
}
```

Аналогічно, інструкцію повідомлення запишемо так.

```
switch(state) {
  case 0: cout << "===\n" << res<<endl; break;
  case 1: cout << "Division by zero\n"; break;
  default: cout << "Wrong operator\n";
}
```

Слова **switch**, **case**, **default** є зарезервованими; вони позначають відповідно «перемикач», «випадок», «за відсутності». Інструкція **break;** закінчує виконання інструкції, в якій її записано (тут це вся наша інструкція-перемикач). Вираз у дужках після слова **switch** (у прикладі це ім'я **sign**) називається ***селектором варіантів***.

За виконання інструкції вибору спочатку обчислюється значення селектора. Потім воно послідовно порівнюється зі значеннями (***мітками варіантів***), які вказано після слів **case**. Тільки-но значення селектора збігається з міткою, виконується послідовність інструкцій, записана після цієї мітки й двокрапки, до найближчої інструкції **break**, яка закінчує виконання всієї інструкції вибору варіанту. Якщо значення селектора не збіглося з жодною міткою варіанту, то виконуються інструкції, записані після слова **default**, а якщо слова **default** немає, виконання перемикача закінчується.

Селектор варіантів є виразом цілого (*integral*) типу, або такого, який однозначно приводиться до цілого, мітки варіантів — константними значеннями того самого типу, що й у селектора.

Варіант з «міткою» **default** можна записати будь-де, але рекомендується записувати його останнім.

Послідовність операторів після мітки варіанту може бути порожньою. За збігу значення селектора з цією міткою виконуються найближчі оператори після наступних міток (або нічого не виконується, якщо ця мітка остання).

Якщо деякий варіант не містить інструкції **break**, то після операторів цього варіанту виконуються оператори, записані в подальших варіантах — до найближчого **break** або до кінця перемикача. В останньому варіанті писати **break** немає сенсу.

Вправа 4.27. Описати, як вихід залежить від значення змінної **char c** при виконанні такої інструкції.

зокрема, умови для її аргументів (фактичних параметрів);

– ***післяумови*** — умови, що виконуються *після* її виклику, зокрема, як функція змінює глобальні змінні програми;

– поведінку функції за некоректних фактичних параметрів.

- Пам'ятайте: прототип функції пишеться не тільки для компілятора, але й для програміста, який має використовувати функцію. Отже, прототип слід писати так, щоб, не читаючи тіла функції, можна було зрозуміти, як нею користуватися.

Запишемо прототип функції обчислення відстані зі с. 67.

```
double dist(double a1, double b1, double a2, double b2);
//distance from (a1,b1) to (a2,b2)
```

У прототипі, на відміну від справжнього заголовка функції, можна не вказувати імен параметрів. Зокрема, прототип функції **dist** з синтаксичної точки зору може мати вигляд.

```
double dist(double, double, double, double);
```

Проте він нічого не каже про призначення параметрів функції. Тому це, скоріше, приклад того, як ***не слід*** писати прототип функції, адже вдало названі параметри й сама функція дозволяють уникнути зайвих пояснень щодо її призначення. Замість скороченого імені **dist** краще було б узяти **distance** (*відстань*), але його означено в бібліотеках мови C++, тому залишимо **dist**.

- Кожен елемент програми (змінна, функція та інші) повинен мати ім'я, яким він позначається. Перш ніж користуватися елементом, необхідно його оголосити. Оголошення імені елемента лише описує його, але не створює сам цей елемент. Проте, щоб використовувати елемент програми, необхідно спочатку створити його в пам'яті програми. Створення елемента задається його означенням. Функція, складена заголовком і тілом, є означенням, оскільки дії, задані функцією, у вигляді машинних команд записуються в певну ділянку пам'яті. Прототип же лише повідомляє про функцію й не описує жодних дій, тому є оголошенням її імені. Означення та оголошення змінних розглядаються в розділі 9.2 на с. 114.

5.2.3. Функція, яка не повертає значень

Приклад. Точку площини задано двома її дійсними координатами. Напишемо функцію виведення координат точки у вигляді **(x, y)**, наприклад, **(1.5, 3.12)**.

Тут немає значення, яке було б потрібно повернути, тому функція нічого не повертає. У заголовку таких функцій замість імені типу записується слово **void** («вільний», «порожній»). Отже, функція виведення координат точки виглядає так.

```
void writePoint(double x, double y)
{ cout << "(" << x << ", " << y << ")"; }
```

[13]

[13] Замість рядкової константи **"("** можна було б вивести символ **'('**.

перетворений до типу значень, що повертаються. В іншому випадку виконання функції може мати непередбачувані наслідки.

Вправи

5.1. Написати функцію, яка повертає максимальне зі значень двох її дійсних параметрів.

5.2. Написати функцію з цілим параметром n, яка повертає значення $(-1)^n$.

5.3. Написати функцію з дійсним параметром x, яка повертає:
а) –1, 0, 1, відповідно, якщо $x < 0$, $x = 0$, $x > 0$; б) дробову частину x; в) результат округлення x до цілого.

5.4. Написати функцію, яка за значеннями дійсних змінних x і y обчислює значення z за формулою

$$z = \begin{cases} x + 2, & \text{якщо } 0 < x \le 3, \\ x - y, & \text{якщо } x \le 0, \\ x + y, & \text{якщо } x > 3. \end{cases}$$

5.5. Написати функцію, яка за значеннями дійсних змінних x і y обчислює значення $\log_x y$, якщо це можливо, а інакше повертає 0.

5.6. Написати функцію, яка перевіряє, чи можна утворити трикутник з трьох відрізків заданих довжин.

5.7. Написати функцію, яка за заданим цілим числом, що задає вік людини, визначає яке зі слів «рік», «роки», «років» треба до нього приписати відповідно до правил української граматики. Наприклад, 21 рік (результат 1), 34 роки (результат 2), 14 років (результат 3).

5.2.2. Прототип функції

Ім'я функції має бути оголошено вище за текстом програми, ніж воно використовується в інших функціях. Проте записувати всю функцію вище від її викликів не обов'язково — достатньо записати лише її заголовок.

Заголовок функції зі знаком «`;`» у кінці називається ***прототипом функції***. Прототип є ***оголошенням функції***, тобто повідомляє компілятору, що в програмі є така функція. Після оголошення функцію все одно необхідно означити, тобто вказати задані нею дії. ***Означення функції*** складається з заголовку та тіла.

Загальноприйнятою практикою програмування мовою C++ є запис прототипів функцій програми на її початку (зазвичай, після зовнішніх оголошень імен констант, типів і змінних). Після прототипів, як правило, записують головну функцію, а за нею означення оголошених та інших функцій. Порядок розташування означень оголошених функцій може бути довільним. Поруч з прототипом функції варто вказати:

– призначення функції, тобто що саме вона виконує та яким є зміст її параметрів;

– ***передумови*** — умови, які мають виконуватися *перед* її викликом,


```
switch(c) {
   case 'A': cout << 'A'; break;
   default: cout << '*';
   case 'B': case 'C': cout << "[B or C]";
}
```

Вправа 4.28. Написати програму, яка за номером місяця виводить пору року.

Контрольні запитання

4.1. Навести знаки операцій порівняння та логічних операцій.
4.2. Які значення в мові C++ зображають логічні значення «хибність» та «істина»?
4.3. За якими правилами обчислюються логічні операції?
4.4. При обчисленні яких операцій у мові C++ використовується стратегія ледачого обчислення?
4.5. Описати різновиди та порядок виконання інструкцій розгалуження.
4.6. Що таке блок?
4.7. Що таке область дії оголошення імені?
4.8. Описати правила, за якими визначається область дії оголошення імені.
4.9. Описати вигляд і порядок виконання інструкції вибору варіанту.
4.10. Чи може селектор варіантів мати дійсний тип?
4.11. Чи може селектор варіантів мати логічний тип?

Задачі

4.1. Написати програму, яка за дійсними x та y обчислює значення $\log_x y$.
4.2. Написати програму, яка за трьома дійсними числами — довжинами сторін трикутника, — визначає, чи є він гострокутним, прямокутним або тупокутним.
4.3. Написати програму, яка для рівняння $ax^2+bx+c = 0$ знаходить дійсні розв'язки та їх кількість.
4.4. Написати програму, яка до заданого цілого числа, що визначає вік людини, дописує слово «рік», «роки», «років» відповідно до правил української граматики. Наприклад, 21 рік, 34 роки, 14 років.
4.5. Найближча крамниця працює з 7.00 до 19.00 з перервою на обід з 13.00 до 15.00. Гастроном, що знаходиться в 20 хвилинах ходьби, працює з 8.00 до 20.00 та має перерву з 14.00 до 16.00. На відстані, подолати яку можна тільки за 45 хвилин, знаходиться супермаркет, який працює з 8.00 до 23.00 без перерви. Написати програму, яка за заданим часом визначає, що краще:
– відвідати найближчу крамницю;
– дійти до гастронома (з урахуванням часу на дорогу);
– рушати в супермаркет (теж з урахуванням часу на дорогу);
– залишитися вдома, оскільки всі магазини зачинено.
Час уводиться так: години, двокрапка, хвилини, наприклад, 15:30 означає 15 годин 30 хвилин.

Глава 5. Допоміжні функції, або підпрограми

5.1. Підпрограма — опис розв'язання підзадачі

Програма — це опис розв'язання деякої інформаційної задачі. Практично в кожній задачі можна виділити окремі ***підзадачі***, допоміжні до неї. Деякі підзадачі доводиться розв'язувати в багатьох різних задачах, наприклад, обчислення математичних функцій або введення та виведення даних. Для таких ***стандартних підзадач*** у кожній системі програмування є величезний набір ***готових підпрограм***, зібраних у спеціальні набори — ***бібліотеки***. Виконання такої бібліотечної підпрограми задається її ***викликом***, у якому вказано вирази або змінні, що мають оброблятися під час її виконання.

- У практичному програмуванні знати стандартні підпрограми корисно й необхідно, адже застосовувати готові деталі набагато легше, ніж створювати їх самому.

Проте запрограмувати розв'язання «всіх можливих підзадач» — утопія, тому програмістам доводиться постійно створювати власні підпрограми.

- Використання підпрограм втілює загальний принцип «розділяй і пануй», дозволяє створювати добре структуровані й зрозумілі програми, суттєво полегшує їх проектування та розробку.

5.2. Функція та її виклики

5.2.1. Приклад функції у програмі

Розглянемо засоби створення підпрограм мовою C++. У цій мові підпрограма називається ***функцією***. Програма, як правило, містить головну функцію та кілька допоміжних, які описують розв'язання підзадач основної задачі. Познайомимося зі створенням функцій на простому прикладі.

Приклад. Обчислити периметр трикутника на площині, заданого координатами його вершин.

Нехай (x_1, y_1), (x_2, y_2), (x_3, y_3) — координати вершин трикутника. Щоб обчислити його периметр, потрібні довжини трьох сторін, тобто відрізків з кінцями у вершинах: $\sqrt{(x_2-x_1)^2+(y_2-y_1)^2}$, $\sqrt{(x_3-x_1)^2+(y_3-y_1)^2}$, $\sqrt{(x_3-x_2)^2+(y_3-y_2)^2}$. Вони обчислюються, по суті, однаково, лише з різними парами точок. Але ці доволі громіздкі й дуже схожі вирази писати тричі не будемо, оскільки всі вони описують конкретні розв'язання такої ***загальної підзадачі***: обчислити довжину відрізка за чотирма координатами його кінців. У цій загальній підзадачі замість конкретних координат є їх позначення, тобто ***параметри*** підзадачі. Коли підзадача розв'язується з конкретними координатами, вони підставляються на місце параметрів і є ***аргументами*** в цьому ***конкретному розв'язанні***.

Опишемо розв'язання вказаної підзадачі у вигляді окремої функції з іменем **dist**, параметрами якої є чотири дійсні значення. У головній же функції напишемо три ***виклики*** функції **dist**, в яких вкажемо координати потрібних нам точок.

```
// обчислення периметра трикутника
#include <iostream>
#include <cmath>
using namespace std;
double dist(double a1, double b1, double a2, double b2)
//distance from (a1,b1) to (a2,b2)
{
  return sqrt((a1-a2)*(a1-a2)+(b1-b2)*(b1-b2));
}
int main()
{ double x1, y1, x2, y2, x3, y3;
  cout << "Enter three pairs of coordinates: ";
  // уведення конкретних координат
  cin>>x1>>y1>>x2>>y2>>x3>>y3;
  cout << "Perimeter = " <<
    dist(x1,y1,x2,y2)+  //виклик для першої сторони
    dist(x1,y1,x3,y3)+  //виклик для другої сторони
    dist(x2,y2,x3,y3);  //виклик для третьої сторони
  system("pause"); return 0;
}
```

prog010.cpp

У наведеному тексті перший рядок є ***заголовком функції***. Функція описує обчислення довжини відрізка — дійсного числа, яке ***повертається з виклику*** функції. Тип значень, що повертаються, вказано в її заголовку перед іменем функції. Після імені в круглих дужках оголошено параметри функції — імена **a1**, **b1**, **a2**, **b2**, що позначають дійсні значення. Після заголовку записано коментар, який хоча й не обов'язковий, але дуже корисний: він повідомляє, що повинна обчислювати ця функція. Далі йде ***тіло функції*** — блок, що містить послідовність інструкцій. Тут лише одна інструкція, і вона задає обчислення й повернення дійсного значення виразу — зарезервоване слово **return** означає «повернути».

У головній функції виклики функції **dist** записано у виразі — значення, що повертаються з викликів, додаються й ця сума виводиться. ◂

Оголошення параметрів функції, хоча й розташовано за межами блоку функції, діє в цьому блоці до його кінця. Параметри функції, тобто імена, оголошені в її заголовку, у літературі часто називаються ***формальними параметрами***, а вирази або імена змінних, записані у виклику, — ***фактичними параметрами***.

- Функція може не мати параметрів, але круглі дужки в її заголовку обов'язкові.

Виклик функції є виразом і може виступати операндом в інших виразах. Його типом вважається вказаний у заголовку тип значення, яке повертається.

- Будь-яке ***виконання*** функції, що повертає значення, має закінчуватися виконанням інструкції **return** з виразом, тип якого може бути