1. Парадигми програмування

Слово *парадигма* грецького походження і означає стиль міркування, спосіб дій або набір концепцій у певній галузі знань. В останні десятиліття, після того, як американський вчений Роберт Флойд під час вручення йому Тьюрінгової медалі ACM 1978 року виступив з лекцією «Парадигми програмування» [16], термін «парадигми» став часто вживаним у програмуванні, а тому варто зупинитися на ньому окремо.

1.1. Парадигми і мови програмування

Поняття програми суттєво залежить від того, на якого виконавця вона розрахована. Так, речення «знайдіть корені квадратного рівняння $x^2+2x-3=0$» є прикладом програми, виконавцем якої може бути учень або студент, причому різні виконавці можуть робити це по-різному: один – користуючись теоремою Вієта, інший – за допомогою дискримінанту. Це приклад *декларативної* (описової) парадигми: програма відповідає на запитання **«що?»** – що шукати? що будувати? що рахувати? При цьому вибір відповіді на запитання **«як?»** залишається за виконавцем.

Інша програма могла б виглядати так: «Додайте до одиниці три, добудьте із суми квадратний корінь, відніміть його від мінус одиниці. Це буде перший корінь. Потім додайте його до мінус одиниці. Це буде другий корінь». Характерну особливість цієї програми можна визначити як імперативність: **«спочатку роби одне, а потім інше»**. Вона є прикладом *імперативної* (наказової) парадигми програмування. Зазвичай декларативна парадигма потребує досконалішого виконавця, тоді як застосування імперативної парадигми покладає більше відповідальності на програміста, адже він повинен дати більш детальний рецепт розв'язування задачі. Декларативне програмування – це щось на зразок вживання скатертини-самобранки, яка сама приготує за вас обід, тоді як імперативне потребує вправності та майстерності Попелюшки у виконанні забаганок вередливої мачухи.

Поділ на **«що?»** і **«як?»** у програмуванні відображає також природний перебіг процесу розв'язування задач. Спочатку визначаємося з тим, що треба зробити, проектуємо інтерфейси – виникає щось на зразок декларативної програми. Потім з'ясовуємо наявність виконавця, здатного цю програму зрозуміти і, можливо, виконати. Якщо його підібрано, оцінюємо характеристики виконання ним програми та порівнюємо їх на відповідність заданим вимогам. Якщо придатного виконавця немає, беремо найбільш пі-

дхожого з наявних і підсилюємо його можливості додатковими імперативними засобами, необхідними для розв’язання поставленого завдання.

Цікаво, що як перша механічна програмована машина Чарльза Бебіджа в XIX сторіччі, так і сучасні електронні обчислювальні машини зобов’язані своєю появою соціальному запиту на підхожих виконавців. У позаминулому столітті це було зумовлено проблемою перерахунку таблиць тригонометричних функцій у зв’язку із введенням під час Великої французької революції метричної системи вимірювання кутів, що надихнуло Бебіджа на ідею механічних обчислень, які виконуватимуться за наперед складеною програмою. У роки другої світової війни причиною стала постійна загроза бомбардувань та обстрілів Лондона, що потребувало швидкого автоматичного виконання розрахунків траєкторій та дешифрування переговорів німецьких пілотів.

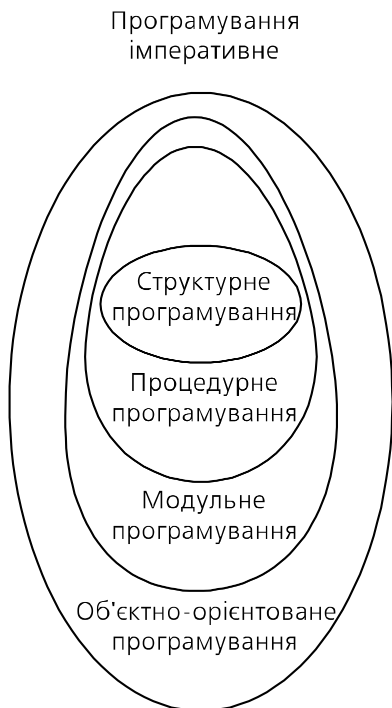

Рис. 1.1. Парадигми імперативного програмування

Всередині кожної з двох основних парадигм – імперативної і декларативної – виділяють парадигми менші. В цьому підручнику йтиметься про наказові парадигми – саме вони становлять основу сучасних виробничих технологій розробки програмних систем, – але ми не забуватимемо й про існування інших. Декларативні парадигми дають нам змогу точніше описувати властивості наказових програм. Зауважимо, що сам професор Джон Бекус – винахідник першої мови наказового програмування Фортран – успішно працював над пошуком нових парадигм, зокрема декларативної парадигми функціонального програмування, що також викладено у його Тьюрінговій лекції 1977 року [17]. Одночасне застосування кількох парадигм, або *мультипарадигменність*, дає можливість подивитися на задачу під різними кутами зору, завдяки чому поєднуються різні підходи і збільшуються шанси на відшукання кращого розв’язку.

Підкреслимо, що парадигми програмування – це моделі, які відтворюють спосіб мислення розробника програми. Мова програмування може підтримувати або не підтримувати ту чи іншу парадигму. В першому випадку застосування парадигми стає зручним, тобто простим, безпечним і ефективним, в іншому – складним і ненадійним. Ми розглянемо три основних наказових парадигми – *процедурне*, *об'єктне* (модульне) і *об'єктно-орієнтоване* (ієрархічне) програмування (рис. 1.1) – та одну допоміжну – *узагальнене* програмування.

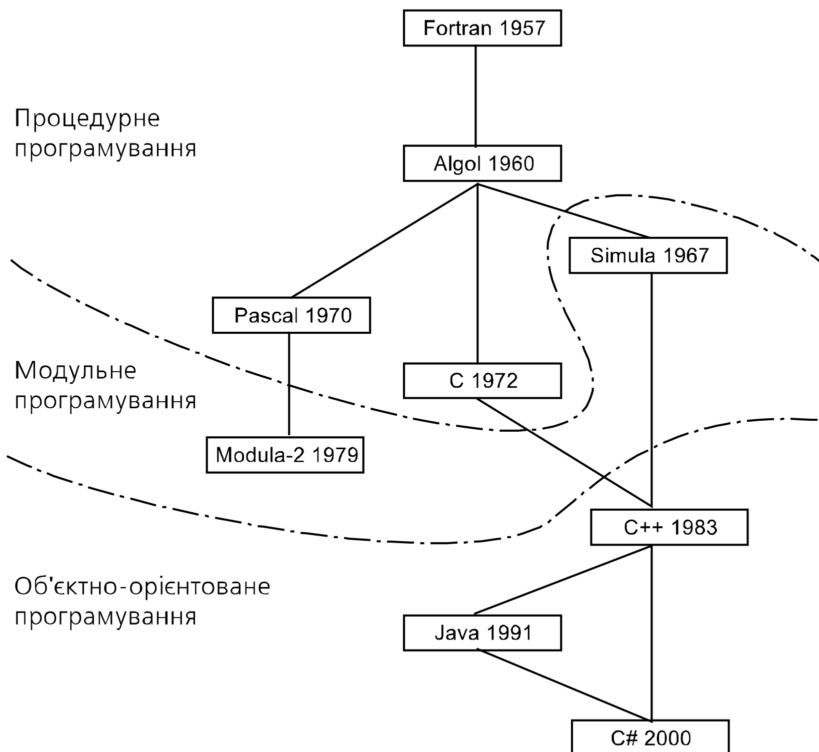

Рис. 1.2. Хронологія мов і парадигм програмування

П'ятдесят років розвитку програмування стали роками пошуків та утвердження виражальних засобів, роками зміни поколінь комп'ютерів, програмного забезпечення і мов програмування. На рис. 1.2 показано мови програмування, які зробили суттєвий внесок у розбудову сучасних парадигм. Так, мова програмування ФОРТРАН (FORmule TRANslator – перек-

ладач формул) уперше розв'язала проблему автоматизації програмування математичних формул, а її компілятори уперше впоралися із задачею роздільної компіляції.

Мова програмування АЛГОЛ (ALGOrithmic Language – алгоритмічна мова), у якій було запроваджено блочну структуру програми, стала першою мовою структурованого програмування із сучасними розгалуженнями і циклами. Майже одночасно з'явилися дві інші знакові мови програмування – Паскаль і С, які дотримувалися майже діаметрально протилежних підходів. Паскаль – прямий нащадок АЛГОЛу – доповнив його структури керування розвиненими структурами даних. Він стандартизував модель обчислювальної Паскаль-машини, призначеної для виконання програм на цій мові, й усі реальні обчислювальні середовища повинні були настроюватися на стандартну модель. Мова програмування С не передбачала стандарту обчислювача, а навпаки, містила засоби настроювання програми на конкретне обчислювальне середовище. Дослідження внеску в системне програмування, зробленого мовою С, стане одним із предметів вивчення цього підручника.

Проблема складності програмування на основі процедурної парадигми стимулювала пошук нових підходів. Так з'явилася концепція абстрактних типів даних як способу об'єднання даних і засобів їхньої обробки та одночасно як методу відокремлення специфікацій (інтерфейсів) від їх реалізації. Типи даних Паскалю були розвинуті в абстрактні типи даних у вигляді модулів мови МОДУЛА-2.

Особливо важливий крок було зроблено внаслідок залучення до мови С засобів мови моделювання СІМУЛА (SIMUlation Language – мова моделювання), в якій свого часу вперше з'явилися класи і об'єкти. З цього союзу виникла мова програмування С++, яка сама пройшла еволюційний шлях вдосконалень і розвитку. Цій темі присвячена одна з книг автора С++ Б'єрна Страуструпа [4].

Зазначимо, що хоча мова МОДУЛА-2 не набула широкого розповсюдження, закладені в ній ідеї призвели до розширення мови Паскаль класами та об'єктами. Так під назвою Object Pascal виникла об'єктна версія мови Паскаль, реалізована в системі програмування Delphi. Зрештою, назву Delphi одержала й сама мова програмування, яка продовжила давню традицію конкуренції між Паскалем і С власною конкуренцією з С++.

Головне досягнення С++ полягає у *зміні парадигми програмування* (*paradigm shift*) з процедурної на об'єктно-орієнтовану, яка визначає стандарт розробки програмного забезпечення вже упродовж більш ніж чверті століття. Головним доробком об'єктно-орієнтованої парадигми стала ієрархічність програмних структур, яка відображається в агрегації об'єк-

тів та успадкуванні класів. Продуктивність цього підходу полягає в його природності та гнучкості. Справді, програма, подана у вигляді архітектури класів та об'єктів, більш повно, ніж набір функцій і процедур, відповідає реальним об'єктам, обчислювальною моделлю яких вона є. Внаслідок цього з'явилася можливість проектувати програми більш звичними методами інженерного проектування. Водночас властивість ієрархічності дала змогу об'єднувати програмні структури у складні архітектурні конструкції, максимально використовуючи наявний код, настроювання якого тепер не потребує перепрограмування.

Повертаючись до питання про вибір мови для курсу об'єктно-орієнтованого програмування, хочу наголосити, що я дотримуюсь тези, висловленої Стівом Макконнеллом у книзі «Довершений код» [7] (раджу її для подальшого самостійного читання), про те, що програмувати варто не мовою, а з використанням мови програмування. Тому я рекомендую C++ як мультипарадигменну мову, найбільш повну з погляду концепцій сучасного програмування. Наступники C++, а саме Java і C#, поки що парадигму не змінили, зате деякі важливі риси, наявні в C++, втратили.

Саме тому тут вивчатимемо насамперед найважливіші поняття сучасного програмування і зовсім не приділятимемо (або приділятимемо дуже мало) уваги проблемам синтаксису мови, хоча всі приклади програмних текстів перевірені системою програмування Microsoft Visual Studio. Для більшості прикладів достатньо рівня MS Visual Studio 6, але для деяких, що використовуватимуть порівняно нові властивості C++, будуть потрібні новіші версії.

1.2. Процедурне програмування

Процедурне програмування зображає програму у вигляді набору алгоритмів, для оформлення яких можуть бути застосовані іменовані програмні блоки – процедури та функції. В останньому випадку передбачено наявність механізмів передавання параметрів і повернення результату.

Спочатку процедурне програмування користувалося довільними засобами керування, зокрема *переходом* за міткою (`go to`) – одним із найбільш уживаних операторів керування в Фортрані. Ось приклад програмного тексту мовою Фортран:

ЛІСТИНГ 1.1.

```
C  КВАДРАТНИЙ КОРІНЬ З ДІЙСНОГО ЧИСЛА
   ПРОЦЕДУРА-ФУНКЦІЯ НА ФОРТРАНІ
   REAL FUCTION ROOT(A, EPS)
   S=A*.5
```

```
   IF (A.EQ.0) GO TO 20
10 T=S
   S=(S+A/S)*.5
   IF (ABS((S-T)/S).GT.EPS) GO TO 10
20 ROOT=S
   END
```

В 1968 році голландський вчений Едсгер Дейкстра вперше звернув увагу на проблеми, що виникають у програмах з неконтрольованими переходами, а в 1970 році проголосив новий напрямок, який він назвав структурним програмуванням. *Структурне програмування* (не зовсім вдалий переклад англійського *structured programming* – структуроване програмування) – це варіант процедурного, що використовує лише три типи структур керування: послідовне виконання дій, розгалуження і цикл. Не дивно, що Фортран не підтримував цю парадигму: в наборі його засобів не було циклів за умовами. Починаючи з Алголу, цикли стають основним засобом організації обчислень у програмі.

ЛІСТИНГ 1.2.

```
{ Квадратний корінь з дійсного числа
  Процедура-функція на Паскалі }
function root(a: real, eps: real): real;
   var s, t: real;
begin
   s:=a*0.5;
   if a<>0 then
   repeat
      t:=s; s:=(s+a/s)*0.5;
   until abs(s-t)/s<eps;
   root:=s;
end
```

Вправа 1.1.

Встановіть відповідність між конструкціями процедур, наведених у лістингах 1.1 і 1.2.

Професор Ніклаус Вірт, автор мови програмування Паскаль, відібрав до неї лише прості в поясненні й легкі в реалізації конструкції. Внаслідок сильної типізації програми на Паскалі відзначаються високою надійністю, закладена в них концепція Паскаль-машини робить їх мобільними, їх легко читати і розуміти завдяки дисципліні програмування, яка продиктована вжитою парадигмою.

Але водночас застосування мови Паскаль гальмувалося складністю виходу за межі віртуальної машини, потребою ефективного використання наявної апаратури. Головним критерієм, застосованим Д. Річі до створеної ним мови С, стала саме гнучкість використання особливостей конкретної апаратури та ефективність виконання програм.

1.3. Об’єктне (модульне) програмування

Процедурна парадигма віддала належне алгоритмічній компоненті програмування. Але зі зростанням обсягу програм і складності даних з’явилася нова проблема, а саме проблема структурної організації даних, найбільш влучно висловлена Віртівською формулою: «*алгоритми + структури даних = програми*» [18].

Поняття *модуля* як абстракції даних було вперше запропоноване Девідом Парнасом у 1972 році [20]; правда, на той час уже існувала мова програмування Симула–67, в якій була використана парадигма модулів у вигляді об’єктів. Найбільш повно поняття абстракції даних було реалізоване в мові програмування Модула–2 [19].

Головна ідея модульності даних полягає в забезпеченні доступу до них і оперування ними незалежно від способу їхнього конкретного кодування у пам’яті комп’ютера. Самі дані разом із процедурами їх обробки вбудовують (*інкапсулюють*) в окрему одиницю програми. Ось простий приклад, який демонструє, як складність обчислень може перетікати у складність даних.

Обчислення математичної функції, скажімо, квадратного кореня, можна виконати, обчислюючи границю відповідної послідовності, наприклад, такої:

$$x_0 = \frac{a}{2};\ x_n = (x_{n-1} + \frac{a}{x_{n-1}});\ n = 1,2,...;\ \lim_{n\to\infty} x_n = \sqrt{a}.$$

Цей же квадратний корінь можна знайти у відповідному рядку чотиризначних математичних таблиць Брадіса. У першому випадку вся складність зосереджена в алгоритмі, який необхідно придумати і запрограмувати (що було зроблено у наведених вище прикладах програм на Фортрані та Паскалі). У другому – в структурах даних, тобто в таблицях, які професор В.М. Брадіс повинен був попередньо розрахувати. Тут алгоритм обчислення зводиться до вибору потрібної клітинки таблиці.

Модулі мають дві головні риси. По-перше, вони об’єднують структури даних з алгоритмами їхньої обробки. По-друге, у них відокремлено специфікацію від реалізації інкапсульованих у модулі конструкцій, і це

перетворює модуль на так званий *абстрактний тип даних (abstract data type)*, на що свого часу звернув увагу Джон Гуттаг [21].

Ознайомитися з абстрагуванням даних можна на прикладі тексту мовою Модула–2, який дає певне уявлення про поділ модуля на дві частини – визначення і реалізацію:

ЛІСТИНГ 1.3.

```
{ Визначення модуля обчислення кількості днів
  у місяці мовою Модула-2 }
definition module Months;
   type Month=
      (jan, feb, mch, apr, may, jun, jul, aug, sep, oct, nov, dec);
   procedure length(m: Month): Cardinal;
end Month;
```

Наведеного у лістингу 1.3 визначення модуля `Months` достатньо для його використання. З тексту бачимо, що модуль містить тип даних `Month` для позначення місяців року і процедуру `length`, яка обчислює кількість днів у місяці, заданому як параметр. Результат цієї процедури є натуральним (кардинальним) числом, тип якого позначено як `Cardinal`. Щось подібне можна було б запрограмувати й на Паскалі, проте зв’язок між процедурою і типом даних, інкапсульований у цьому модулі, втратився б.

Окремо програмують реалізацію модуля (лістинг 1.4). Вона складається із визначення структур даних (у нашому випадку це масив кількості днів у місяцях), реалізації процедури та ініціалізації модуля (вона виконує підготовчу роботу з наповнення структур даних).

ЛІСТИНГ 1.4.

```
{ Реалізація модуля }
implementation module Month;
var
   len: array Month of Cardinal;
   procedure length(m: Month): Cardinal;
   begin
      return len[m];
   end length;
{ Ініціалізація модуля }
begin
   len[jan]:=31;   len[feb]:=28;   len[mch]:=31;   len[apr]:=30;
   len[may]:=31;   len[jun]:=30;   len[jul]:=31;   len[aug]:=31;
   len[sep]:=30;   len[oct]:=31;   len[nov]:=30;   len[dec]:=31;
end Month.
```

Як побачимо далі, поділ програмних конструкцій на визначення інтерфейсу та його реалізацію став типовою рисою сучасних мов програмування.

1.4. Об'єктно-орієнтовне програмування

Як було зазначено раніше, об'єктно-орієнтована парадигма розвиває об'єктне (модульне) програмування засобами створення ієрархій об'єктів і класів. Об'єктно-орієнтоване програмування, за метафорою Б'єрна Страуструпа, – це «*високоінтелектуальний синонім доброго програмування*». Справді, хоча нові парадигми програмування з'являються не так часто – приблизно одна на десятиліття, лише деякі з них, як, наприклад, структурне програмування, стають справжніми довгожителями. Той факт, що об'єктно-орієнтовану парадигму успішно використовують упродовж більш ніж чверті сторіччя, сам собою є вагомим підтвердженням її життєздатності.

Справді, алгоритми, реалізовані у процедурному програмуванні, надто конкретні. Будь-яка модифікація – це вже новий алгоритм, і тому кількість використовуваних процедур і функцій, як і затрати на їхнє розроблення або перепрограмування, надмірно зростають. Модульне програмування групує алгоритми в модулі (класи), інкапсулюючи разом з алгоритмами і відповідні структури даних. Тепер залишається зробити наступний крок – побудувати ієрархічну структуру.

Для чого потрібні ієрархії? Чому саме ієрархії виявилися головною рисою новітніх технологій програмування? По-перше, ієрархічні структури дають змогу керувати складністю програмних проектів, відділяючи внутрішню складність конструкції від її зовнішнього використання. По-друге, ієрархічні структури забезпечують ефективні будівельні блоки для конструювання програм, надаючи ємніші програмні конструкції, ніж окремі процедури або функції. І, нарешті, ієрархічні структури забезпечують можливість пристосування до нових умов вже наявних програмних кодів, не втручаючись у їхню внутрішню будову.

Ієрархії можуть бути двох типів. Перший – це бути частиною чогось. Наприклад, грань є частиною многогранника, ребро – грані, вершина – ребра. Цей тип ієрархії дає змогу збирати об'єкти з частин, які самі є об'єктами. Інший, складніший тип дає можливість будувати узагальнення або, навпаки, конкретизації. Наприклад, овал і многокутник є конкретизацією плоскої фігури, коло – овалу, чотирикутник – многокутника; подальшими конкретизаціями чотирикутника можуть бути паралелограм, прямокутник, ромб, квадрат. Те, що квадрат, ромб чи прямокутник є повноцінними паралелограмами, дає їм змогу користуватись усіма програмними засобами, створеними для паралелограмів; паралелограм зі свого боку є повноцінним чотирикутником і так далі. Цей принцип, відомий під

назвою *reusable – знову (або навіть заново) вживаний*, – став одним із найважливіших досягнень об'єктно-орієнтованої парадигми. Щоб знову використати вже наявне програмне забезпечення у більш конкретизованих чи навіть зовсім несподіваних умовах, достатньо дописати лише ту його частину, яка стосується особливостей відповідної конкретизації. Цей принцип отримав назву *programming by difference* або *дописування програм*.

І, нарешті, об'єктно-орієнтована парадигма доводить до логічної завершеності принцип моделювання реального світу, а точніше – тієї його частини, абстракцією якої є програма. За такого підходу програма складається з об'єктів, які відповідають реальним поняттям або предметам, а її виконання зводиться до взаємодії цих об'єктів, що є абстракцією реальної взаємодії їхніх прототипів. Все це разом забезпечує об'єктно-орієнтованому підходу беззаперечне лідерство в галузі розробки програм.

Сьогодні в сімействі мов об'єктно-орієнтованого програмування три найбільш відомі представники: C++, Java і C# (читають як «Сі шарп»). C++ дотепер залишається визнаним лідером у розробленні великих і складних програмних систем. Java і C# виросли з C++, вони мають свою сферу застосування в розподіленому програмуванні.

Ще одна перевага C++ – її мультипарадигменність. C++ містить у собі мову C (з деякими застереженнями), а тому, природно, підтримує процедурну парадигму. Ось приклад функції для обчислення квадратного кореня в C/C++, після якої для порівняння розмістимо аналогічну функцію на Паскалі.

ЛІСТИНГ 1.5.

```
/* Квадратний корінь з дійсного числа
   Функція мовою C   */
double root(double x, double eps) {
   double s=0.5*x;
   double t=s;
   if (x!=0)
      do {
         t=s; s=(s+x/s)*0.5;
      } while ((fabs(s-t)/s)>eps);
   return s;
}
```

ЛІСТИНГ 1.6.

```
{ Квадратний корінь з дійсного числа
  Процедура-функція на Паскалі }
function root(a: real,eps: real): real;
```

```
   var s, t: real;
begin
   s:=a*0.5;
   if a<>0 then
      repeat
         begin
            t:=s; s:=(s+a/s)*O.5;
         end
      until abs(s-t)/s<eps;
   root:=s;
end
```

Обидві функції не відрізняються одна від одної з точністю до позначень для структур керування. Несуттєві відмінності полягають у способі запису заголовка, визначеннях та ініціалізації змінних, позначеннях для розгалуження і циклу (зокрема є відмінність семантики циклу з умовою закінчення), позначеннях операторних дужок і способі повернення результату.

С++ також підтримує традиційні для модульного програмування механізми абстракції даних, доповнені можливостями об'єктно-орієнтованої парадигми. Ось якого вигляду на С++ набуде клас місяців року, відповідний до прикладу з лістингів 1.3-1.4 мовою Модула–2:

ЛІСТИНГ 1.7.

```
// Клас С++ для обчислення кількості днів у місяці
// Специфікація класу
class Month {
public:
   enum month
      {jan=1, feb, mch, apr, may, jun, jul, aug, sep, oct, nov, dec};
   int length();
   Month(month n): _mon(n) {}
private:
   month _mon;
   static int _len[12];
};
// Реалізація класу
int Month::length() {
   return len[mon-1];
}
int Month::len[12] {31,28,31,30,31,30,31,31,30,31,30,31};
```

Відповідність текстів обома мовами у цьому випадку стала не такою очевидною, як для процедурного варіанту. Більше того, відмінність мо-

дульного і об'єктного підходів стає помітною вже на цьому простому прикладі.

На відміну від модуля, клас `Month` – це повноцінний тип даних, який можна використовувати, нарівні зі стандартними, для визначення змінних, наприклад:

ЛІСТИНГ 1.8.

```
Month theMonth=mai;
int i=theMonth.length();
```

Як побачимо далі, на новостворений тип можна навіть поширити набір стандартних операції, наприклад, визначивши операцію збільшення місяця на ціле число, що дасть змогу писати вирази на кшталт `theMonth+5`.

І, нарешті, крім процедурної, об'єктної та об'єктно-орієнтованої парадигм, C++ реалізує також для кожної з них парадигму узагальненого програмування, надаючи можливість використання узагальнених функцій і параметризованих класів, але про це далі.

1.5. Програмні середовища

Від складання програмістом до виконання комп'ютером програма проходить доволі тривалий процес оброблення спеціальними службовими програмами, які становлять систему автоматизації програмування. З часом слово «автоматизація» випало із наведеного словосполучення, внаслідок чого воно перетворилося на «систему програмування». Ця система складається з кількох компонент, а саме *препроцесора (preprocessor), компілятора (compiler)* та *компонувальника (linker)*, а також засобів підтримки етапу виконання, зокрема *налагоджувача (debugger)*, об'єднаних спільним інтерфейсом у так зване універсальне середовище розробки програм. Прикладами таких середовищ можуть бути система програмування Visual C++ та діалогове середовище розробки програм Developer Studio.

Головна особливість підготовки програми до виконання полягає в тому, що програму збирають із багатьох, часом різнорідних, складових частин, об'єднаних у *програмний проект (software project)*. Кожну програмну розробку оформлюють як окремий проект, який зазвичай складається з багатьох різнорідних файлів. Файли першого типу називають *вхідними (source file, исходный файл*), вони містять тексти, написані певною мовою програмування. Робота над проектом може бути достатньо тривалою, в її процесі виникатимуть нові вхідні файли, які додаватимуться до наявних. Тому в кожен момент часу частина вхідних файлів може виявитися вже готовою до використання, тобто попередньо відкомпільованою. Цю частину вико-

ристовують у вигляді готових машинних кодів, що зберігаються в проекті як особливі *об'єктні файли (object file)*, котрі складають другий тип файлів програмного проекту. Система програмування автоматично відстежує необхідність повторного компілювання кожного вхідного файлу. Компіляції підлягає кожен новий файл, долучений до проекту, а також будь-який інший файл після внесення до нього змін.

Крім спеціально розроблених власних кодів, проекти можуть використовувати стандартне програмне забезпечення, що зберігається в системних бібліотеках, передбачена також можливість створення і подальшого використання власних бібліотек. С++ послідовно продовжує закладену ще авторами мови С тенденцію широкого використання програмних бібліотек. Зокрема, мовою не визначено ніяких способів зв'язку програми з операційною системою – ці функції повністю перекладено на системні бібліотеки.

Вже упродовж не одного десятиріччя під час складання великих за розмірами програм використовують принцип структурної декомпозиції. Логічним складовим частинам програми (у процедурному програмуванні це процедури і функції, у більш розвинених парадигмах – модулі або класи) відповідають фізичні складові – файли, що містять ці окремі завершені логічні частини, одну або декілька разом. Одночасне використання в одному програмному проекті багатьох файлів з текстами різних частин програми одержало назву *роздільної компіляції (separate compilation)*, яка, без сумніву, стала одним із найбільших досягнень систем програмування. Завдяки їй програму тепер поділено на частини, які називаються *одиницями трансляції* (*translation unit*). З використанням роздільної компіляції стало можливим колективне розроблення великих програм: окремі розробники зайняті підготовкою кожен своїх файлів.

Програмування – це діяльність, яка потребує великої організованості. Прийнято не тільки розділяти текст програми на структурні частини, але й розрізняти описові та виконавчі частини текстів. Тому під час складання програм дотримуються певних правил доброго тону, одне з яких полягає в необхідності розподілення визначень і обчислень між вхідними файлами двох типів: *файлами заголовків (header) і файлами реалізації (implementation)*. Особливий компонент системи програмування – *препроцесор* – у відповідності з директивами, розміщеними, як правило, на початку вхідного файлу, приєднує до нього відповідні файли заголовків. Вони містять інформацію про те, які функції, об'єкти або класи, визначені поза цим вхідним файлом, можуть бути використані в ньому. Один файл заголовків можна приєднувати до багатьох вхідних файлів, які ви-

користовуватимуть оголошені в ньому конструкції. Приєднанням потрібних заголовних файлів препроцесор готує вхідні файли до подальшої обробки компілятором. Тому вміст вхідного файлу для компілятора, взагалі кажучи, відрізняється від вмісту вхідного файлу, складеного розробником, оскільки обробка препроцесором може значно його змінити. Далі розглянемо, в яких межах варто зосередити цю обробку.

Набір пов'язаних один з одним заголовних і вхідних файлів, з яких складається проект, в сукупності утворює *вхідну програму*, яку система програмування перетворюватиме на машинний код. Машинний код генерується у два етапи. На першому компілятор генерує так звані *об'єктні коди (object code)*, по одному для кожного файлу реалізації (за заголовними файлами об'єктні не генеруються). Коди називають об'єктними (від слова «objective», що англійською мовою означає «мета»), оскільки їхнє створення є метою роботи компілятора, який становить центральну частину системи програмування. Інша її частина, названа *компонувальником (linker)*, збирає об'єктні файли в одну *виконавчу програму (executable program*, *exe-file)*, приєднуючи до неї також об'єктні коди зі стандартних або власних бібліотек проекту.

Ми розглядатимемо систему програмування такою мірою, наскільки вона впливає на написання та організацію текстів вхідних програм. За детальним ознайомленням із засобами і можливостями систем програмування відсилаємо до системної документації та спеціальної літератури.

Ось типовий сценарій підготовки програми.

Система програмування працює з *програмними проектами*, де розміщують файли вхідної програми. Розглянемо, наприклад, проект **sqrt**. До файлу реалізації **root.cpp** запишемо текст функції `root` для обчислення квадратного кореня, а програмний код для її виклику помістимо у файл реалізації **main.cpp**.

Пам'ятаймо, що компіляція цих двох файлів виконуватиметься нарізно. Виникає запитання: звідки під час компіляції файлу **main.cpp** компілятор візьме відомості про ідентифікатор `root`? У принципі можливі два вирішення. Перше – це перемістити текст функції з файлу **root.cpp** до файлу **main.cpp**, де функцію використано. Проте це було б не дуже вдалим виходом, оскільки незрозуміло, як одну й ту ж функцію розмістити у різних вхідних файлах, де її також можна було б викликати. Друге вирішення, а саме воно прийняте в системах програмування, полягає у створенні допоміжного заголовного файлу **root.h**, який міститиме не всю інформацію про функцію `root`, а лише ту, що необхідна для обробки її виклику компілятором, – так звану *сигнатуру функції*:

```
// root.h
double root(double x, double eps);
```

Проект **sqrt**, наповнений вхідними файлами, зображено на рис. 1.3.

Заголовний файл приєднують до вхідного за допомогою команд препроцесора. Їх записують за допомогою спеціальних директив, які починаються символом **#**. Зокрема, файл **main.cpp** міститиме директиву `#include "root.h"`, виконання якої препроцесором полягає у включенні тексту з файлу заголовку **root.h** до вхідного файлу **main.cpp**. Наведемо текст, доповнений коментарями, покликаними зробити його зрозумілим навіть для читача, який ще зовсім не знає C.

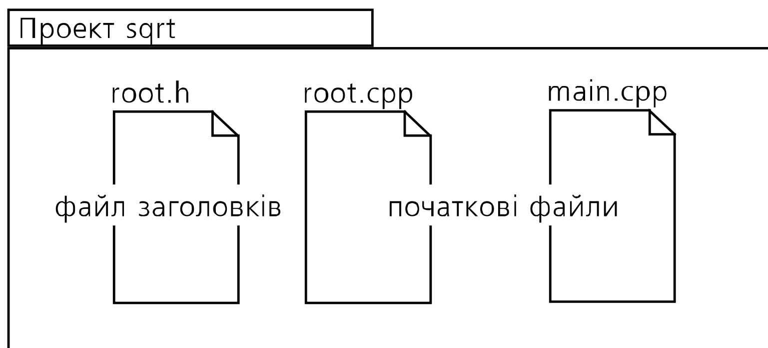

Рис. 1.3. Склад вхідних файлів проекту sqrt

ЛІСТИНГ 1.9.

```
// main.cpp
// Спочатку системні заголовки
// Підключення бібліотеки введення та виведення
#include <iostream>
// Підключення бібліотеки математичних функцій
#include <cmath>
// Посилання на стандартні позначення
using namespace std;
// Тепер власні заголовки
#include "root.h"
// Початок власне програми, вона називається main
int main() {
// cout<<someting; вивести something
// cout<<someting<<endl; вивести те ж саме і перейти на новий рядок
   cout<<"The square root of 2 calculated with"<<
      endl<<" a standart fuction is"<<sqrt(2.0)<<endl;
```

```
   cout<<" your function is "<<root(2.0,0.000001)<<endl;
   return 0;
}
```

Як бачимо, крім заголовного файлу **root.h**, вхідний файл використовує ще кілька інших, службових файлів, приєднаних препроцесором (їх не показано на рис. 1.3). Файли **iostream** і **cmath** описують стандартні бібліотеки: введення-виведення **iostream** (вона потрібна для обслуговування вихідного потоку `cout`) та математичних функцій **cmath**, звідки буде взято, наприклад, функцію обчислення квадратного кореня `sqrt`. Директива `using namespace std` є вказівкою компілятору вживати стандартні імена для системних об'єктів: наприклад, через `cout` позначено стандартний вихідний потік.

Відповідного доповнення потребуватиме також файл **root.cpp**, у якому використано стандартну функцію `fabs` обчислення абсолютної величини дійсного числа:

ЛІСТИНГ 1.10.

```
// root.cpp
#include <cmath>
double root (double x, double eps) {
   double s=0.5*x;
   double t;
   do {
      t=s; s=(s+x/s)*0.5;
   } while ((fabs(s-t)/s)>eps);
   return s;
}
```

Тепер коротко про призначення інших компонентів системи програмування. Після того, як буде проінтепретовано команди препроцесора, файли повних текстів надійдуть на вхід компілятора, який створить із них об'єктні коди (рис. 1.4). Об'єктний код ще не призначено для безпосереднього виконання. У проекті таких кодів багато, взагалі кажучи, по одному на кожен вхідний файл, і вони містять взаємні посилання один на одного, а також на бібліотеки. Об'єктні коди надходять на вхід компонувальника, завдання якого – зібрати їх разом, доповнити необхідними компонентами бібліотек та перетворити на виконавчу програму.

На вимогу розробника компілятор може доповнити об'єктні коди спеціальними командами спостереження за ходом виконання програми. Тоді самим виконанням програми займеться *налагоджувач (debugger)*, який дасть змогу простежити за перебігом цього процесу, призупинити його, проконтролювати значення змінних тощо.

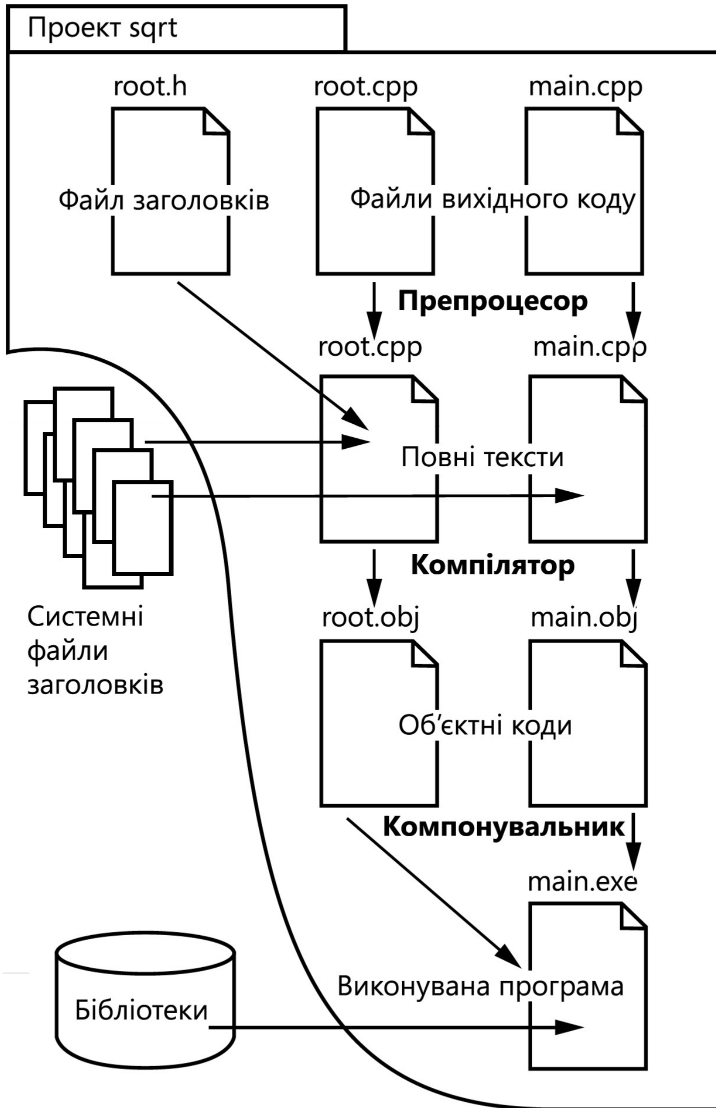

Рис. 1.4. Повний склад файлів проекту sqrt

Завдання до розділу 1

1. Інсталюйте систему програмування C++ (MicrosoftVisualStudio, GCC або іншу). Керуючись підказками системи, створіть консольний порожній проект **sqrt**.
2. Створіть у проекті файл **root.cpp**. Приєднайте до нього системний заголовний файл **cmath**. Розмістіть у ньому текст функції `root` та виконайте його компіляцію.
3. Створіть у проекті файл **root.h**. Розмістіть в ньому заголовок функції **root**.
4. Створіть у проекті файл головної функції **main.cpp**. Приєднайте до нього власний заголовний файл **root.h** та системні файли **iostream** і **cmath**. Включіть директиву `using namespace std`. Запишіть текст функції `main`. Виконайте компіляцію.
5. Користуючись документацією системи програмування, зберіть і виконайте проект в автоматичному режимі.
6. Модифікуйте головну функцію, визначивши в ній змінну `x` та вказавши оператори введення підкореневого значення:

   ```
   double x(2.0);
   cout<<"give your value for x ";
   cin<<x;
   ```

   та відповідно замініть аргументи функцій `sqrt` та `root` на `x`. Виконайте модифіковану в цей спосіб програму.
7. Користуючись налагоджувачем, виконайте функцію `main` в покроковому режимі, спостерігаючи за ходом обчислень.
8. Увійдіть в процесі виконання до функції `root` та простежте за зміненням значень змінних.