004.43(075.8)
В19

ОЛЕКСІЙ ВАСИЛЬЄВ

ПРОГРАМУВАННЯ на C++

В ПРИКЛАДАХ І ЗАДАЧАХ



Ліра-К

Олексій Васильєв

Програмування на C++

в прикладах і задачах

Навчальний посібник

Київ
Видавництво Ліра-К
2017

УДК 004
В 191

Рецензенти:
Решетняк Віктор Юрійович – доктор фізико-математичних наук, професор кафедри теоретичної фізики фізичного факультету Київського національного університету імені Тараса Шевченка
Єщенко Олег Анатолійович – доктор фізико-математичних наук, професор кафедри експериментальної фізики фізичного факультету Київського національного університету імені Тараса Шевченка

Рекомендовано до друку кафедрою теоретичної фізики фізичного факультету Київського національного університету імені Тараса Шевченка протокол №23 від 9 червня 2017 р.

Васильєв О.
В 191 Програмування на С++ в прикладах і задачах : Навч. посіб. / О. Васильєв. – Київ : Видавництво Ліра-К, 2017. – 382 с.

ISBN 978-617-7507-41-2

Книга присвячена мові програмування С++. Це мова професійного програмування, яка знаходить широке застосування при розв'язанні самих різноманітних задач, від фінансових розрахунків до моделювання складних систем і космічних розробок. Для більшості програмістів шлях до професії починається саме з вивчення мови С++.

Книга складається із тематично підібраних прикладів і задач, котрі перекривають всі основні розділи та питання, важливі при вивченні мови програмування С++. Матеріал подається в простій формі, адаптованій для вивчення мови програмування в найкоротші терміни. Представлена велика кількість оригінальних програм, які в повній мірі розкривають красу та потужність мови С++.

Книга буде корисною для студентів, викладачів, інженерів, науковців та всіх, хто вивчає мову програмування С++ самостійно.

ISBN 978-617-7507-41-2

© Васильєв О., 2017
© Видавництво Ліра-К, 2017

Вступ

Про книгу та мову C++

- Яка ж гульня гарна!

- Горілки все одно пити не будемо...

з х/ф "Пропала грамота"

На розгляд читача пропонуємо книгу про мову програмування C++. Книга для тих, хто не знайомий (або мало знайомий) із мовою C++ і хоче швидко навчитися програмувати нею.

Власне про книгу

Книга складається з тематично підібраних прикладів і задач, які перекривають усі основні розділи і питання, важливі з огляду вивчення основ мови C++. Тут хочеться відзначити два важливих моменти.

По-перше, досвід показує, що найкраще матеріал засвоюється, якщо він ілюстрований прикладами. Окрім того, іноді дуже важко пояснити певну концепцію чи підхід, якщо їх не підкріпити практичним матеріалом. У цьому сенсі обраний для книги спосіб подання інформації дає надію на скромний оптимізм.

По-друге, сподіваємося, що під час роботи з матеріалом книги читач виявить певну наполегливість і готовність до поглибленого аналізу програмних кодів. З іншого боку, автор доклав максимальних зусиль, щоб зробити вміст книги простим і наочним навіть для непідготовленого читача.

У певному сенсі особливою виступає остання глава, де зібрані математичні задачі обчислювального характеру. Вони орієнтовані на більшу самостійність читача в плані аналізу програмних кодів (порівняно з прикладами в інших главах книги). Цю главу включено до книги, оскільки подані в ній задачі

розглядаються в рамках університетського лекційного курсу з обчислювальної математики.

Для кращого закріплення матеріалу наприкінці перших чотирьох глав наведено список задач для самостійного розв'язування. Вони є концептуально близькими до тих задач і прикладів, які розглядалися в основній частині відповідної глави. Деякі задачі дуже прості, проте є й такі, для розв'язання яких потрібно виявити кмітливість. Також варто зазначити, що для певних задач для самостійного розв'язування в книзі (зазвичай у наступних главах) наводяться розв'язки. Можливо, контекст задачі під час розв'язування дещо змінюється, та впізнати знайомі "ноти" все ж можливо. Це зроблено спеціально, оскільки, як показує досвід, розв'язати одну й ту саму задачу різними методами іноді буває набагато корисніше, ніж просто розв'язати різні задачі.

У п'ятій і шостій главах, де обговорюються принципи об'єктно-орієнтованого програмування, список задач для самостійного розв'язування також є, але він набагато менший. Запропоновані там задачі "евристичного" характеру і сформульовані в загальному вигляді. Це обдуманий хід, оскільки в даному випадку важливою складовою є вибір контексту постановки задачі. Іншими словами, частиною задачі, яку необхідно розв'язати читачу, є, в тому числі, і її кінцеве формулювання.

У главах із п'ятої по восьму виокремлюються деякі "ідеологічні" напрямки на той випадок, якщо читач вирішить самостійно вивчати відповідні аспекти програмування (рекомендації для самостійної роботи). Задачі для самостійного розв'язування в сьомій і восьмій главах не пропонуються, оскільки матеріал там складний і деякі теми наведені швидше в режимі ознайомлення.

Нарешті, в останній главі, де розглядаються математичні обчислювальні задачі, немає ані задач для самостійного розв'язування, ані рекомендацій для самостійної роботи. Причина в тому, що глава орієнтована на самостійне вивчення нетривіального матеріалу, тому перевантажувати небажано.

Мова програмування С++

Мова програмування С++ є однією з найпопулярніших мов програмування. Неможливо уявити собі професійного програміста, який би не знав мови С++. У цьому сенсі вибір мови С++ для вивчення здається досить вдалим.

Окрім безпосередньої практичної користі від уміння складати програми на мові С++, є ще й неабиякий методологічний аспект. Пов'язаний він із винятковою гнучкістю й багатством мови С++. Після її вивчення набагато легше "брати на озброєння" інші мови програмування. Проте, хоч би там як, мова С++ на сьогодні затребувана й найближчим часом навряд чи це зміниться.

Середовище розробки

Програмного забезпечення для роботи з програмними кодами на С++ не бракує. На даний момент існує багато якісних засобів розробки - як комерційних, так і безкоштовних. Зазвичай використовують *середовища розробки*, до складу яких входить редактор кодів, налагоджувач, звичайно компілятор і ряд інших допоміжних утиліт. Читач, безумовно, вільний у виборі того середовища розробки, яке йому найбільш симпатичне. Коди, що розглядаються в основній частині книги, універсальні й не прив'язані до якогось конкретного середовища. Разом з тим важливо зазначити, що тестувалися програми у Visual Studio Express 2013. Хто бажає, може завантажити (безкоштовно, але з реєстрацією) це середовище розробки з сайту компанії Microsoft.

Про автора

Автор книги - *Васильєв Олексій Миколайович*, доктор фізико-математичних наук, професор кафедри теоретичної фізики фізичного факультету Київського національного університету імені Тараса Шевченка. Автор книг із програмування й математичного моделювання. Сфера наукових інтересів:

фізика рідин, біофізика, синергетика, математичні методи в економіці, моделювання соціально-політичних процесів, математична лінгвістика.

Зворотний зв'язок

Свої зауваження й пропозиції щодо книги можна висловлювати на електронну пошту alex@vasilev.kiev.ua. На сайті автора www.vasilev.kiev.ua можна знайти інші контактні дані, додаткову інформацію про автора, а також деякі корисні матеріали щодо цієї та раніше виданих книг.

Подяка

Автор висловлює щиру вдячність своїм читачам за інтерес до книг. Розуміння того, що книги затребувані читачами, є найкращим стимулом для їх написання.

Також найщиріші слова подяки *Ілоні Васильєвій*, яка приймала безпосередню участь у підготовці рукопису і всіляко сприяла роботі над книгою.

До уваги

В тексті книги в кодах програм деякі занадто довгі команди відображаються в декількох рядках. В більшості випадків (якщо розрив рядка виконаний не в межах текстового літералу) це не є проблемою. Архів з вихідними кодами програм можна завантажити за адресою

www.vasilev.kiev.ua/data/cpp_ukr_examples.zip.

Глава 1

Прості програми

- Чи ти козак, чи волох, чи що ти таке?

- Я чорт. Тільки добрий чорт.

з х/ф "Пропала грамота"

У цій главі розглянемо найпростіші програми. Окрім того, ми дізнаємося:

- як організована програма в C++;
- як здійснити введення й виведення даних через консольне вікно;
- як оголосити змінні;
- як виконати прості арифметичні операції,

ну й дещо інше. Розпочнемо з розгляду задач.

Перша програма

Розглянемо невелику програму, яка відображає привітання в консольному вікні. Відповідний програмний код наведено в лістингу 1.1.

Лістинг 1.1. Перша програма

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Відображення повідомлення в консолі:
   cout<<"Програмуємо на C++!"<<endl;
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
```

Після виконання програми отримаємо такий результат:

Результат виконання програми (з лістингу 1.1)

```
Програмуємо на C++!
```

Відповідне повідомлення з'являється в консольному вікні.

Розглянемо і проаналізуємо програмний код. Насамперед, виокремимо основні блоки в програмі. Фактично, він там один – це тіло *головної функції* програми. Головна функція програми має назву `main()` й описана за таким шаблоном:

```
int main(){
   // Код головної функції програми
}
```

Власне тіло функції виділено за допомогою пари фігурних дужок: тієї, що відкриває блок `{` та тієї, що закриває блок `}`. Те, що між цими дужками, - код функції `main()`. Ключове слово `int` в описі функції означає, що функція повертає результат, і результат функції є цілим числом.

Виконання програми в C++ - це виконання головної функції `main()`. Іншими словами, щоб зрозуміти, які команди виконуються в програмі, варто звернутися до функції `main()`. У нашому випадку в тілі функції декілька (точніше, сім) рядків коду, причому три з них - *коментарі*.

Коментарі починаються з подвійної скісної риски і під час компіляції програми ігноруються. Призначення коментаря - пояснити тому, хто аналізує програмний код, роль тих чи інших команд. Коментарі потрібні людині, а не комп'ютеру. Наявність коментарів у програмному коді необов'язкова, проте вони істотно полегшують роботу програміста.

Теорія

У C++ використовують однорядкові й багаторядкові коментарі. Однорядковий коментар починають з подвійної скісної риски `//`. Усе, що праворуч від неї - коментар. Однорядковий коментар розташовують в одному рядку.

Багаторядковий коментар, як неважко здогадатися, містить декілька рядків. Починають такий коментар комбінацією символів `/*`, а закінчують символами `*/`. Усе, що знаходиться між інструкціями `/*` та `*/`, є коментарем.

Дві команди в тілі функції `main()` мають технічний, "допоміжний" характер. Їхня наявність у програмному коді пов'язана з особливостями функціонування консольного вікна й необхідністю відображати в консольному вікні кириличний текст. Йдеться про команди `system("chcp 1251>nul")` та `system("pause>nul")`. Перша з них переводить консоль у режим використання кодування *Windows-1251*, а друга потрібна для того, щоб після виконання програми консольне вікно не закривалося відразу, а затримувалося на екрані до натискання користувачем клавіші <Enter> (або будь-якої іншої).

Подробиці



Працюючи з операційною системою Windows і введенням/виведенням інформації через консольне вікно, користувач, швидше за все, зустріне дві проблеми. Перша пов'язана з відображенням кириличного тексту в консольному вікні.

Вона полягає в тому, що редактор кодів, у якому набирають код програми, має кодування, відмінне від кодування консольного вікна. Тому кириличний текст, набраний у вікні редактора кодів, буде нормально виглядати в ньому, але зовсім некоректно – у консолі. Та сама проблема із введенням тексту: якщо через консоль вводиться кириличний текст, то програмою він буде зчитаний як набір дивних символів.

Вирішити проблему можна, якщо встановити кодування вікна консолі таке саме, як і кодування вікна редактора кодів. Зробити це можна "вручну", або безпосередньо через програмний код. Для цього на початку програмного коду розташовуємо команду `system("chcp 1251>nul")`.

Ще одна проблема пов'язана з тим, що після закінчення виконання програми консольне вікно, до якого здійснюється виведення, автоматично закривається. Відбувається все дуже швидко - настільки, що користувач не встигає помітити, яка інформація виводилася в консольне вікно під час виконання програми. Щоб консоль не закривалася після закінчення виконання програми, додаємо до програмного коду команду

system("pause>nul").

Команди system("chcp 1251>nul") і system("pause>nul") є викликом функції system(). Щоб функція system() була доступна, підключаємо заголовок <cstdlib>.

Аргументом функції system() передається текстовий рядок, що містить команду, яку потрібно виконати в консолі. Зокрема, команда pause для консолі означає затримку консольного вікна, а команду chcp 1251 використовують для застосування відповідного кодування. В принципі, у програмному коді можна використовувати команди system("chcp 1251") і system("pause"), однак, у цьому випадку до консольного вікна виводяться повідомлення відповідно про встановлення кодування і про необхідність натиснути кнопку для закриття консольного вікна. Щоб уникнути цього, використовуємо інструкцію nul, вказану разом з символом > після власне команди, яку необхідно виконати в консолі.

Мабуть, найважливіша команда в тілі функції main() - це cout<<"Програмуємо на C++!"<<endl. Саме вона "виконує" основне завдання, відображаючи повідомлення в консольному вікні, заради чого, власне, і створювалася програма. Основу команди складає *оператор виведення* <<. Дані, вказані праворуч, оператор виведення відображає в пристрої, посилання на який вказано ліворуч від оператора. Наприклад, команда cout<<"Програмуємо на C++!" означає, що текст "Програмуємо на C++!" необхідно вивести у пристрій, "схований" за ідентифікатором cout. За замовчуванням ідентифікатор cout означає консоль (*cout* від *console output*). Текст, як не важко здогадатися, беруть у подвійні лапки. В одній команді можна використовувати декілька операторів виведення. Інструкцію endl (від *end of line*) використовують, щоб перевести курсор до нового рядка у вікні виведення (консолі). Отже, уся команда cout<<"Програмуємо на C++!"<<endl означає таке: у консольне вікно необхідно вивести текст "Програмуємо на C++!", після чого виконати перехід до нового рядка.

До уваги

У принципі, необов'язково використовувати інструкцію `endl` для переходу до нового рядка. Замість команди `cout<<"Програмуємо на C++!"<<endl` цілком підійде і команда `cout<<"Програмуємо на C++!"`.

Остання команда `return 0` в тілі функції `main()` закінчує її виконання та повертає як результат функції значення `0`. Значення `0` сповіщає операційну систему, що роботу функції закінчено в нормальному режимі, без помилки. Зазвичай команда `return 0` є останньою у функції `main()`.

До уваги

Також слід зауважити, що команди в C++ закінчують крапкою з комою.

Ми розібрали код головної функції програми, проте є ще декілька інструкцій на самому початку програми, перед описом головної функції.

Інструкції `#include <iostream>` і `#include <cstdlib>` є директивами препроцесора, які використовують для внесення до програмного коду заголовкових файлів, що містять важливу інформацію, необхідну для виконання програми. Фактично, йдеться про використання базових стандартних бібліотек (`<iostream>` відповідає стандартній бібліотеці введення/виведення, а `<cstdlib>` відповідає стандартній загальній бібліотеці).

Подробиці

Стандартна бібліотека введення/виведення містить визначення таких ідентифікаторів, як `cout` (консольний пристрій) і `cin` (клавіатура - пристрій уведення). Стандартна загальна бібліотека містить визначення, окрім іншого, функції `system()`.

Інструкція `using namespace std` є командою для використання стандартного *простору імен* `std`.

Теорія

Простір імен - це певне абстрактне сховище, що дозволяє уникнути конфлікту імен. Створюючи програмний код, необхідно зазначати, який саме простір імен буде використано для "зберігання" назв утиліт, що створюються в програмі. Ми будемо використовувати стандартний простір імен, який називається `std`.

Власне на цьому ми закінчуємо аналіз програмного коду. Зазначимо, що надалі під час написання програм ми плануємо використовувати такий шаблонний код (без урахування коментарів):

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   // Код головної функції
   system("pause>nul");
   return 0;
}
```

Якщо ж читач вирішить проблему з кодуванням і затримкою консольного вікна способом, відмінним від вищезазначеного, то шаблонний код буде ще простішим:

```
#include <iostream>
using namespace std;
int main(){
   // Код головної функції
   return 0;
}
```

Тепер переходимо до розгляду наступного прикладу. В ньому ми познайомимося з *оператором уведення* (дізнаємося, як уводиться інформація з клавіатури) та *змінними*.

Знайомство зі змінними

Далі ми розглянемо задачу про переведення значення відстані в милях у значення в кілометрах. Розв'язуючи задачу, нам доведеться вводити дані з клавіатури й передавати їх до програми, щоб виконати необхідні обчислення.

До уваги

Британська статутна миля - міра довжини, що дорівнює 1609344 міліметрам, або це 1.609344 кілометра. Щоб перевести значення, подане в милях, у кілометри, достатньо помножити значення в милях на 1.609344.

Програма досить проста: спочатку виводимо запит на введення користувачем відстані в милях, потім уведене значення зчитуємо, виконуємо необхідні обчислення, а результат виводимо в консольне вікно.

У процесі обчислень нам потрібно кудись записувати введене користувачем значення, так само як і результати обчислень. Для цього в програмі використовують *змінні*.

Теорія

Змінна - іменована область пам'яті, до якої можна записати значення і з якої можна прочитати значення. У C++ змінні перед використанням оголошують. Для кожної змінної вказують тип значення, яке можна зберігати в змінній. Тип змінної визначають через ключове слово - ідентифікатор типу. Найактуальніші базові типи: `int` (цілі числа), `double` (дійсні числа), `char` (символи).

У програмі змінну оголошують у будь-якому місці (але до першого її використання). Межі доступності змінної визначають блоком, у якому вона оголошена. Блок, у свою чергу, обмежують парою фігурних дужок. Якщо змінна оголошена в головній функції програми, вона доступна в будь-якому місці головної функції.

Окрім змінних, нерідко використовують ***константи***. Константа від змінної відрізняється тим, що значення константи змінити неможна. Константи описують так само, як змінні, але з ідентифікатором `const`.

Тепер розглянемо програмний код, поданий у лістингу 1.2.

Лістинг 1.2. Переведення миль у кілометри

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
```

```
    system("chcp 1251>nul");
    // Константа визначає, скільки в одній милі кілометрів:
    const double kmInMile=1.609344;
    // Змінні для запису відстані у милях і кілометрах:
    double distMile,distKm;
    // Запит на введення відстані в милях:
    cout<<"Вкажіть відстань у милях: ";
    // Зчитування значення для відстані в милях:
    cin>>distMile;
    // Обчислення відстані в кілометрах:
    distKm=distMile*kmInMile;
    // Відображення результату обчислень:
    cout<<"Відстань (у км): "<<distKm<<endl;
    // Затримання консольного вікна:
    system("pause>nul");
    return 0;
}
```

У тілі головної функції програми інструкцією `const double kmInMile=1.609344` оголошено константу дійсного числового типу. Константа називається `kmInMile`, і її значення дорівнює `1.609344` (кілометрів в одній милі). Також у програмі оголошено дві змінні `distMile` і `distKm`, обидві типу `double`. У змінну `distMile` записується значення для відстані в милях, яке користувач уводить з клавіатури. Відбувається це так: спочатку командою `cout<<"Вкажіть відстань у милях: "` у консолі відображається повідомлення з пропозицією ввести числове значення. Потім користувач із клавіатури вводить значення й натискає клавішу <Enter>. Така ситуація оброблюється командою `cin>>distMile`, основу якої складає *оператор уведення* `>>`. Ідентифікатор `cin` (абревіатура від *console input*) позначає пристрій уведення (як замовчуванням, ідеться про клавіатуру). Праворуч від

оператора уведення зазначають змінну (у цьому випадку `distMile`), у яку записується значення, що вводить користувач.

До уваги

Оскільки змінна `distMile` оголошена з типом `double`, то введене користувачем значення автоматично приводиться до типу `double`.

Командою `distKm=distMile*kmInMile` обчислюється відстань у кілометрах (відстань у милях `distMile` множиться на константу `kmInMile`, яка визначає кількість кілометрів у одній милі).

Теорія

Тут ми зустрічаємо оператор множення `*`. Інші арифметичні оператори: оператор додавання `+`, оператор віднімання `-`, оператор ділення `/`. Оператор ділення в C++ специфічний: якщо обидва операнди цілочислові, то ділення виконується націло (цілочислове ділення). Наприклад, результатом виразу `9/4` є значення `2`. Щоб ділення виконувалося на множині дійсних чисел, перед відповідною командою вказують інструкцію `(double)` (ключове слово `double` у круглих дужках). Наприклад, результатом виразу `(double)9/4` є значення `2.25`.

Присвоювання виконується за допомогою оператора `=`. Значення, вказане праворуч від оператора, записується в змінну, вказану ліворуч від оператора.

Результат обчислень записується в змінну `distKm`. Після цього командою `cout<<"Відстань (у км): "<<distKm<<endl` відображається обчислене значення в консольному вікні. Результат виконання програми може бути таким (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (із лістингу 1.2)

```
Вкажіть відстань у милях: 3.5
Відстань (у км): 5.6327
```

Вище ми оперували з нецілими (дійсними) числовими значеннями. Такий підхід математично коректний, але з практичної точки зору не дуже зручний, оскільки в реальності для визначення відстаней використовуються різномасштабні метричні одиниці. Схожу ситуацію розглянемо далі.

До уваги

Пояснимо, що мається на увазі. Скажімо, у попередньому прикладі під час тестування програми ми вводили значення 3.5 для відстані в милях. Якщо ж цю саму відстань виразити в милях і футах, то буде 3 милі й 2640 футів (у одній милі 5280 футів). Та сама відстань у кілометрах складає величину 5.6327, або, що те ж саме, 5 кілометрів і 632 метри (якщо відкинути 70 сантиметрів).

Розглянемо аналогічний приклад, але дещо змінимо загальну постановку задачі. Зокрема, відстань вкажемо в милях і футах, а перераховуватимо - у кілометри й метри. Відповідний програмний код подано в лістингу 1.3.

Лістинг 1.3. Переведення миль і футів у кілометри й метри

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Кількість футів у милі:
   const int ftInMile=5280;
   // Кількість кілометрів у милі:
   const double kmInMile=1.609344;
   // Цілочислові змінні для запису кількості
   // миль, футів, кілометрів і метрів:
   int dMile,dFt,dKm,dM;
   // Уведення відстані в милях і футах:
   cout<<"Відстань у милях і футах."<<endl;
   cout<<"Милі: ";
   cin>>dMile;
   cout<<"Фути: ";
   cin>>dFt;
   // Відстань у милях:
```

```
    double distMile=dMile+(double)dFt/ftInMile;
    // Відстань у кілометрах:
    double distKm=distMile*kmInMile;
    // Тільки кілометри:
    dKm=(int)distKm;
    // Метри:
    dM=(int)((distKm-dKm)*1000);
    // Відображення результату обчислень:
    cout<<"Відстань у кілометрах і метрах."<<endl;
    cout<<"Кілометри: "<<dKm<<endl;
    cout<<"Метри: "<<dM<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Проаналізуємо програму. В цьому випадку оголошується цілочислова константа `ftInMile` зі значенням `5280`, яке визначає кількість футів у милі. Дійсна числова константа `kmInMile` має значення `1.609344`, яке визначає кількість кілометрів у милі. Цілочислові змінні `dMile`, `dFt`, `dKm` і `dM` ми використовуємо для зчитування кількості миль, футів і запису обчислених значень для кількості кілометрів і метрів. Зчитування миль і футів виконуємо за допомогою команд `cin>>dMile` і `cin>>dFt` відповідно. Далі в програмному коді за допомогою команди `double distMile=dMile+(double)dFt/ftInMile` обчислюємо відстань у милях (маємо на увазі відстань, виражену тільки в милях, і тому таке значення в загальному випадку є дійсним числом). Ця команда є показовою в багатьох відношеннях. По-перше, командою не тільки обчислюється значення, а й одночасно оголошується змінна, в яку записується значення. Конкретніше, оголошуємо змінну `distMile` типу `double`, і змінній присвоюємо значення виразу `dMile+(double)dFt/ftInMile`. Цей вираз є сумою значення

змінної dMile і виразу (double)dFt/ftInMile, який, у свою чергу, є часткою від ділення значення змінної dFt на значення змінної ftInMile. Оскільки обидві змінні цілочислові, а ділення потрібно виконувати на множині дійсних чисел (тобто звичайне ділення, не цілочислове), то перед виразом вказуємо інструкцію (double).

До уваги

Відношення значень змінних dFt і ftInMile є частиною милі, яка у вихідному представленні виражена у футах.

Відстань у кілометрах обчислюємо командою double distKm=distMile*kmInMile. Тут така ж сама ситуація: оголошуємо змінну distKm, і їй одразу присвоюємо значення, причому значення обчислюємо (як добуток відстані в милях distMile на кількість кілометрів у одній милі kmInMile).

До уваги

Ситуація, коли значення змінної присвоюється під час її оголошення і при цьому значення, яке присвоюється, визначається виразом, називається *динамічною ініціалізацією змінної.*

У змінну distKm записано відстань у кілометрах. Це дійсне число. Нам необхідно отримати з нього цілу частину (кілометри) і дробову частину, яка, будучи помноженою на 1000 (в одному кілометрі 1000 метрів), дає значення для кількості метрів.

Щоб виділити зі значення змінної distKm цілу частину, використовуємо команду dKm=(int)distKm. У ній застосовано явне приведення типу: за допомогою інструкції (int) ми double-значення змінної distKm зводимо до цілочислового значення, відкидаючи дробову частину. Іншими словами, значенням виразу (int)distKm є ціла частина значення змінної distKm, при цьому значення самої змінної distKm не змінюється.

Змінній dM присвоюємо значенням вираз (int)((distKm-dKm)*1000). Його обчислюємо так: різницею значень змінних distKm-dKm є дробова

частина змінної `distKm`. Отримане значення множимо на `1000`, і від результату відкидаємо дробову частину (за рахунок інструкції `(int)`).

Значення, обчислені для змінних `dKm` і `dM`, відображуються в консольному вікні командами `cout<<"Кілометри: "<<dKm<<endl` і `cout<<"Метри: "<<dM<<endl`. Нижче показано, як може виглядати результат виконання програми (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (з лістингу 1.3)

```
Відстань у милях і футах.
Милі: 3
Фути: 2640
Відстань у кілометрах і метрах.
Кілометри: 5
Метри: 632
```

Розглянуті приклади дають деяке уявлення про те, як використовують змінні. Далі ми познайомимося із *функціями*.

Знайомство з функціями

Тепер ми розглянемо таку ж саму задачу про обчислення відстані в кілометрах за значенням, заданим у милях, але цього разу вдамся до допомоги функцій.

Теорія

Функція - іменований блок програмного коду, який можна викликати за іменем. Описуючи функцію зазначають її прототип (тип результату, назву та список аргументів) і, у фігурних дужках, тіло функції (програмний код, який виконується під час виклику функції).

Новий спосіб розв'язання вже знайомої нам задачі наведено в програмному коді в лістингу 1.4.

Лістинг 1.4. Використання функцій

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція для зчитування відстані в милях:
double getDistMile(){
    // Змінна для запису результату функції:
    double dist;
    // Запит на введення відстані в милях:
    cout<<"Вкажіть відстань у милях: ";
    // Зчитування значення для відстані в милях:
    cin>>dist;
    // Результат функції:
    return dist;
}
// Функція для переведення миль у кілометри:
double getDistKm(double dist){
    // В одній милі кілометрів:
    double kmInMile=1.609344;
    // Результат функції:
    return dist*kmInMile;
}
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Змінна для запису відстані в милях:
    double distMile=getDistMile();
    // Виконання обчислень:
    cout<<"Відстань (у км): "<<getDistKm(distMile)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми може бути таким (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (із листингу 1.4)

```
Вкажіть відстань у милях: 3.5
Відстань (у км): 5.6327
```

З точки зору кінцевого результату програма виконує ті ж самі операції, що й в одному з попередніх прикладів (див. лістинг 1.2). Водночас, у цьому випадку код принципово інакший. Проаналізуймо його.

У програмі окрім головної функції `main()`, описано ще дві функції: функція `getDistMile()`, яка призначена для зчитування з клавіатури значення для відстані в милях, і функція `getDistKm()`, призначена для перетворення миль у кілометри. Оголошення функції `getDistMile()` починається з ключового слова `double`: функція повертає результат, і це значення типу `double`. Після імені функції круглі дужки пусті - аргументів у функції немає. У тілі функції командою `double dist` оголошується локальна (доступна лише в тілі функції) змінна, значення якої зчитується з клавіатури (команда `cin>>dist`). Потім значення змінної `dist` повертається як результат функції (команда `return dist`).

Теорія

Нагадаємо, що область доступності змінної обмежена блоком, у якому вона оголошена. Змінні, оголошені в тілі функції, називаються локальними и доступні лише в тілі функції.

Місце в пам'яті під локальні змінні виділяється під час виклику функції, а після виконання функції локальні змінні видаляються з пам'яті. Локальна змінна, таким чином, існує лише поки виконується функція.

Варто розрізняти опис функції та її виклик. Опис функції не передбачає виконання коду функції. Код функції виконують, викликаючи функцію.

Інструкція `return` закінчує виконання функції. Якщо після інструкції `return` вказано значення, то воно повертається як результат функції.

Функція `getDistKm()` призначена для переведення миль у кілометри: у неї один аргумент (позначений як `dist`) типу `double`, а як результат функція повертає також значення типу `double`. Командою `return dist*kmInMile` повертається результат функції, який є добутком значення аргументу функції `dist` на значення `1.609344` локальної змінної `kmInMile`.

Теорія

Аргументи функції "мають силу" локальної змінної. Оголошення аргументу під час опису функції розповсюджується лише на тіло функції.

У головній функції програми командою `double distMile=getDistMile()` оголошується змінна `distMile`, а результат функції `getDistMile()` присвоюється значенням цій змінній. Виклик функції `getDistMile()` спричинює відображення запиту в консольному вікні й закінчується зчитуванням значення з клавіатури (і поверненням цього значення функцією). Записане в змінну `distMile` значення передається аргументом під час виклику функції `getDistKm()` (мається на увазі команда `cout<<"Відстань (в км): "<<getDistKm(distMile)<<endl`). У результаті отримуємо в консольному вікні повідомлення зі значенням для відстані в кілометрах.

Знайомство з оператором циклу

У наступному прикладі обчислюється сума квадратів натуральних чисел.

До уваги

Йдеться про обчислення суми $1^2 + 2^2 + 3^2 + \cdots + n^2$, де верхня межа суми n задана. В програмі така сума обчислюється напряму, безпосереднім додаванням. Щоб перевірити коректність результату можна скористатися формулою $1^2 + 2^2 + 3^2 + \cdots + n^2 = \frac{n(n+1)(2n+1)}{6}$.

Для обчислення використовуємо оператор циклу `while`. Приклад наведено в лістингу 1.5.

Лістинг 1.5. Обчислення суми квадратів натуральних чисел

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Верхня межа суми, значення суми й індексна змінна:
   int n=10,s=0,k=1;
   // Оператор циклу для обчислення суми:
   while(k<=n){
      // Додавання нового доданка до суми:
      s=s+k*k;
      // Збільшення (на 1) значення індексної змінної:
      k++;
   }
   // Відображення результату обчислень:
   cout<<"Сума квадратів від 1 до "<<n<<": "<<s<<endl;
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 1.5)

```
Сума квадратів від 1 до 10: 385
```

У програмі командою `int n=10,s=0,k=1` оголошуються три цілочислові змінні, причому їм одразу присвоюється значення. Змінна `n` визначає верхню межу для суми, у змінну `s` буде записано суму квадратів натуральних чисел, а змінна `k` знадобиться в операторі циклу для "загинання пальців" - підрахунку

кількості доданків, доданих до суми (змінну k будемо називати індексною змінною). Усі основні обчислення виконуються в операторі циклу, опис якого починається з ключового слова while. У круглих дужках після ключового слова while є умова k<=n, яка означає, що оператор циклу виконується доти, доки значення змінної k не перевищує значення змінної n. За кожний цикл виконуються дві команди: командою s=s+k*k до значення змінної s додається квадрат поточного значення змінної k, після чого командою k++ значення змінної k збільшується на одиницю. Усе це триває, допоки значення змінної k не стане більшим за значення змінної n.

Теорія

У виразі k<=n використано оператор порівняння <= (менше або дорівнює), а результатом виразу є значення ***логічного типу***. Логічний тип має всього два значення: true (істина) і false (хиба).

Оператор циклу while виконується доти, поки вираз, зазначений у круглих дужках після ключового слова while, має значення true. Умова перевіряється кожного разу перед початком виконання команд у тілі оператора циклу.

Унарний оператор ***інкремента*** ++ дозволяє збільшити на одиницю значення свого операнда. Команда виду k++ еквівалентна команді k=k+1.

Унарний оператор ***декремента*** -- зменшує на одиницю значення свого операнда. Команда k-- еквівалентна команді k=k-1.

Результат обчислень в консольному вікні відображається командою cout<<"Сума квадратів від 1 до "<<n<<": "<<s<<endl. Тут ми використовуємо значення змінної n для верхньої межі суми й обчислене значення s для суми квадратів натуральних чисел.

Дещо видозмінену версію розглянутого вище програмного коду наведено в лістингу 1.6.

Лістинг 1.6. Альтернативний спосіб обчислення суми квадратів

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Верхня межа суми та значення суми:
    int n,s=0;
    // Введення значення для верхньої межі ряду:
    cout<<"Верхня межа суми: ";
    cin>>n;
    // Оператор циклу для обчислення суми:
    while(n){
        // Додавання нового доданка до суми:
        s+=n*n;
        // Зменшення (на 1) значення змінної:
        n--;
    }
    // Відображення результату обчислень:
    cout<<"Сума квадратів: "<<s<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У цій програмі декілька нових моментів. По-перше, ми скористалися тим, що число можна використовувати замість логічних значень: відмінне від нуля число інтерпретується як значення `true`, а нульове числове значення інтерпретується як `false`. Це дозволило нам обійтися без індексної змінної `k`. По-друге, ми скористалися скороченою формою оператора присвоювання.

Теорія

У C++ існують ***скорочені форми оператора присвоювання***. Наприклад, замість команди виду `x=x+y` можна використовувати команду `x+=y`. Це ж саме зауваження

стосується й інших арифметичних операторів: замість команди `x=x-y` можна використовувати команду `x-=y`, замість команди `x=x*y` використовують команду `x*=y`, і так далі.

Значення змінної `n`, що визначає верхню межу суми, тепер зчитується з клавіатури. Ця ж змінна вказана умовою в операторі циклу `while`. Оператор циклу працює, поки значення змінної `n` відмінне від нуля. В тілі оператора циклу послідовно виконуються команди `s+=n*n` і `n--`. Перша з них до поточного значення змінної `s` додає квадрат поточного значення змінної `n`, після чого командою `n--` значення змінної `n` зменшується на одиницю. Весь процес триває доти, допоки значення змінної `n` не дорівнюватиме нулю.

До уваги

Таким чином, у програмі обчислюється сума $n^2 + (n-1)^2 + \cdots + 2^2 + 1^2$, яка, очевидно, збігається із сумою $1^2 + 2^2 + \cdots + n^2$.

Нижче показано, як може виглядати результат виконання програми (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (із лістингу 1.6)

```
Верхня межа суми: 10
Сума квадратів: 385
```

У розглянутому програмного коді є один невеликий недолік: якщо користувач уведе від'ємне значення для змінної `n`, отримаємо нескінченний цикл, оскільки при зменшенні від'ємного значення нулем воно не стане. В наступному прикладі ми "заблокуємо" цю неприємність.

Знайомство з умовним оператором

У новій версії попередньої програми додано перевірку значення змінної `n` після того, як воно зчитане з клавіатури. Для цього використано *умовний оператор* `if`.

Теорія

Умовний оператор описують так: після ключового слова `if` у круглих дужках вказується умова. Далі йде блок команд у фігурних дужках, які виконуються при істинності умови. Якщо умова хибна, виконується блок команд у фігурних дужках після ключового слова `else`.

Розглянемо програмний код прикладу в лістингу 1.7.

Лістинг 1.7. Перевірка коректності введеного значення

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Верхня межа суми й значення суми:
   int n,s=0;
   // Введення значення для верхньої межі ряду:
   cout<<"Верхня межа суми: ";
   cin>>n;
   // Якщо введено додатне число:
   if(n>0){
      // Оператор циклу для обчислення суми:
      while(n){
         // Додавання нового доданка до суми:
         s+=n*n;
         // Зменшення значення змінної:
         n--;
      }
      // Відображення результату обчислень:
      cout<<"Сума квадратів: "<<s<<endl;
   }
   // Якщо введено від'ємне число чи нуль:
```

```
    else{
        cout<<"Вказано некоректне значення"<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

В умовному операторі перевіряється умова `n>0`, і якщо вона істинна, виконується блок команд, якими обчислюється сума квадратів натуральних чисел. Якщо ж умова `n>0` хибна, то виконуються команди в `else`-блоці. У цьому випадку там усього одна команда `cout<<"Вказано некоректне значення"<<endl`, якою в консольне вікно виводиться відповідне повідомлення. Нижче показано, як може виглядати результат виконання програми, якщо користувач уводить коректне значення для верхньої межі суми (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (із лістингу 1.7)

```
Верхня межа суми: 10
Сума квадратів: 385
```

Якщо користувач уводить некоректне значення, результат може бути таким (жирним шрифтом виділено значення, що вводить користувач):

Результат виконання програми (із лістингу 1.7)

```
Верхня межа суми: -5
Вказано некоректне значення
```

Ще одна корисна конструкція, з якою часто доводиться мати справу - *масиви*. Наступний приклад дає певне уявлення про те, як масиви використовують у програмних кодах.

Знайомство з масивами

Масив (одномірний) є набором елементів, об'єднаних в одну спільну "конструкцію". Кожен елемент масиву, по суті, є окремою змінною. Але доступ до таких змінних отримують через ім'я масиву. Оскільки в масиві елементів багато, то для однозначної ідентифікації елемента в масиві вказується *індекс елемента*.

Теорія

Масив оголошується досить просто. Спочатку вказують ідентифікатор типу, до якого належать елементи масиву (всі елементи масиву повинні бути одного типу). Потім вказують ім'я масиву й у квадратних дужках - розмір масиву (кількість елементів у масиві). Під час звертання до елемента масиву вказують ім'я масиву й у квадратних дужках індекс елемента. Індексація елементів починається з нуля, тому перший елемент масиву має індекс 0, а індекс останнього елемента масиву на одиницю менший за розмір масиву. Розмір масиву задають константою (це не може бути звичайна змінна).

Розглянемо приклад, у якому створюється числовий масив і заповнюється біноміальними коефіцієнтами.

До уваги

За означенням, біноміальні коефіцієнти обчислюються як $C_n^k = \frac{n!}{k!(n-k)!}$, де використано позначення для факторіала числа $m! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot m$ (добуток натуральних чисел від 1 до m). Нескладно помітити, що $C_n^k = C_n^{n-k}$, а також $C_n^0 = 1$, $C_n^1 = n$, $C_n^2 = \frac{n(n-1)}{2}$ і так далі. Ми скористаємося в програмі рекурентним співвідношенням $C_n^{k+1} = C_n^k \cdot \frac{n-k}{k+1}$.

Звернемося до програмного коду, наведеного в лістингу 1.8.

Лістинг 1.8. Біноміальні коефіцієнти

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

```
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Константа для визначення розміру масиву:
    const int n=10;
    // Створення цілочислового масиву:
    int bnm[n+1];
    // Індексна змінна:
    int k=0;
    // Перший елемент масиву:
    bnm[0]=1;
    // Відображення першого елемента масиву:
    cout<<bnm[0];
    // Оператор циклу для заповнення масиву:
    while(k<n){
       // Значення елемента масиву:
       bnm[k+1]=bnm[k]*(n-k)/(k+1);
      // Відображення значення елемента масиву:
      cout<<" "<<bnm[k+1];
      // Зміна значення індексної змінної:
      k++;
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У результаті виконання програми отримуємо таке:

Результат виконання програми (із лістингу 1.8)

```
1 10 45 120 210 252 210 120 45 10 1
```

Проаналізуємо програмний код. Спершу командою `const int n=10` оголошується цілочислова константа `n` зі значенням `10`. Потім створюється масив `bnm`, для чого використано команду `int bnm[n+1]`. Під час створення масиву під нього виділяється місце в пам'яті, проте значеннями елементи масиву не заповнюються. Розмір створеного масиву на одиницю більший за значення константи `n`, тому індекси елементів масиву перебувають у діапазоні значень від `0` до `n` включно. Для перебору елементів масиву в операторі циклу оголошується цілочислова індексна змінну `k` з початковим значенням `0`. Також командою `bnm[0]=1` явно присвоюється значення першому елементу масиву. Воно відразу відображається в консольному вікні (команда `cout<<bnm[0]`). Для заповнення інших елементів масиву запускається оператор циклу `while`. Умовою, що перевіряється, є `k<n`, тому, с урахуванням команди `k++` в тілі оператора циклу, змінна `k` послідовно перебирає значення від `0` до `n-1` включно. Значення чергового елемента обчислюється командою `bnm[k+1]=bnm[k]*(n-k)/(k+1)`. Значення обчисленого у такий спосіб елемента відображається командою `cout<<" "<<bnm[k+1]` у консольному вікні. Таким чином, біноміальні коефіцієнти (значення елементів масиву `bnm`) відображаються в одному рядку, і як роздільник між елементами використовується пробіл.

Задачі для самостійного розв'язування

1. Напишіть програму, яка в консольне вікно виводить натуральні числа від 1 до 10.

2. Напишіть програму, яка відображає п'ять перших непарних чисел.

3. Напишіть програму, яка відображає в консолі числа, які під час ділення на 4 у залишку дають 3 (такі числа описуються формулою $4k + 3$, де число k набуває значення 0, 1, 2 і так далі). Можна скористатися тим, що залишок від

цілочислового ділення обчислюється оператором %. Кількість чисел уводить користувач з клавіатури.

4. Напишіть програму, яка відображає в консолі числа Фібоначчі (перші два числа в послідовності Фібоначчі дорівнюють одиниці, а кожне наступне – сумі двох попередніх). Кількість чисел, які виводяться, уводить користувач з клавіатури.

5. Напишіть програму, яка відображає (без попереднього запису в масив) біноміальних коефіцієнтів C_n^k (параметр $k = 0,1,2,\dots,n$). Параметр n для нижнього індексу коефіцієнтів уводять із клавіатури. Передбачити перевірку уведеного користувачем значення на коректність.

6. Напишіть програму, яка переводить кілометри в милі.

7. Напишіть програму, яка переводить відстань, вказану в кілометрах і метрах, у милі й фути.

8. Напишіть програму, яка переводить відстань, вказану в *сажнях* (1 сажень дорівнює 2.16 метра), у метри.

9. Напишіть програму, яка переводить відстань, вказану в сажнях й аршинах (1 сажень дорівнює 2.16 метра, а 3 аршини дорівнюють 1 сажню), у метри й сантиметри (в 1 метрі 100 сантиметрів).

10. Напишіть програму з функцією переведення швидкості, вказаної в км/год ("кілометри за годину") у м/с ("метри за секунду"). Урахувати, що в 1 кілометрі - 1000 метрів, у 1 хвилині - 60 секунд, у 1 годині - 60 хвилин.

11. Напишіть програму з функцією переведення швидкості, вказаної в м/с ("метри за секунду") у км/год ("кілометри за годину").

12. Напишіть програму з функцією обчислення суми натуральних чисел.

13. Напишіть програму з функцією обчислення суми непарних натуральних чисел.

14. Напишіть програму, яка створює масив і заповнює його парними натуральними числами.

15. Напишіть програму, яка створює масив і заповнює його непарними натуральними числами.

16. Напишіть програму, яка створює масив і заповнює його квадратами натуральних чисел (значення 1^2, 2^2, 3^2 і так далі).

17. Напишіть програму, яка створює масив і заповнює його степенями двійки (значення 2^0, 2^1, 2^2, 2^3 і так далі).

18. Напишіть програму, яка створює масив і заповнює його числами Фібоначчі (перші два числа дорівнюють одиниці, а кожне наступне дорівнює сумі двох попередніх).

19. Напишіть програму, яка створює масив і заповнює його так: елементи з парними індексами значенням отримують індекс, а елементи з непарними індексами значенням отримують квадрат індексу. Іншими словами, масив заповнюється числами: 0, 1^2, 2, 3^2, 4 , 5^2, 6, 7^2 і так далі.

20. Напишіть програму, яка створює масив і заповнює його поелементно: користувач для кожного елемента вводить значення з клавіатури (організувати введення значень для елементів масиву з залученням оператора циклу).

Глава 2

Керуючі інструкції

Передай йому щоб забирався звідси до дідька. Я тут гулять буду.
з х/ф "Пропала грамота"

У цій главі розглянемо приклади, у яких ключове місце займають керуючі інструкції: оператори циклу, оператор вибору й умовний оператор. У попередній главі ми зустрічалися з оператором циклу `while`. Далі ми познайомимося:

- з оператором циклу `for`;
- з оператором циклу `do-while`;
- розглянемо особливості умовного оператора `if`;
- розглянемо оператор вибору `switch`,

а також розширимо пізнання щодо деяких інших програмних конструкцій, операторів і прийомів програмування.

Оператор циклу `for`

У попередній главі ми розглянули приклад, у якому обчислювалася сума квадратів натуральних чисел. Там ми використовували оператор циклу `while`. Зараз розв'яжемо аналогічну задачу, але застосуємо оператор циклу `for`.

Теорія

Оператор циклу `for` описують таким чином: після ключового слова `for` у круглих дужках вказують три блоки інструкцій (блоки розділяють крапкою з комою). У фігурних дужках розташовують команди (тіло оператора циклу), що виконуються у рамках оператора циклу.

Оператор циклу `for` працює за таким алгоритмом:

- інструкції в першому блоці виконуються один раз на початку виконання циклу;
- перевіряється умова у другому блоці інструкцій - для продовження процесу вона

повинна бути істинною;

- виконуються команди в тілі оператора циклу (команди у фігурних дужках);
- виконуються команди в третьому блоці інструкцій;
- перевіряється умова в другому блоці інструкцій, і якщо вона істинна - виконуються команди в тілі оператора циклу, і так далі;
- якщо умова в другому блоці хибна, виконання оператора циклу закінчується.

Розглянемо програмний код із лістингу 2.1.

Лістинг 2.1. Обчислення суми квадратів за допомогою оператора циклу for

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Змінна для запису верхньої межі суми,
   // змінна для запису значення суми й індексна
   // змінна для оператора циклу:
   int n,s=0,k;
   cout<<"Вкажіть верхню межу для суми: ";
   // Зчитування верхньої межі для суми:
   cin>>n;
   // Оператор циклу для обчислення суми квадратів чисел:
   for(k=1;k<=n;k++){
      // Додавання до суми наступного доданка:
      s+=k*k;
   }
   // Відображення результату обчислень:
   cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl;
   // Затримка консольного вікна:
   system("pause>nul");
```

```
    return 0;
}
```

Нижче наведено можливий результат виконання програми (жирним шрифтом виділено уведене користувачем значення):

Результат виконання програми (із лістингу 2.1)

```
Вкажіть верхню межу для суми: 10
Сума квадратів чисел від 1 до 10: 385
```

У програмі оголошуються три цілочислові змінні: у змінну `n` буде записуватися значення для верхньої межі суми, змінна `s` із початковим нульовим значенням потрібна для запису значення суми квадратів натуральних чисел, а індексна змінна `k` використовується в операторі циклу `for` для підрахунку кількості доданків, що додаються до суми.

Після зчитування значення для змінної `n` запускається оператор циклу `for`. Перший блок у круглих дужках після ключового слова `for` складається з однієї команди `k=1`, яка виконується один раз на початку виконання оператора циклу. Після присвоювання змінній `k` одиничного значення, перевіряється умова `k<=n`. Якщо умова істинна, то виконується команда `s+=k*k` в тілі оператора циклу, після чого виконується команда `k++` і знову перевіряється умова `k<=n`. Якщо умова залишається істинною, виконуються команди `s+=k*k` і `k++`, опісля знову перевіряється умова `k<=n`, і так далі. Процес триває доти, поки під час перевірки виразу `k<=n` не буде отримано значення `false`.

Після закінчення виконання оператора циклу в змінну `s` записана сума квадратів натуральних чисел від `1` до `n`. Результат обчислень відображається командою `cout<<"Сума квадратів чисел від 1 до "<<n<<":"<< s<<endl`.

Варто зазначити, що програмний код "відловлює" ситуацію, коли користувач уводить некоректне (від'ємне) значення для змінної `n`. У такому випадку для суми буде обчислено нульове значення:

Результат виконання програми (із лістингу 2.1)

```
Вкажіть верхню межу для суми: -5
Сума квадратів чисел від 1 до -5: 0
```

Причина в тому, що за від'ємного значення змінної `n` під час першої перевірки умови `k<=n`, що відбувається одразу після виконання команди `k=1` в першому блоці інструкцій, умова виявляється хибною, і виконання оператора циклу на цьому закінчується. Програма відображає значення змінної `s`, яке в цьому випадку дорівнює початковому нульовому значенню цієї змінної.

Блоки інструкцій в операторі циклу `for` можуть бути пустими чи навпаки, містити по декілька команд. Якщо блок інструкцій має декілька команд, вони відокремлюються комами. Як приклади різних варіантів синтаксису цього оператора розглянемо все той же приклад обчислення суми квадратів натуральних чисел, проте кожного разу оператор циклу `for` буде використано по-іншому.

У лістингу 2.2 наведено версію програми, у якій блоки інструкцій в операторі циклу `for` місять декілька команд (для зменшення обсягу коду всі коментарі видалено).

Лістинг 2.2. Блоки інструкцій в операторі циклу `for` містять декілька команд

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   int n,s,k;
   cout<<"Вкажіть верхню межу для суми: ";
   cin>>n;
   for(k=1,s=0;k<=n;s+=k*k,k++);
   cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl;
```

```
    system("pause>nul");
    return 0;
}
```

У цьому випадку присвоювання нульового початкового значення змінній `s` винесено в перший блок інструкцій оператора `for`, а команда `s+=k*k` тепер знаходиться в третьому блоці інструкцій (перед командою `k++`). Тіло оператора циклу, таким чином, виявилося пустим (там тепер немає команд), тому ми не використовуємо фігурні дужки, а просто ставимо після `for`-інструкції крапку з комою.

Певного роду протилежну ситуацію наведено в лістингу 2.3 - там перший і третій блоки оператора циклу `for` пусті.

Лістинг 2.3. Перший і третій блоки інструкцій в операторі циклу `for` пусті

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    system("chcp 1251>nul");
    int n,s=0,k=1;
    cout<<"Вкажіть верхню межу для суми: ";
    cin>>n;
    for(;k<=n;){
        s+=k*k;
        k++;
    }
    cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl;
    system("pause>nul");
    return 0;
}
```

Початкові значення змінним `k` і `s` присвоюють, коли їх оголошують, тому перший блок інструкцій в операторі циклу `for` пустий. Команди `s+=k*k` і `k++` винесено в тіло оператора циклу, тому пустий і третій блок інструкцій в операторі циклу `for`.

Нарешті, досить екзотична ситуація проілюстрована в лістингу 2.4. Там усі три блоки інструкцій (включаючи другий блок з умовою, що регулює виконання оператора циклу) пусті.

Лістинг 2.4. Усе блоки інструкцій в операторі циклу `for` пусті

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   system("chcp 1251>nul");
   int n,s=0,k=1;
   cout<<"Вкажіть верхню межу для суми: ";
   cin>>n;
   for(;;){
      s+=k*k;
      k++;
      if(k>n){
         break;
      }
   }
   cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl;
   system("pause>nul");
   return 0;
}
```

Пустий другий блок в операторі циклу `for` еквівалентний умові `true`, тому формально виходить нескінченний цикл. Щоб вийти з такого нескінченного циклу, в тіло оператора циклу `for` додано умовний оператор `if`. В умовному

операторі перевіряється умова k>n. Якщо вона істинна, то виконується лише одна команда break, яка закінчує реалізацію оператора циклу.

Подробиці



Тут використано спрощену форму умовного оператора, яка не має else-гілки. Умовний оператор у такій "полегшеній" версії реалізується наступним чином: перевіряється умова після ключового слова if, і якщо вона істинна, виконуються команди у фігурних дужках після if-інструкції. Якщо умова хибна, то нічого не відбувається й керування переходить до команди після умовного оператора.

Щодо інструкції break, то це стандартна команда для виходу із операторів циклу (таких як while, do-while і for), а також оператора вибору switch. Є ще одна корисна інструкція continue, яка дозволяє достроково закінчити не весь оператор циклу, а лише поточну його ітерацію (поточний цикл).

Оператор циклу do-while

Окрім операторів циклу while і for, у C++ є оператор циклу do-while.

Теорія

Опис оператора циклу do-while починають з ключового слова do, після якого у фігурних дужках (тіло оператора циклу) вказують команди, які виконуються в контексті оператора циклу. Після блока із фігурних дужок іде інструкція while, а після неї, у круглих дужках, вказують умову. Оператор циклу виконується доти, поки істинна ця умова. Умова перевіряється кожного разу, як закінчено виконання команд у тілі оператора циклу.

Оператор циклу do-while принципово відрізняється від оператора циклу while тим, що в операторі do-while спочатку виконуються команди, а потім перевіряється умова, у той час як в операторі while спочатку перевіряється умова, а після цього виконуються команди. Таким чином, команди в тілі оператора циклу do-while виконуються, принаймні, один раз.

Приклад використання оператора циклу do-while наведено в лістингу 2.5. У програмі обчислюється значення експоненти для заданого аргументу.

Результат перевіряється вбудованою математичною функцію `exp()`, призначеною для цих же цілей.

До уваги

Для обчислення значення експоненти від аргументу x можна використовувати ряд $\exp(x) = \sum_{k=0}^{\infty} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^k}{k!} + \cdots$. На практиці, оскільки обчислити нескінченну суму проблематично, обчислюється обмежена сума $\exp(x) \approx \sum_{k=0}^{n} \frac{x^k}{k!} = 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \cdots + \frac{x^n}{n!}$. Таким чином, обчислення значення експоненти зводиться до обчислення відповідної суми. Чим більша кількість доданків, тим точніше обчислюється значення для експоненти.

Обчислювану суму зручно представити у вигляді $\exp(x) \approx \sum_{k=0}^{n} q_k$, де позначено $q_k = \frac{x^k}{k!}$. Отже, має місце співвідношення $q_{k+1} = q_k \cdot \frac{x}{k+1}$. Цим співвідношенням скористаємося в програмі під час обчислення суми для експоненти.

Розглянемо наведений нижче програмний код.

Лістинг 2.5. Оператор циклу `do-while`

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Індекс останнього доданка в сумі:
   int n=100;
   // Аргумент для експоненти:
   double x=1;
   // Змінна для запису значення суми, змінна
   // для запису значення добавки і індексна змінна:
   double s=0,q=1,k=0;
   // Оператор циклу для обчислення суми:
```

```
    do{
        // Додавання до суми добавки:
        s+=q;
        // Нове значення індексної змінної:
        k++;
        // Нове значення для добавки:
        q*=x/k;
    }while(k<=n);
    // Відображення результату обчислень:
    cout<<"Обчислене значення: "<<s<<endl;
    // Використання вбудованої функції для обчислення
    // значення експоненти:
    cout<<"Контрольне значення: "<<exp(x)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У програмі оголошується цілочислова змінна `n`, значення якої визначає індекс останнього доданка в сумі (і на одиницю менше за кількість доданків у сумі). У дійсну змінну `x` записується значення аргументу експоненти. Також у програмі оголошуються змінні: `s` із нульовим початковим значенням (значення суми), `q` із одиничним початковим значенням (значення добавки до суми) й індексна змінну `k` із нульовим початковим значенням.

Для обчислення суми використовується оператор циклу `do-while`. У тілі оператора циклу командою `s+=q` до суми додається значення добавки `q`. Потім командою `k++` на одиницю збільшується значення індексної змінної, після чого командою `q*=x/k` обчислюється значення добавки для наступної ітерації. Оператор циклу виконується, поки істинна умова `k<=n`. Після закінчення виконання оператора циклу командою `cout<<"Обчислене значення: "<<s<<endl` відображається обчислене в

програмі значення для експоненти, а для порівняння командою `cout<<"Контрольне значення: "<<exp(x)<<endl` відображається значення для експоненти, обчислене за допомогою функції `exp()`.

До уваги

Для використання функції `exp()` додаємо в програму інструкцію `#include <cmath>`, яка дозволяє підключити бібліотеку математичних функцій.

Результат виконання програми такий:

Результат виконання програми (із лістингу 2.5)

```
Обчислене значення: 2.71828
Контрольне значення: 2.71828
```

Як бачимо, обидва значення фактично збігаються, що свідчить про прийнятну точність обчислень.

Оператор вибору `switch`

Розглянемо наступну просту задачу: користувач уводить число в діапазоні значень від `1` до `3` включно, а програма виводить назву цього числа (`"один"`, `"два"` або `"три"`). При написанні програмного коду скористаємося оператором вибору `switch`.

Теорія

Оператор вибору `switch` дозволяє перевіряти значення деякого виразу, і залежно від отриманого результату виконувати різні блоки команд. У певному сенсі оператор вибору нагадує умовний оператор, проте в цьому випадку альтернатив більше, ніж дві. Щодо значення виразу, що перевіряється, то це може бути або числовий, або символьний (тип `char`) вираз.

Описують оператор вибору таким чином. Після ключового слова `switch` у круглих дужках вказують вираз, значення якого перевіряється. Далі в тілі оператора (у фігурних дужках) ідуть `case`-блоки. У кожному `case`-блоці вказано контрольне значення для порівняння щодо збігу зі значенням виразу, що перевіряється, а також команди, які слід виконати у випадку, якщо збіг має місце. Кожний `case`-блок

зазвичай закінчується інструкцією `break`. Останнім в операторі вибору може бути `default`-блок, команди якого виконуються у випадку, якщо в жодному з `case`-блоків не було збігу контрольного значення і значення виразу.

Розглянемо програмний код у лістингу 2.6. Наведена там програма працює таким чином:

- Запускається оператор циклу, в тілі якого виводиться запит на введення числа (у діапазоні значень від `1` до `3`).
- Уведене користувачем значення зчитується й перевіряється в операторі вибору `switch`.
- Залежно від уведеного значення в консольному вікні відображається повідомлення з назвою введеного числа.

Тепер звернемося до програмного коду:

Лістинг 2.6. Знайомство з оператором вибору `switch`

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Цілочислові змінні:
   int num,k;
   // Виконується оператор циклу:
   for(k=1;k<=5;k++){
      cout<<"Уведіть число від 1 до 3: ";
      // Зчитування значення змінної:
      cin>>num;
      // Виконується оператор вибору:
      switch(num){
         case 1:
            cout<<"Це одиниця"<<endl;
```

```
            break;
        case 2:
            cout<<"Це двійка"<<endl;
            break;
        case 3:
            cout<<"Це трійка"<<endl;
            break;
        default:
            cout<<"Я не знаю такого числа"<<endl;
    }
  }
  // Затримка консольного вікна:
  system("pause>nul");
  return 0;
}
```

Результат виконання програми може бути таким, як показано нижче (жирним шрифтом виділено значення, які ввів користувач):

Результат виконання програми (із лістингу 2.6)

```
Уведіть число від 1 до 3: 2
Це двійка
Уведіть число від 1 до 3: -2
Я не знаю такого числа
Уведіть число від 1 до 3: 1
Це одиниця
Уведіть число від 1 до 3: 3
Це трійка
Уведіть число від 1 до 3: 5
Я не знаю такого числа
```

Щодо програмного коду, то в тілі оператора циклу (виконується рівно п'ять циклів) командою `cout<<"Уведіть число від 1 до 3: "` виводиться

повідомлення з пропозицією ввести число. Уведене користувачем число записується в змінну num завдяки команді cin>>num. Опісля запускається оператор вибору. Виразом, що перевіряється, є змінна num. У трьох case-блоках указані відповідно значення 1, 2 і 3. Якщо значення змінної num дорівнює 1, виконується команда cout<<"Це одиниця"<<endl. Якщо значення змінної num дорівнює 2, виконується команда cout<<"Це двійка"<<endl, а якщо значення змінної num дорівнює 3, то виконується команда cout<<"Це трійка"<<endl. Кожен case-блок закінчують інструкцією break.

До уваги

Оператор вибору працює так: послідовно перебираються контрольні значення в case-блоках до першого збігу. Якщо збіг знайдено, то починається виконання команди після відповідної case-інструкції. Команди виконуються до кінця тіла оператора вибору, або ж поки не зустрінеться інструкція break. Отже, якщо нам треба, щоб реалізувалися команди лише в одному case-блоці, то цей case-блок повинен закінчуватися break-інструкцією.

У default-блоці виконується команда cout<<"Я не знаю такого числа"<<endl. Цей блок "вступає у гру" лише тоді, якщо під час перевірки контрольних значень у case-блоках не було збігів.

Ще одну невелику ілюстрацію до використання оператора вибору switch наведено в лістингу 2.7. Певною мірою цей приклад схожий на попередній, та в ньому є деякі особливості. Зокрема, у програмі генеруються випадкові числа, а оператор вибору містить пусті case-блоки.

Теорія

Для генерування випадкових чисел використовують функцію rand(), яка стає доступною після підключення заголовка <cstdlib>. Результатом функції rand()

повертається ціле, рівномірно розподілене невід'ємне випадкове число від 0 до певної верхньої межі (яка досить велика). Якщо нам треба отримати випадкове число в діапазоні значень від a до b, то можна скористатися виразом a+rand()%(b-a+1). У цьому виразі використано оператор % для обчислення залишку від цілочислового ділення. Якщо rand() - деяке випадкове число, то результат виразу rand()%(b-a+1) - залишок від ділення на число b-a+1, і це число в діапазоні від 0 до b-a. Отже, результатом виразу a+rand()%(b-a+1) є число (випадкове) у діапазоні значень від a до b, що й треба було отримати.

Для генерування випадкових (насправді, псевдовипадкових) чисел потрібне початкове "робоче" числове значення, щоб запустити весь процес. Процедура називається ***ініціалізацією генератора випадкових чисел***. Для виконання ініціалізації використовують функцію srand(), аргументом якій передають числове значення. Конкретне числове значення, передане аргументом функції, особливого навантаження не несе. Важливо те, що при одному й тому ж аргументі функції srand() отримуємо одну й ту саму послідовність (псевдо)випадкових чисел.

Ідея, реалізована в програмі, дуже проста. Виконується декілька ітерацій (циклів), і за кожної ітерації генерується випадкове число в діапазоні можливих значень від 2 до 8 включно. Залежно від отриманого значення оператор вибору switch відображає певне повідомлення:

- для чисел 2, 4 і 8 це повідомлення зі значенням числа і фразою про те, що число є степенем двійки;
- для чисел 3 і 6 – повідомлення зі значенням числа і фразою про те, що число ділиться на три;
- для чисел 5 і 7 повідомлення містить значення числа і його текстову назву.

Розглянемо поданий нижче програмний код:

Лістинг 2.7. Використання оператора switch

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
```

```
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Цілочислова змінна:
    int num;
    // Ініціалізація генератора випадкових чисел:
    srand(2);
    // Оператор циклу:
    for(int k=1;k<=10;k++){
        // Випадкове число від 2 до 8:
        num=2+rand()%7;
        // Оператор вибору:
        switch(num){
            case 3:
            case 6:
                cout<<num<<": число ділиться на три"<<endl;
                break;
            case 2:
            case 4:
            case 8:
                cout<<num<<": степінь двійки"<<endl;
                break;
            case 5:
                cout<<num<<": п'ятірка"<<endl;
                break;
            case 7:
                cout<<num<<": сімка"<<endl;
        }
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми може бути таким (із поправкою на те, що використано генератор випадкових чисел):

Результат виконання програми (із лістингу 2.7)

```
5: п'ятірка
7: сімка
8: степінь двійки
3: число ділиться на три
2: степінь двійки
3: число ділиться на три
4: степінь двійки
8: степінь двійки
4: степінь двійки
6: число ділиться на три
```

У програмі командою `srand(2)` ініціалізується генератор випадкових чисел, і оголошується цілочислова змінна `num`. Цій змінній у тілі оператора циклу командою `num=2+rand()%7` присвоюється випадкове значення в діапазоні значень від 2 до 8 включно.

До уваги

Результат виразу `rand()%7` є залишком від ділення випадкового числа на число 7, тобто це число в діапазоні від 0 до 6. Якщо додати до нього 2, то отримаємо число в діапазоні значень від 2 до 8.

Також зверніть увагу, що індексна змінна `k` оголошена безпосередньо в `for`-операторі в першому блоці інструкцій. Так можна робити, проте слід пам'ятати, що оголошена в `for`-операторі змінна доступна лише в цьому операторі.

Далі в операторі `switch` "перебираються" деякі варіанти. Специфіка оператора вибору пов'язана з наявністю пустих `case`-блоків: після однієї `case`-інструкції одразу йде інша. Загальний ефект такий: для всіх відповідних контрольних значень виконується один і той же блок команд.

До уваги

В останньому case-блоці інструкцію break не використовують, оскільки в ній нема необхідності: це останній блок, і після його виконання інші блоки виконуватися не будуть (оскільки їх просто немає).

Вкладені умовні оператори

Умовні оператори ми уже не раз використовували. Тут розглянемо ситуацію, коли в тілі умовного оператора розміщений інший умовний оператор. Як ілюстрацію реалізуємо програмний код одного із попередніх прикладів (див. лістинг 2.6). Але якщо там ми використовували оператор вибору switch, то тут скористаємося умовними операторами. Програму наведено в лістингу 2.8.

Лістинг 2.8. Вкладені умовні оператори

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Цілочислові змінні:
    int num,k;
    // Виконується оператор циклу:
    for(k=1;k<=5;k++){
        cout<<"Уведіть число від 1 до 3: ";
        // Зчитування значення змінної:
        cin>>num;
        // Зовнішній умовний оператор (1-ий рівень):
        if(num==1){
            cout<<"Це одиниця"<<endl;
        }
```

```
    // Блок else зовнішнього
    // умовного оператора (1-ий рівень):
    else{
        // Внутрішній умовний оператор (2-ий рівень):
        if(num==2){
            cout<<"Це двійка"<<endl;
        }
        // Блок else внутрішнього оператора (2-ий рівень):
        else{
            // Внутрішній умовний оператор (3-ій рівень):
            if(num==3){
                cout<<"Це трійка"<<endl;
            }
            // Блок else внутрішнього умовного
            // оператора (3-ій рівень):
            else{
                cout<<"Я не знаю такого числа"<<endl;
            }
        }
    }
}
// Затримка консольного вікна:
system("pause>nul");
return 0;
}
```

Нижче наведено результат виконання програми (жирним шрифтом виділено значення, які ввів користувач):

Результат виконання програми (із лістингу 2.8)

```
Уведіть число від 1 до 3: 2
Це двійка
Уведіть число від 1 до 3: -2
```

```
Я не знаю такого числа
Уведіть число від 1 до 3: 1
Це одиниця
Уведіть число від 1 до 3: 3
Це трійка
Уведіть число від 1 до 3: 5
Я не знаю такого числа
```

Фактично, результат такий, як і в прикладі з використанням оператора `switch` (див. лістинг 2.6). Тут ми улаштували перевірку значення змінної `num` через вкладені умовні оператори.

Теорія

Оператор рівності == є, разом зі знайомим нам оператором <= (менше чи дорівнює), одним із операторів порівняння. Крім цих операторів, корисними можуть бути оператори < (менше), > (більше), >= (більше чи дорівнює) і != (не дорівнює).

Вкладені оператори циклу

Розглянемо приклад, у якому виконується сортування числового масиву. Задача така. Є масив, заповнений числами. Необхідно розташувати числа (поміняти місцями значення елементів масиву) у порядку збільшення значень.

Подробиці

Для сортування елементів масиву використовуємо ***метод бульбашки***. Суть його в тому, що послідовно перебираються елементи масиву і порівнюються значення двох сусідніх елементів. Якщо значення елемента ліворуч більше, ніж значення елемента праворуч, то такі елементи "обмінюються" значеннями. У результаті одного "проходження" по масиву найбільше значення опиниться в правій крайній позиції (стане значенням останнього елемента). Після ще одного перебору елементів масиву в передостаннього елемента буде друге за величиною значення, і так далі. Таким чином, за один перебір елементів масиву один елемент виявляється з "правильним" значенням.

Програмний код, у якому цілочисловий масив створюється, заповнюється випадковими числами, а потім сортується *методом бульбашки*, наведено у лістингу 2.9.

Лістинг 2.9. Сортування масиву

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Розмір масиву:
   const int n=10;
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Створення масиву:
   int nums[n];
   // Цілочислові змінні:
   int i,j,k,s;
   cout<<"Масив до сортування:\n| ";
   // Заповнення масиву випадковими числами:
   for(k=0;k<n;k++){
      nums[k]=rand()%10;
      cout<<nums[k]<<" | ";
   }
   cout<<"\nМасив після сортування:\n| ";
   // Сортування масиву:
   for(i=1;i<=n-1;i++){
      // Перебір елементів масиву:
      for(j=0;j<n-i;j++){
         // "Обмін" значеннями:
         if(nums[j]>nums[j+1]){
```

```
                s=nums[j+1];
                nums[j+1]=nums[j];
                nums[j]=s;
            }
        }
    }
    // Відображення відсортованого масиву:
    for(k=0;k<n;k++){
        cout<<nums[k]<<" | ";
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми може бути таким:

Результат виконання програми (із лістингу 2.9)

```
Масив до сортування:
| 5 | 6 | 8 | 5 | 4 | 0 | 0 | 1 | 5 | 6 |
Масив після сортування:
| 0 | 0 | 1 | 4 | 5 | 5 | 5 | 6 | 6 | 8 |
```

У програмному коді оголошено цілочислову константу `n`, яка визначає розмір масиву. Сам масив створюється командою `int nums[n]`. Заповнення масиву відбувається за допомогою оператора циклу, в якому перебираються елементи. Значення елементам масиву в тілі оператора циклу (при певному значенні індексної змінної `k`) присвоюється командою `nums[k]=rand()%10` (виходить випадкове число в діапазоні значень від `0` до `9`). Одночасно з присвоюванням значень елементам масиву відбувається відображення значень елементів в консольному вікні.

До уваги

Роздільником під час відображення елементів масиву є вертикальна риска. Для деяких текстових значень у програмі використано інструкцію `\n`, яка є командою переходу до нового рядка в процесі відображення відповідного тексту в консолі. Інструкцію `\n` включають безпосередньо в текст, а розрив рядка виконується в том місці, де знаходиться ця інструкція.

Зверніть увагу, що для масиву, який складається з `n` елементів, змінна `k`, яка визначає індекс елемента, пробігає значення в діапазоні від `0` до `n-1` включно.

Після заповнення масиву та відображення в консолі значень його елементів, починається сортування масиву. Для цього використовують вкладені оператори циклу `for`. Зовнішній оператор циклу (із індексною змінною `i`) відраховує повні перебори елементів масиву. Один такий перебір загалом означає розташування в "правильній" позиції одного значення.

До уваги

Якщо масив складається з n елементів, то для сортування всього масиву необхідно послідовно виконати $n - 1$ повний перебір елементів. Один повний перебір елементів спричинює визначення "правильного" місця для одного значення. Але справа в тому, що коли на другій позиції опиняється "правильне" значення, на першій позиції автоматично опиняється найменше значення. Тому кількість повних переборов на одиницю менша, ніж кількість елементів у масиві.

Крім того, варто врахувати, що під час повного перебору елементів немає сенсу переглядати ті значення, які вже знаходяться на "правильній" позиції. Тому під час кожного нового повного перебору переглядається на один елемент менше.

Внутрішній оператор циклу (із індексною змінною `j`) перебирає елементи масиву. Верхня межа для діапазону змін індексу `j` залежить від значення індексу `i` зовнішнього масиву. Тут враховано той факт, що немає сенсу перебирати уже відсортовані значення.

У тілі внутрішнього оператора циклу за допомогою умовного оператора порівнюються значення двох сусідніх елементів (умова `nums[j]>nums[j+1]`), і якщо значення елемента ліворуч більше за значення елемента праворуч, то елементи обмінюються значеннями.

Коли сортування масиву закінчено, за допомогою ще одного оператора циклу значення елементів уже відсортованого масиву відображаються в консольному вікні.

Цикл по колекції

У розглянутих раніше прикладах із масивами звертання до елементів масиву виконувалося через явне зазначення індексу елемента. Існує й інший спосіб перебору елементів масиву, в рамках якого доступ здійснюється безпосередньо до елемента масиву. Реалізується такий підхід за допомогою спеціальної версії оператора циклу `for`. Відповідну форму виконання циклів зазвичай називають *циклом по колекції*.

Цикл по колекції реалізують через `for`-оператор, який визначається в такому вигляді (жирним шрифтом виділені ключові елементи шаблона):

```
for(тип &змінна: масив){
    // Тіло оператора циклу
}
```

Після ключового слова `for` у круглих дужках вказують `тип` елемента масиву і назву `змінної`, яка ототожнює собою елемент масиву. Перед іменем змінної зазвичай ставлять інструкцію `&`. Після імені `змінної` через двокрапку вказують назву `масиву`, елементи якого перебираються. Команди, виконувані за кожний цикл, записують у фігурних дужках (тіло оператора циклу).

Оператор циклу працює так: `змінна`, вказана в `for`-інструкції, послідовно посилається на елементи `масиву`. Для кожного такого значення виконуються команди в тілі циклу, після чого посилання переходить на інший елемент масиву.

Теорія

Інструкція `&` потрібна для того, щоб змінна, вказана в `for`-інструкції, не просто переймала на себе значення елементів масиву, а була ***посиланням*** на елемент масиву. ***Посилання*** - це фактично альтернативна назва для елемента масиву, тому

через таке посилання можна не лише прочитати значення елемента масиву, а й присвоїти йому значення.

Далі розглянемо приклад, у якому створено числовий масив, і за допомогою циклу по колекції цей масив заповнено випадковими числами, значення елементів виведено в консольне вікно і підраховано кількість елементів у масиві. Розберемо програмний код, поданий у лістингу 2.10.

Лістинг 2.10. Цикл по колекції

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Створення масиву:
   int nums[12];
   cout<<"Масив випадкових чисел:\n";
   // Цикл по колекції:
   for(int &x: nums){
      x=rand()%10;   // Випадкове число від 0 до 9
      cout<<x<<" ";  // Відображення значення елемента
   }
   cout<<endl;
   // Змінна для запису розміру масиву:
   int length=0;
   // Цикл по колекції:
   for(int &x: nums){
      length++;
   }
```

```
    cout<<"Розмір масиву: "<<length<<endl;
    cout<<"Перевірка вмісту масиву:\n";
    // Звичайний оператор циклу:
    for(int k=0;k<length;k++){
       cout<<nums[k]<<" ";
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми (ураховуючи, що використовується генератор випадкових чисел) може бути таким:

Результат виконання програми (із лістингу 2.10)

```
Масив випадкових чисел:
5 6 8 5 4 0 0 1 5 6 7 9
Розмір масиву: 12
Перевірка вмісту масиву:
5 6 8 5 4 0 0 1 5 6 7 9
```

У програмному коді команда `int nums[12]` створює числовий масив. Заповнення масиву відбувається в операторі циклу, який описано for-інструкцією виду `for(int &x: nums)`. Вона означає, що перебір здійснюється по масиву `nums`, а через `x` позначено посилання на елемент цього масиву. У фігурних дужках вказано команду `x=rand()%10` (елементу масиву присвоюється випадкове число в діапазоні можливих значень від `0` до `9`) і команду `cout<<x<<" "`, якою відображається значення елемента.

Ще один приклад використання циклу по колекції пов'язаний із обчисленням розміру масиву.

Теорія

Розмір масиву в C++ зазвичай записують в окрему змінну (константу). Масив однозначно ідентифікується двома параметрами: точкою входу (адресою початкового елемента) і розміром масиву. Загалом визначити розмір масиву за одним лише його ім'ям проблематично.

У програмному коді оголошується цілочислова змінна `length` з початковим нульовим значенням. Потім запускається цикл по колекції, і в тілі цього оператора виконується команда `length++`. Таким чином, під час отримання посилання на кожен з елементів масиву `nums` значення змінної `length` збільшується на одиницю. Після завершення оператора циклу значення змінної `length` дорівнює кількості елементів у масиві `nums`.

Для перевірки вміст масиву `nums` виводиться в консольне вікно за допомогою звичайного `for`-оператора, в якому доступ до елементів здійснюється через індекс.

Генерування й перехоплення виключень

У C++ є механізм обробки *виключних ситуацій* (помилок), який, окрім свого безпосереднього призначення, може використовуватися для організації в програмі точок розгалуження. У такому разі послуговуються процедурою умисного генерування виключної ситуації.

Теорія

Під час обробки виключних ситуацій код, який може викликати помилку, поміщають у `try`-блок: такий блок помічають ключовим словом `try` і беруть у фігурні дужки. Після `try`-блока йде один або декілька `catch`-блоків. Кожний `catch`-блок призначений для обробки помилки певного типу. Якщо під час виконання коду в `try`-блоці виникає помилка, то виконання цього блока припиняється, а керування переходить до `catch`-блоку, призначеного для обробки помилки відповідного типу.

Для штучного генерування помилки використовують інструкцію `throw`, а після неї вказують об'єкт, змінну чи значення, які відіграють роль об'єкта помилки і передаються в `catch`-блок під час її обробки.

Якщо в програмі використовують обробку виключних ситуацій, то відповідний фрагмент програмного коду з одним блоком для перехоплення помилок може виглядати так (жирним шрифтом виділено ключові фрагменти коду):

```
try{
    // Код, який може викликати помилку
}
catch(тип_помилки объект_помилки){
    // Код, що виконується під час обробки помилки
}
```

Якщо під час виконання коду в блоці `try` помилок не було, то програмний код у `catch`-блоці (чи блоках - якщо їх декілька) ігнорується. Якщо у `try`-блоці виникла помилка, то починає виконуватися код у `catch`-блоці. Все це дуже сильно нагадує підхід, реалізований в умовному операторі, чим, власне, можна скористатися.

Розглянемо програму в лістингу 2.11. Там, за допомогою механізму штучного генерування виключної ситуації з наступною її обробкою, розв'язується задача про пошук кореня лінійного рівняння виду $Ax = B$ (відносно змінної x).

До уваги

Хоча формально рівняння $Ax = B$ дуже просте, проте воно має деякі особливості. По-перше, якщо $A \neq 0$, то розв'язок рівняння визначається формулою $x = \frac{B}{A}$. Якщо ж $A = 0$, і при цьому $B \neq 0$, то розв'язків у рівняння немає. І, нарешті, якщо $A = 0$ й одночасно $B = 0$, то розв'язком рівняння є будь-яке число.

У програмі для пошуку розв'язку рівняння $Ax = B$ із клавіатури вводять значення для параметрів A і B рівняння. Залежно від ситуації обчислюється розв'язок рівняння чи виводиться повідомлення про те, що розв'язком може бути будь-яке число або що розв'язків немає.

До уваги

Зрозуміло, що аналіз можливих ситуацій легко організувати за допомогою вкладених умовних операторів. Ми використовуємо в певному сенсі альтернативний підхід, який, однозначно, не найоптимальніший із точки зору організації програмного коду, проте показовий у плані використання механізму обробки виключних ситуацій для організації точок розгалуження.

Програмний код, який нас цікавить, виглядає так:

Лістинг 2.11. Генерування і перехоплення виключень

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   cout<<"Розв'язання рівняння Ax = B\n";
   // Параметри рівняння:
   double A,B;
   // Зчитування значень для параметрів рівняння:
   cout<<"A = ";
   cin>>A;
   cout<<"B = ";
   cin>>B;
   // Контрольований блок коду:
   try{
      if(A!=0){
         // Генерування виключення:
```

```
            throw A;
        }
        if(B!=0){
            // Генерування виключення:
            throw "Розв'язків немає";
        }
        cout<<"Розв'язок - будь-яке число"<<endl;
    }
    // Обробка числового виключення:
    catch(double e){
        cout<<"Розв'язок рівняння: "<<B/e<<endl;
    }
    // Обробка текстового виключення:
    catch(char* e){
        cout<<e<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Залежно від уведених значень, результати виконання програми різні. Нижче показано, як виглядає результат виконання програми, якщо для параметра *A* вводиться ненульове значення:

Результат виконання програми (із лістингу 2.11)

```
Розв'язання рівняння Ax = B
A = 2
B = 10
Розв'язок рівняння: 5
```

Якщо значення для параметра *A* нульове, а при цьому параметр *B* ненульовий, результат може бути таким:

Результат виконання програми (із лістингу 2.11)

```
Розв'язання рівняння Ax = B
A = 0
B = 3
Розв'язків немає
```

Якщо значення для обох параметрів нульові, результат такий:

Результат виконання програми (із лістингу 2.11)

```
Розв'язання рівняння Ax = B
A = 0
B = 0
Розв'язок - будь-яке число
```

Проаналізуємо програмний код, що приводить до таких результатів.

У програмі після зчитування значень змінних `A` і `B`, починається виконання `try`-блока. У цьому блоці в умовному операторі перевіряється умова `A!=0`, і якщо вона істинна, командою `throw A` генерується виключення. Значення змінної `A` передається в `catch`-блок, призначений для обробки помилки (обговорюється далі).

До уваги

Генерування виключення означає припинення виконання команд у `try`-блоці.

Якщо помилку не було згенеровано (що, вочевидь, відбувається при нульовому значенні змінної `A`), у гру вступає наступний умовний оператор, де перевіряється умова `B!=0`. Якщо умова істинна, командою `throw "Розв'язків немає"` генерується помилка з текстовим літералом, який передається в `catch`-блок для обробки помилки. Проте якщо і ця помилка не генерується, то виконується команда `cout<<"Розв'язок - будь-яке число"<<endl`. Оскільки генерування другої помилки можливе лише, якщо не згенерувалася перша, то відсутність згенерованих помилок означає, що обидві змінні `A` і `B` мають нульові значення. Іншими словами, команда

`cout<<"Розв'язок - будь-яке число"<<endl` виконується, лише якщо для змінних `A` і `B` з клавіатури було введено нульові значення.

Після `try`-блока описано два `catch`-блоки. У кожному з них у круглих дужках вказують:

- тип значення, яке передається в `catch`-блок для обробки помилки;
- позначення для змінної, в яку записується (або через яку передається) значення, передане в код обробки під час генеруванні помилки.

Під час генерування помилки в `try`-блоці для її обробки використовують лише один із двох `catch`-блоків. Рішення про те, який саме, приймається на основі типу значення, яке передається для обробки: тип значення, що передається, порівнюється з типом, вказаним у `catch`-блоці. Блок використовується для обробки помилки, якщо типи значень (переданого для обробки й зазначеного в `catch`-блоці) збігаються. В даному випадку є `catch`-блок для обробки `double`-значення, і `catch`-блок для обробки текстового значення (тип `char*`).

Подробиці



За замовчуванням, текст в C++ реалізується у вигляді символьного масиву - тобто масиву, що складається з символів. Символьний тип - це тип `char`. Символьний (і не лише) масив передається в функції і, зокрема, у `catch`-блок через вказівник на свій перший елемент. ***Вказівник*** - це змінна, значенням якої є адреса іншої змінної. Тому масив фактично передається через адресу свого першого елемента (який є значенням типу `char`). Щоб показати, що в `catch`-блок передається адреса `char`-елемента, а не сам `char`-елемент, в описі `catch`-блока в ідентифікаторі типу використовують зірочку `*`, а вся інструкція, що ідентифікує тип переданого в `catch`-блок значення, виглядає як `char*`.

У `catch`-блоці для обробки помилки з передачею `double`-значення перехоплюється перша помилка (яка генерується командою `throw A`). У цьому блоці під `e` мається на увазі передане в блок значення - тобто значення

змінної A. Тому під час обчислення значення виразу B/e насправді обчислюється значення виразу B/A.

У catch-блоці для обробки виключення з передачею char*-значення перехоплюється друга помилка (згенерована командою throw "Розв'язків немає"). У цьому блоці значенням змінної e є текст "Розв'язків немає" (насправді, значенням e є вказівник на літерал "Розв'язків немає", але в даному конкретному випадку це не так важливо). Тому під час виконання команди cout<<e<<endl у консольному вікні відображається саме зазначений текст.

Інструкція безумовного переходу

Як невелику ілюстрацію швидше гнучкості мови C++, ніж практичного боку програмування, розглянемо приклад, у якому задача про обчислення суми квадратів натуральних чисел розв'язується за допомогою *інструкції безумовного переходу* goto.

Теорія

Інструкція goto дозволяє передавати керування в певне місце програмного коду. Місце в програмному коді позначається ***міткою***, і цю мітку вказують після інструкції goto.

Мітка - звичайний ідентифікатор (назву вибирають довільно, з урахуванням наявних зарезервованих ключових слів), після якої ставлять двокрапку.

Використання інструкції goto в програмному коді багато програмістів вважає недоцільним.

Розглянемо програмний код, наведений у лістингу 2.12.

Лістинг 2.12. Інструкція безумовного переходу

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
```

```
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Цілочислові змінні:
    int n=10,s=0,k=1;
    start: // Мітка
    s+=k*k;
    if(k<n){
       k++;
       // Перехід до місця, позначеного міткою:
       goto start;
    }
    // Відображення результату обчислень:
    cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 2.12)

```
Сума квадратів чисел від 1 до 10: 385
```

У програмі оголошено змінні `n` зі значенням `10` (верхня межа суми), `s` зі значенням `0` (змінна для запису суми квадратів чисел), `k` зі значенням `1` (аналог індексної змінної).

Перед командою `s+=k*k` вказано мітку `start`. Під час першого виконання команди `s+=k*k` наявність мітки ніяк не проявляється. Після виконання команди в умовному операторі перевіряється умова `k<n`. Якщо умова істинна, командою `k++` значення змінної `k` збільшується на одиницю, а в результаті виконання інструкції `goto start` керування переходить до команди, позначеної міткою `start` - тобто знову виконується команда `s+=k*k`, після

чого виконується умовний оператор, і так далі. Якщо під час перевірки умови `k<n` в умовному операторі вона виявиться хибною, то інструкція `goto start` не виконується, а виконується команда `cout<<"Сума квадратів чисел від 1 до "<<n<<": "<<s<<endl` після умовного оператора.

Задачі для самостійного розв'язування

1. Напишіть програму для обчислення суми непарних натуральних чисел з використанням оператора `for`.

2. Напишіть програму для обчислення суми парних натуральних чисел з використанням оператора `do-while`.

3. Напишіть таку програму: користувач із клавіатури вводить ціле число, а програма відображає всі числа, на які це число ділиться без залишку.

4. Напишіть програму для обчислення найбільшого спільного дільника двох чисел.

5. Напишіть програму для обчислення значення натурального логарифма $\ln(1+x) \approx x - \frac{x^2}{2} + \frac{x^3}{3} - \frac{x^4}{4} + \cdots + \frac{(-1)^{n+1}x^n}{n}$.

6. Напишіть програму для обчислення значення синуса $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!}$.

7. Напишіть програму для обчислення значення косинуса $\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{(-1)^n x^{2n}}{(2n)!}$.

8. Напишіть програму для обчислення значення гіперболічного синуса $\mathrm{sh}(x) = \frac{\exp(x)-\exp(-x)}{2} \approx x + \frac{x^3}{3!} + \frac{x^5}{5!} + \frac{x^7}{7!} + \cdots + \frac{x^{2n+1}}{(2n+1)!}$.

9. Напишіть програму для обчислення значення гіперболічного косинуса $\mathrm{ch}(x) = \frac{\exp(x)+\exp(-x)}{2} \approx 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \frac{x^6}{6!} + \cdots + \frac{x^{2n}}{(2n)!}$.

10. Напишіть програму для обчислення значення виразу $\frac{\exp(x)-1}{x} \approx 1 + \frac{x}{2!} + \frac{x^2}{3!} + \frac{x^3}{4!} + \cdots + \frac{x^n}{(n+1)!}$.

11. Напишіть програму для розв'язання рівняння виду $Ax = B$, використовуючи вкладені умовні оператори.

12. Напишіть програму для розв'язання рівняння виду $Ax = B$ на множині цілих чисел: параметри A і B є цілими числами, і розв'язок x також повинен бути цілим числом (інакше вважаємо, що розв'язків нема).

13. Напишіть програму для знаходження (у межах першої сотні) таких цілочислових параметрів a, b і c, що виконується рівність $a^2 + b^2 = c^2$ (наприклад, $3^2 + 4^2 = 5^2$). Для обчислення квадратного кореня можна скористатися функцією `sqrt()` (доступна після підключення заголовка `<cmath>`).

14. Напишіть програму для розв'язання квадратного рівняння виду $Ax^2 + Bx + C = 0$ (запропонувати різні методи розв'язання).

15. Напишіть програму з функцією, яка для свого цілочислового аргументу повертає як результат суму дільників (сума чисел, на які число-аргумент ділиться без залишку).

16. Напишіть програму (із використанням оператора `switch`), у якій для введеного користувачем числа (у діапазоні від 1 до 10) перевіряється, чи є воно числом Фібоначчі (одне з чисел 1, 2, 3, 5 і 8). Запропонуйте альтернативні способи організації програмного коду.

17. Напишіть програму для створення масиву, заповнення його випадковими числами і сортування в порядку зменшення значень елементів.

18. Напишіть програму для створення числового масиву, заповнення його випадковими числами та зміни порядку елементів у масиві (щоб останній елемент став першим, передостанній - другим і так далі).

19. Напишіть програму для створення числового масиву та заповнення його випадковими числами. Елементи масиву повинні бути впорядковані таким чином: спочатку йдуть у порядку зростання непарні значення, а потім у

порядку спадання парні значення. Наприклад, якщо спочатку в масиві були елементи зі значеннями 7, 2, 5, 1, 8, 4, то після впорядкування треба отримати послідовність 1, 5, 7, 2, 4, 8.

20. Напишіть програму для створення числового масиву, заповнення його випадковими цілими числами. Елементи масиву повинні бути впорядковані таким чином: спочатку парні значення йдуть у порядку спадання, а потім у парні значення порядку зростання. Наприклад, якщо спочатку в масиві були елементи зі значеннями 7, 2, 5, 1, 8, 4, то після упорядкування треба отримати послідовність 8, 4, 2, 1, 5, 7.

Глава 3

Вказівники, масиви та посилання

Схаменіться, божевільні. Це вам все сниться!

з х/ф "Пропала грамота"

У цій главі розглядаються приклади й задачі, пов'язані в основному з використанням *вказівників* і масивів. Ми, крім того, дізнаємося:

- як створювати і де використовувати вказівники;
- що пов'язує масиви й вказівники;
- чим *посилання* відрізняється від вказівника;
- які особливості символьних масивів, і як за їхньою допомогою реалізується текст;
- як динамічно виділяється пам'ять під змінні й масиви;
- як створювати двомірний масив,

а також познайомимося з деякими іншими особливостями реалізації вказівників, посилань і масивів у C++.

Знайомство з вказівниками

Вказівник - змінна, значенням якої є адреса іншої змінної. Якщо звичайна змінна дозволяє отримати доступ до області пам'яті за іменем змінної, то вказівник дозволяє отримати доступ до області пам'яті за адресою. Ці два способи звернення до однієї й тієї ж області пам'яті можуть комбінуватися й використовуватися одночасно. У розглянутому далі прикладі створюються дві змінні, але доступ до них для присвоєння значення виконується за допомогою вказівників.

Теорія

Вказівник оголошується так: задається тип значення, на яке він може вказувати, а після ідентифікатора типу ставлять зірочку *. Далі традиційно йде ім'я вказівника.

Щоб отримати адресу змінної (звичайної, не вказівника), перед іменем змінної ставлять &. Щоб отримати значення змінної, адреса якої записана у вказівнику, перед іменем вказівника ставлять *.

У лістингу 3.1 наведено невеликий програмний код, у якому використовуються вказівники.

Лістинг 3.1. Знайомство з вказівниками

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Символьна змінна:
    char symb;
    // Цілочислова змінна:
    int num;
    // Вказівник на символьне значення:
    char* p;
    // Вказівник на цілочислове значення:
    int* q;
    // Значення вказівника p - адреса змінної symb:
    p=&symb;
    // Значення вказівника q - адреса змінної num:
    q=&num;
    // Значення змінній symb присвоюється
    // через вказівник на неї:
    *p='A';
    // Значення змінній num присвоюється
    // через вказівник на неї:
    *q=100;
    // Перевіряється значення змінної symb:
```

```
    cout<<"symb = "<<symb<<endl;
    // Перевіряється значення змінної num:
    cout<<"num = "<<num<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 3.1)

```
symb = A
num = 100
```

Командами `char symb` й `int num` у програмі відповідно оголошуються символьна змінна `symb` (значенням може бути символ - одиночний символ береться в одинарні лапки) і цілочислова змінна `num`.

Вказівник `p` на символьне значення оголошується командою `char* p`. Вказівник `p`, таким чином, може мати значенням адресу змінної типу `char`. Значення вказівнику `p` присвоюється командою `p=&symb`, так що в результаті значенням вказівника `p` є адреса змінної `symb`. Аналогічна ситуація з вказівником `q` на цілочислове значення. Вказівник оголошується командою `int* q`, а значення вказівнику присвоюється командою `q=&num`, у результаті чого значенням вказівника `q` є адреса змінної `num`.

До уваги

Значення вказівникам `p` і `q` присвоюються до того, як змінним `symb` і `num` були присвоєні значення. Простіше кажучи, щоб вказівнику присвоїти значенням адресу змінної, цій змінній зовсім необов'язково присвоювати значення. Достатньо, щоб змінна існувала (тобто була оголошена). Причина в тому, що вказівнику необхідна лише адреса змінної, яка в неї з'являється під час оголошення.

Якщо значенням вказівника є адреса певної області пам'яті, то будемо говорити, що вказівник посилається на цю область, або що його встановлено на цю область.

Значення змінним symb і num присвоюються через вказівники. Значення змінній symb присвоюється командою *p='A'. Значення змінній num присвоюється командою *q=100. Ми врахували, що для доступу до області пам'яті, на яку встановлено вказівник, перед іменем вказівника слід поставити зірочку *. Після присвоювання значень змінним symb і num ми перевіряємо значення змінних командами cout<<"symb = "<<symb<<endl і cout<<"num = "<<num<<endl, що містять явне звертання до змінних symb і num.

Масиви та вказівники

Наступний приклад ілюструє зв'язок між масивами та вказівниками.

Теорія

Є декілька важливих фактів стосовно масивів (поки що ми говоримо про одномірні статичні масиви) і вказівників. Наприклад:

- Під час оголошення масиву місце під його елементи в пам'яті виділяється так, що елементи розташовані один за одним.
- Ім'я масиву є вказівником на його перший елемент.
- До вказівників можна додавати цілі числа та віднімати від них цілі числа. Результатом є вказівник, зміщений від поточного вказівника на відповідну кількість комірок пам'яті (у напрямку збільшення чи зменшення адреси).
- Різницею двох вказівників (одного типу) є ціле число, яке визначає кількість позицій (комірок) між комірками пам'яті, на які посилаються вказівники.
- Вказівники можна індексувати (індекс вказується у квадратних дужках після імені вказівника й формально може бути від'ємним). Значення індексованого вказівника - це значення в комірці, яка знаходиться від цього вказівника на кількість позицій, визначену індексом.

Резюме досить просте: звертання до елемента масиву є операцією індексування вказівника.

У лістингу 3.2 наведено програму, в якій створюється і заповнюється значеннями символьний масив. При цьому значною мірою використовуються вказівники.

Лістинг 3.2. Масиви та вказівники

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Розмір масиву:
    const int size=12;
    // Індексна змінна:
    int k;
    // Створення символьного масиву:
    char symbs[size];
    // Вказівники на символьні значення:
    char* p;
    char* q;
    // Вказівник на перший елемент масиву:
    p=symbs;
    // Значення першого елементу масиву:
    p[0]='A';
    // Вказівник на останній елемент масиву:
    q=&symbs[size-1];
    // Кількість позицій між першим
    // і останнім елементами масиву:
    cout<<"Між першим і останнім елементами "<<q-p
<<" позицій\n";
    // Заповнення масиву значеннями:
    while(p!=q){
        // Вказівник на наступний елемент:
        p++;
        // Обчислення значення елемента масиву:
```

```
        *p=p[-1]+1;
    }
    cout<<"Елементи масиву\n| ";
    // Відображення вмісту масиву:
    for(k=0;k<size;k++){
        cout<<symbs[k]<<" | ";
    }
    cout<<"\nЕлементи у зворотному порядку\n| ";
    // Відображення вмісту масиву у зворотному порядку:
    for(k=0;k<size;k++){
        cout<<q[-k]<<" | ";

    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 3.2)

```
Між першим і останнім елементами 11 позицій
Елементи масиву
| A | B | C | D | E | F | G | H | I | J | K | L |
Елементи у зворотному порядку
| L | K | J | I | H | G | F | E | D | C | B | A |
```

У програмі оголошено цілочислову константу `size`, яка визначає розмір символьного масиву, що створюється командою `char symbs[size]`. Також у програмі оголошено два символьних вказівники `p` і `q` (команди `char* p` і `char* q`). Вказівнику `p` значення присвоюється командою `p=symbs`. Оскільки ім'я масиву є посиланням на перший елемент масиву, то після

виконання цієї команди вказівник p посилається на перший елемент масиву symbs. Командою p[0]='A' першому елементу масиву symbs присвоюється значення 'A'. Так відбувається унаслідок правила індексування вказівників: p[0] являє собою значення області пам'яті, яка зміщена на 0 позицій відносно комірки, на яку посилається вказівник p. Оскільки на момент виконання програми вказівник p має значенням адресу першого елемента масиву, то саме цей елемент отримує значення 'A'.

Значення вказівнику q присвоюється командою q=&symbs[size-1]. Оскільки symbs[size-1] є ні чим іншим, як останнім елементом масиву symbs, то &symbs[size-1]- адреса останнього елемента масиву. Отже, значенням вказівника q є адреса останнього елемента масиву. Якщо обчислити різницю вказівників q-p, то результатом буде кількість позицій між останнім і першим елементами масиву. Це значення на одиницю менше за довжину масиву (і в цьому конкретному випадку дорівнює 11, оскільки в масиві 12 елементів).

Щоб заповнити масив значеннями запускається оператор циклу while. В операторі перевіряється умова p!=q, яка полягає в тому, що значення вказівників p і q різні. У тілі оператора циклу командою p++ операцію інкременту застосовано до вказівника p. Ця команда є еквівалентною команді p=p+1. Додавання до вказівника одиниці дає в результаті вказівник, який встановлено на сусідню комірку. Оскільки в масиві елементи в пам'яті розташовані підряд один за другим, то після виконання команди p++ вказівник p буде посилатися на наступний елемент у масиві. Командою *p=p[-1]+1 обчислюється значення елемента масиву. У лівій частині команди стоїть вираз *p, що є елементом масиву, на який встановлено вказівник p. Вираз p[-1]+1 у правій частині містить інструкцію p[-1]. Це значення елемента, який знаходиться на одну позицію назад (у напрямку зменшення адреси) відносно елемента, на який встановлено вказівник p. Простіше кажучи, щоб визначити значення елемента, на який встановлено вказівник p, береться значення попереднього елемента і додається одиниця.

До уваги

Варто зауважити, що значеннями елементів масиву `symbs` є символи. Операція додавання до символу числа обробляється так. Береться код символу в кодовій таблиці й до нього додається число. Якщо результат необхідно інтерпретувати як символ, то отримане числове значення стає кодом у кодовій таблиці, за яким і повертається символ. Як наслідок додавання одиниці до символу дає результатом наступний символ у кодовій таблиці (якщо йдеться про букви, то отримуємо наступну букву за алфавітом).

Також варто нагадати, що інструкція `\n` є командою переходу до нового рядка під час відображення тексту.

Таким чином, кожного разу під час виконання одного циклу вказівник `p` зміщується праворуч на одну позицію, і це триватиме до тих пір, поки вказівник `p` не дорівнюватиме вказівнику `q` на останній елемент масиву.

Для відображення вмісту масиву (після його заповнення) використовуємо звичайний цикл, у якому звертання до елементів масиву виконується за допомогою індексу. Ще в одному операторі циклу значення елементів масиву відображаються у зворотному порядку: з кінця масиву на початок. У цьому операторі циклу індексна змінна `k` пробігає значення від `0` до `size-1`, а звертання до елементів масиву виконується у вигляді `q[-k]`. Нагадаємо, що `q` - вказівник на останній елемент масиву, і тоді `q[-k]` являє собою значення елемента масиву, котрий стоїть на `k` позицій ліворуч від останнього елемента. У результаті отримуємо перебір елементів масиву в зворотному порядку.

Знайомство з посиланнями

Для ілюстрації особливостей використання *посилань*, розглянемо зовсім невеличкий приклад, який подано в лістингу 3.3.

Лістинг 3.3. Знайомство з посиланнями

```
#include <iostream>
#include <cstdlib>
```

```
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Цілочислова змінна:
    int num;
    // Посилання на змінну:
    int &ref=num;
    // Присвоювання значення змінній:
    num=100;
    // Перевірка значення змінної та посилання:
    cout<<"num = "<<num<<endl;
    cout<<"ref = "<<ref<<endl;
    // Присвоювання значення посиланню:
    ref=200;
    // Перевірка значення змінної та посилання:
    cout<<"num = "<<num<<endl;
    cout<<"ref = "<<ref<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У програмі оголошено цілочислову змінну `num`, а також ще одну змінну, яка називається `ref`. Оголошення й ініціалізація цієї змінної (команда `int &ref=num`) має деякі особливості. А саме:

- Під час оголошення змінної `ref` перед іменем змінної вказано амперсанд `&`.
- Змінній `ref` присвоюється значенням змінна `num`, причому самій змінній `num` значення ще не присвоєно.

Дві згадані особливості характерні для оголошення *посилання*.

Теорія

Посилання є змінною, яка посилається (дозволяє отримати) доступ на ту ж область пам'яті, що і деяка інша змінна. З певним спрощенням посилання можна вважати синонімом певної змінної.

Річ у тім, що під час оголошення звичайної змінної для неї в пам'яті виділяється місце. Під час створення посилання пам'ять під посилання не виділяється, а використовується область пам'яті, виділена під іншу змінну. У результаті отримуємо дві змінні, які мають доступ до однієї й тієї ж області пам'яті.

Змінна, область пам'яті якої використовується посиланням, присвоюється посиланню під час оголошення. Перед іменем посилання під час оголошення вказують символ `&`. Посилання створюється для однієї змінної і не може бути "перекинуте" на іншу змінну.

У програмі виконуються такі операції зі змінною `num` і посиланням `ref`:

- присвоюється значення змінній `num`, після чого перевіряються значення змінної `num` і посилання `ref`;
- присвоюється значення посиланню `ref` і знову перевіряються значення змінної `num` і посилання `ref`.

Результат виконання програми такий:

Результат виконання програми (із лістингу 3.3)

```
num = 100
ref = 100
num = 200
ref = 200
```

Бачимо, що зміна значення змінної `num` спричинює зміну значення посилання `ref`, а зміна значення посилання `ref` спричинює зміну значення змінної `num`. Це цілком логічно, якщо згадати, що і змінна `num`, і посилання `ref` мають справу з однією й тією ж областю пам'яті.

Динамічне виділення пам'яті

Масиви, які ми розглядали до цього, були *статичними* - пам'ять під них виділялася в процесі компіляції програми. Існують *динамічні* масиви, пам'ять під які виділяється в процесі виконання програми.

Теорія

Динамічне виділення пам'яті виконується за допомогою оператора `new`. Після оператора `new` вказують ідентифікатор типу, який визначає тип змінної, для якої виділяється пам'ять. Якщо пам'ять виділяється не під окрему змінну, а під масив, то після ідентифікатора типу в квадратних дужках вказується розмір динамічного масиву.

Результатом інструкції з виділення динамічної пам'яті повертається вказівник на виділену область пам'яті (для окремої змінної) або вказівник на перший елемент масиву (якщо створюється масив).

Після того, як у динамічно виділеній пам'яті потреба відпала, її звільняють за допомогою оператора `delete`. Після оператора `delete` розташовується вказівник на область пам'яті, яка звільняється. Якщо видаляється динамічний масив, то між оператором `delete` і вказівником на перший елемент масиву, що видаляється, розташовуються пусті квадратні дужки `[]`.

У лістингу 3.4 наведено програму, в якій створюється динамічний масив, він заповнюється значеннями, після чого видаляється.

Лістинг 3.4. Динамічний масив

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Вказівник на ціле число:
    int* size;
    // Динамічне виділення пам'яті під змінну:
```

```
    size=new int;
    cout<<"Вкажіть розмір масиву: ";
    // Зчитування значення з клавіатури:
    cin>>*size;
    // Вказівник на символьне значення:
    char* symbs;
    // Створення символьного масиву:
    symbs=new char[*size];
    // Заповнення масиву:
    for(int k=0;k<*size;k++){
       symbs[k]='a'+k;
       cout<<symbs[k]<<" ";
    }
    // Видалення масиву:
    delete [] symbs;
    // Видалення змінної:
    delete size;
    cout<<"\nМасив і змінну видалено\n";
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Можливий результат виконання програми наведено нижче (жирним шрифтом виділено значення, яке вводить користувач):

Результат виконання програми (із лістингу 3.4)

```
Вкажіть розмір масиву: 12
a b c d e f g h i j k l
Масив і змінну видалено
```

У програмі створюються два вказівники: вказівник `size` на цілочислове значення і вказівник `symbs` на символьне значення. Значення вказівнику `size`

присвоюється командою `size=new int`. У результаті виконання цієї команди в пам'яті динамічно виділяється місце під цілочислову змінну, а адреса виділеної області пам'яті записується як значення у вказівник `size`. Тут варто ще раз підкреслити, що `size` є вказівником. Після динамічного виділення пам'яті цей вказівник отримує значення. Але область пам'яті, на яку він посилається, значення не отримує. У програмі воно зчитується з клавіатури, для чого використовується команда `cin>>*size`. Оскільки зчитане значення заноситься не у вказівник `size`, а в область пам'яті, на яку посилається вказівник, то праворуч від оператора введення `>>` використовуємо вираз `*size`.

Динамічний символьний масив створюється командою `symbs=new char[*size]`. У квадратних дужках вказано вираз `*size`, який визначає розмір масиву і є зчитаним із клавіатури значенням динамічно створеної змінної.

До уваги

Ми називаємо динамічно створеною змінною область пам'яті, виділену в динамічному режимі. Це не зовсім коректний термін, але він достатньо зручний, тому ми будемо ним послуговуватися.

Також варта уваги така обставина:якщо під час створення статичного масиву розмір масиву повинен задаватися константою, то під час створення динамічного масиву розмір може задаватися за допомогою змінної (не обов'язково динамічної).

У результаті виконання команди `symbs=new char[*size]` створюється масив розміру `*size`, а адреса цього масиву (адреса його першого елемента) записується у вказівник `symbs`. У статичному масиві ім'я масиву є вказівником на перший елемент масиву. У даному випадку ідентифікатор `symbs` також є вказівником на перший елемент масиву. Тому ми цілком можемо ототожнювати `symbs` з іменем динамічного масиву. Ми це власне і робимо, коли за допомогою оператора циклу заповнюємо масив значеннями і

відображаємо значення елементів масиву в консольному вікні. Значення елементу масиву присвоюється командою `symbs[k]='a'+k`. У правій частині вираз `'a'+k` обчислюється так: до коду символу `'a'` додається значення `k`, і отримане число стає кодом, за яким визначається символ, що є результатом виразу. Як підсумок, масив `symbs` заповнюється послідовністю букв, починаючи з `'a'`.

Для видалення динамічного масиву `symbs` використовуємо команду `delete [] symbs`, а динамічну змінну видаляємо командою `delete size`.

До уваги

Змінні-вказівники `size` і `symbs` є статичними і нікуди не видаляються. Під видаленням масиву `symbs` мається на увазі видалення того масиву, на який посилається вказівник `symbs`. Про видалення динамічної змінної ми говоримо в тому сенсі, що звільнюється область пам'яті, на яку посилається вказівник `size`.

Особливості символьних масивів

Масиви, які складаються з символів (символьні масиви) мають певні особливості, пов'язані з тим, що символьний масив - основний спосіб реалізації текстових значень. Про це наступний приклад.

Теорія

Головна особливість символьного масиву, що містить текст - це те, що розмір масиву загалом не збігається з розміром тексту, який у нього записується (масив повинен бути достатньо великий, щоб він міг містити текст різної довжини). Як індикатор закінчення тексту в символьному масиві використовують нуль-символ `\0`, у якого нульовий код.

Оператор виведення `<<`, якщо праворуч від нього вказано ім'я символьного масиву, виконує виведення всіх елементів символьного масиву, аж до нуль-символу.

У лістингу 3.5 наведено програму, в якій виконуються певні маніпуляції з текстовими значеннями. Основною для реалізації тексту є символьний масив.

Теорія

У C++ існує два способи реалізації тексту: у вигляді символьного масиву і через об'єкт класу `string`. Однак базовим є спосіб реалізації тексту за допомогою символьного масиву - у всякому разі, текстові літерали реалізуються саме так.

Розглянемо наведений нижче програмний код:

Лістинг 3.5. Особливості символьних масивів

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Символьний масив:
    char str[100]="Програмуємо на C++";
    // Відображення вмісту символьного масиву:
    cout<<str<<endl;
    // Поелементне відображення вмісту символьного масиву:
    for(int k=0;str[k];k++){
        cout<<str[k]<<"_";
    }
    cout<<endl;
    // Відображення вмісту, починаючи з певної позиції:
    for(char* p=str;*p;p++){
        cout<<p<<endl;
    }
    // Зміна символу в масиві:
    str[11]='\0';
    // Відображення вмісту масиву:
    cout<<str<<endl;
    // Відображення, починаючи із вказаної позиції:
    cout<<str+12<<endl;
```

```
    // Відображення тексту, до якого додається число:
    cout<<"Один два три"+5<<endl;
    // Вказівник на символьне значення:
    const char* q="Один два три"+9;
    // Значення символу, на який посилається вказівник:
    cout<<q[0]<<endl;
    // Відображення значення вказівника:
    cout<<q<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 3.5)

```
Програмуємо на С++
П_р_о_г_р_а_м_у_є_м_о_ _н_а_ _С_+_+_
Програмуємо на С++
рограмуємо на С++
ограмуємо на С++
грамуємо на С++
рамуємо на С++
амуємо на С++
муємо на С++
уємо на С++
ємо на С++
мо на С++
о на С++
 на С++
на С++
а С++
 С++
```

```
C++
++
+
Програмуємо
на C++
два три
т
три
```

Обговоримо основні та найбільш цікаві фрагменти програми. Отже, командою `char str[100]="Програмуємо на C++"` створюється символьний масив `str`, і в цей масив автоматично записується (посимвольно) текст "Програмуємо на C++". Масив складається зі 100 елементів. Текст, який записується в масив, явно містить набагато менше букв. Тому частина елементів масиву лишаються невикористаними. Щоб під час обробки символьного масиву було зрозуміло, де закінчується текст і починаються елементи, які "не використовуються", в масив автоматично додається нуль-символ `'\0'`.

До уваги

Текст складається з 18 символів. Під час їх запису в масив `str` перші 18 елементів масиву будуть заповнені символами з тексту, а 19-ий елемент масиву автоматично отримує значенням нуль-символ `'\0'`.

Як зазначалося раніше, під час реалізації тексту через символьний масив розмір масиву і розмір тексту відрізняються. Для обчислення довжини тексту, записаного в масив, використовують функцію `strlen()`. Як аргумент функції передається ім'я символьного масиву з текстом, а як результат повертається кількість символів у тексті, записаному в масив.

Для відображення тексту з масиву `str` у консольному вікні використовуємо команду `cout<<str<<endl`. Формально в цьому випадку відображається значення вказівника (оскільки ім'я масиву є вказівником на перший елемент масиву). Якби йшлося, наприклад, про числовий масив, то було б відображено

числовий код адреси, яка є значенням вказівника. Проте символьні масиви (а якщо точніше, то вказівники на символьні значення) обробляються особливим чином. Так, під час спроби вивести в консольне вікно за допомогою оператора виведення значення вказівника на символ насправді відображається символ із комірки, на яку встановлено вказівник, а також символи з наступних комірок. Процес триває доти, поки в черговій комірці не буде прочитано нуль-символ. Підсумок такий: щоб відобразити вміст символьного масиву в консолі, слід праворуч від оператора виведення вказати ім'я масиву.

Разом із тим, ніхто нам не забороняє обробляти й відображати вміст символьного масиву поелементно. Ситуацію ілюструє оператор циклу, в якому текст із масиву `str` відображено символ за символом, при цьому після кожного символу додається підкреслення.

Подробиці



Як умову в операторі циклу вказано вираз `str[k]`, що є символом, записаним у масив `str` під індексом `k`. Тут ми скористалися тим, що відмінні від нуля числові значення інтерпретуються як значення `true`, а нульові значення інтерпретуються як `false`. Якщо йдеться про символ, то числове значення - це код символу. Єдиний символ, який має нульовий код - це нуль-символ. Тому, якщо вираз виду `str[k]` вказано як умову, то така умова є еквівалентною `false` тільки якщо `str[k]` є нуль-символом, і еквівалентною `true` в усіх інших випадках.

Додатковий приклад "нестандартного" відображення тексту наведено ще в одному операторі циклу. У цьому `for`-операторі в першому блоці інструкцій оголошено вказівник `p` на символьне значення, якому як початкове значення присвоюється ім'я масиву `str` - тобто на початку виконання оператора вказівник `p` містить адресу першого елемента в масиві `str`. За кожен цикл командою `p++` збільшується значення вказівника `p` таким чином, що цей вказівник "перекидається" на наступний (за поточним) елемент масиву. Умовою, що перевіряється в операторі циклу, є вираз `*p`. Це символ, адресу

якого записано у вказівник p. Значення умови дорівнює false, якщо вказівник містить адресу елемента масиву з нуль-символом. У тілі оператора циклу виконується команда cout<<p<<endl. Оброблюється вона так: відображаються всі символи в тексті, починаючи з того місця, куди вказує p, і до кінця текстового значення. Оскільки вказівник p поступово переміщується від початку тексту в його кінець, то кожне нове повідомлення в консольному вікні буде коротшим за попереднє на одну початкову букву.

Наступна частина експерименту з текстовим масивом str полягає в тому, що командою str[11]='\0' замість пробілу в текст вставляється нуль-символ. Якщо тепер скористатися командою cout<<str<<endl, то буде відображено текст до першого нуль-символу. Тепер це елемент з індексом 11. Якщо ми скористаємося командою cout<<str+12<<endl, то в результаті буде відображатися текст, починаючи із символу з індексом 12 (індексація елементів у символьному масиві, як і в інших масивах, починається з нуля).

Декілька наступних команд ілюструють специфіку реалізації текстових літералів.

Теорія

Текстові літерали в пам'яті реалізуються у вигляді символьних масивів. Сам літерал насправді передається у вирази і обробляється через неявний вказівник на свій перший символ.

У результаті виконання команди cout<<"Один два три"+5<<endl відображається текст "Один два три", починаючи із символу з індексом 5. Причина в тому, що текстовий літерал "Один два три" в операції додавання до числа оброблюється як вказівник на перший символ у тексті. Після додавання до вказівника числа 5 отримуємо вказівник, зміщений відносно першого символу на 5 позиції.

Схожа ситуація має місце під час виконання команди const char* q="Один два три"+9. У результаті її виконання вказівник q міститиме адресу символу з індексом 9 у тексті "Один два три".

До уваги

Вказівник, який повертається для текстового літерала, є константою (що цілком логічно). Якщо ми цей вказівник присвоюємо як значення змінній (вказівнику), то цей вказівник повинен бути оголошений як константа, а якщо ні, то перед виразом у правій частині ставиться інструкція `(char*)` для виконання явного приведення типу.

Під час виконання команди `cout<<q[0]<<endl` відображається символ, адресу якого записано у вказівнику `q`. А якщо скористатися командою `cout<<q<<endl`, то буде відображено текст від поточного символу до кінця рядка.

Двомірні масиви

У двомірному масиві доступ до елемента виконується за допомогою двох індексів. У лістингу 3.6 наведено програму, де створюється двомірний масив. Масив заповнюється випадковими символами (буквами) і його вміст виводиться в консольне вікно.

Теорія

Під час створення двомірного масиву вказується тип його елементів, назва масиву і розмір по кожному з індексів. Для кожного індексу використовуються окремі квадратні дужки.

Двомірний масив зручно уявляти в вигляді таблиці чи матриці. Розмір по першому індексу визначає кількість рядків у матриці, а розмір по другому індексу визначає кількість стовпців.

Щоб звернутись до елемента двомірного масиву вказують ім'я масиву і пару індексів. Індексація (по кожному з двох індексів) починається з нуля.

Розглянемо наведений нижче програмний код:

Лістинг 3.6. Двомірний масив із випадкових букв

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

```
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Ініціалізація генератора випадкових чисел:
    srand(2);
    // Кількість "стовпців" у масиві:
    const int width=9;
    // Кількість "рядків" у масиві:
    const int height=5;
    // Створення двомірного масиву:
    char Lts[height][width];
    // Заповнення двомірного масиву:
    for(int i=0;i<height;i++){
        for(int j=0;j<width;j++){
            // Випадкова буква від 'A' до 'Z':
            Lts[i][j]='A'+rand()%26;
            // Відображення значення елемента масиву:
            cout<<Lts[i][j]<<" ";
        }
        // Перехід до нового рядка:
        cout<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми (із поправкою на використання генератора випадкових чисел) може бути таким:

Результат виконання програми (із лістингу 3.6)

```
T S S L A U E P T
M T F Q I R T R Z
```

```
C X P D P R X O I
G T Y Y H Y R V Y
H K I T C U Q C F
```

Щодо безпосередньо програмного коду, то там оголошують дві цілочислові константи `width` і `height`, які визначають розміри масиву. Сам масив створюється командою `char Lts[height][width]`. Для заповнення масиву і виведення значень його елементів у консоль використані вкладені оператори циклу. Індексна змінна `i` у зовнішньому операторі циклу перебирає рядки масиву, а індексна змінна `j` у внутрішньому операторі циклу перебирає стовпці масиву. Значення елементам масиву присвоюється командою `Lts[i][j]='A'+rand()%26`. Значення елемента масиву отримуємо додаванням до коду символу `'A'` цілого випадкового числа в діапазоні значень від `0` до `25`, і за отриманим таким чином кодом відновлюється символ. Відображається значення елемента двомірного масиву за допомогою команди `cout<<Lts[i][j]<<" "`. За роздільник між символами в одному рядку використано пробіл. Після закінчення відображення рядка командою `cout<<endl` виконується перехід до нового рядка.

Як ще одну ілюстрацію розглянемо задачу про обчислення добутку двох квадратних матриць.

Подробиці



Квадратною називається матриця, у якої кількість рядків збігається з кількістю стовпців.

Якщо матриця A складається з елементів a_{ij}, а матриця B складається з елементів b_{ij} (індекси $i,j = 1,2,\dots,n$), то елементи c_{ij} матриці $C = AB$, що дорівнює добутку матриць A і B, визначається співвідношенням $c_{ij} = \sum_{k=1}^{n} a_{ik}b_{kj}$. Саме цю формулу ми використовуємо для обчислення значень елементів матриці-добутку.

У лістингу 3.7 наведено програму, у якій визначаються дві квадратні матриці, і обчислюється їх добуток. Вихідні матриці і матриця-результат відображаються в консольному вікні.

До уваги

У програмі для виведення інформації в консольне вікно замість оператора виведення використовується функція `printf()`. Для цього ми підключаємо в програмі заголовок `<cstdio>`. Навпаки, оскільки оператор виведення не використовується, заголовок `<iostream>` можна не підключати.

Функція `printf()` зручна тим, що дозволяє виконувати форматоване виведення у консольне вікно. Ця особливість функції нам знадобиться при відображенні вмісту двомірних масивів.

Розглянемо поданий нижче програмний код:

Лістинг 3.7. Добуток матриць

```
#include <cstdlib>
#include <cstdio>
using namespace std;
// Глобальна константа:
const int n=3;
// Функція для відображення елементів двомірного масиву:
void show(int M[n][n]){
   for(int i=0;i<n;i++){
      for(int j=0;j<n;j++){
         // Відображення елемента масиву:
         printf("%4d",M[i][j]);
      }
      // Перехід до нового рядка:
      printf("\n");
   }
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
```

```
    // Оголошення й ініціалізація двомірного масиву:
    int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}};
    printf("Матриця A:\n");
    // Відображення вмісту масиву:
    show(A);
    // Оголошення й ініціалізація двомірного масиву:
    int B[n][n]={{2,1,-1},{1,3,1},{-2,1,4}};
    printf("Матриця B:\n");
    // Відображення вмісту масиву:
    show(B);
    // Оголошення двомірного масиву:
    int C[n][n];
    // Обчислення добутку матриць:
    for(int i=0;i<n;i++){
       for(int j=0;j<n;j++){
          C[i][j]=0; // Початкове значення елемента
          for(int k=0;k<n;k++){
             C[i][j]+=A[i][k]*B[k][j];
          }
       }
    }
    printf("Матриця C=A*B:\n");
    // Відображення вмісту масиву:
    show(C);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 3.7)

```
Матриця A:
```

```
    1  -2   1
    2   0  -1
    2   3  -1
Матриця B:
    2   1  -1
    1   3   1
   -2   1   4
Матриця C=A*B:
   -2  -4   1
    6   1  -6
    9  10  -3
```

Цей програмний код має деякі особливості. А саме:

- У програмі оголошується глобальна цілочислова константа `n`, яка визначає розмір двомірних масивів.
- Для відображення вмісту двомірних масивів (через які реалізуються матриці) описано функцію `show()`.
- Для виведення повідомлень у консольне вікно використовується функція `printf()` (і, відповідно, не використовується оператор виведення).

Цілочислова константа `n` оголошується звичайним чином, проте інструкція з оголошенням константи знаходиться не в тілі головної функції програми, а на початку коду, до оголошення функції `show()` і функції `main()`. Це необхідно, бо і у функції `show()`, і у функції `main()` використовується константа `n`.

Теорія

Область доступності змінних (і констант) обмежується блоком, у якому їх оголошено. Якщо оголосити константу `n` у функції `main()`, то доступна вона буде лише у функції `main()`.

Функція `show()` призначена для відображення вмісту двомірного масиву. Вона не повертає результат, тому як ідентифікатор типа результату функції вказано ключове слово `void`. Функцію описано з одним аргументом, заданим

інструкцією `int M[n][n]`, яка передбачає, що йдеться про двомірний цілочисловий масив. У тілі функції запускаються вкладені оператори циклу. У зовнішньому операторі циклу індексна змінна `i` перебирає рядки масиву, а індексна змінна `j` у внутрішньому операторі циклу перебирає стовпці масиву. У внутрішньому операторі циклу командою `printf("%4d",M[i][j])` відображається значення елемента з індексами `i` та `j` двомірного масиву `M`. Після закінчення внутрішнього оператора циклу для переходу до нового рядка під час відображення елементів виконується команда `printf("\n")`.

Подробиці



За допомогою функції `printf()` здійснюється форматоване виведення у консольне вікно. Якщо аргументом функції передати текстове значення, то це текстове значення буде відображено в консолі. Приклад такої ситуації дає команда `printf("\n")`, яка "друкує" інструкцію переходу до нового рядка `\n` - іншими словами, виконується перехід до нового рядка. Проте це поодинокий випадок ситуації, коли в консолі відображається текст. Приклади виклику функції `printf()` з одним текстовим аргументом зустрічаються у функції `main()` (наприклад, команда `printf("Матриця A:\n")`).

Якщо йдеться про відображення, наприклад, числового значення, то це числове значення передається другим аргументом функції `printf()`. Першим аргументом функції в такому випадку передається спеціальний текстовий рядок, який, фактично, визначає формат відображення даних. Прикладом такої ситуації є команда `printf("%4d",M[i][j])`. Тут відображуване числове значення `M[i][j]` передається другим аргументом функції `printf()`. Першим аргументом функції передається текстовий рядок форматування `"%4d"`. У цьому рядку символ `d` означає, що відображається ціле число, а цифра `4` визначає кількість позицій, які виділяються для відображення числа (навіть якщо число складається з однієї цифри, під нього виділяється чотири позиції).

У головній функції програми командами `int A[n][n]={{1,-2,1},{2,0,-1},{2,3,-1}}` та `int B[n][n]={{2,1,-1},{1,3,1},{-`

2,1,4}} оголошуються та ініціалізуються два цілочислових двомірних масиви A і B. Значення, що присвоюються елементам масиву, задаються через оператор присвоювання, у фігурних дужках. Для відображення вмісту масивів використано команди show(A) і show(B) відповідно.

Двомірний масив C тільки оголошується (не ініціалізується), для чого задіяно команду int C[n][n]. Заповнюється цей масив значеннями в процесі обчислення добутку матриць. Із цією метою запускаються вкладені оператори циклу. У зовнішньому операторі циклу індексна змінна i визначає перший індекс обчислюваного елемента. У внутрішньому операторі циклу індексна змінна j визначає другий індекс обчислюваного елементу. При фіксованих значеннях індексів i і j обчислення виконуються таким чином. Спочатку командою C[i][j]=0 обчислюваному елементу масиву присвоюється нульове значення. Після цього запускається оператор циклу з індексною змінною k, і для кожного її значення з діапазону від 0 до n-1 за допомогою команди C[i][j]+=A[i][k]*B[k][j] до поточного значення елемента C[i][j] додається добуток A[i][k]*B[k][j]. Тим самим реалізується формула для обчислення елементів матриці-добутку.

Після закінчення обчислень командою show(C) виводиться вміст двомірного масиву C у консольне вікно.

Масиви вказівників

Елементами масиву, окрім іншого, можуть бути вказівники. Простіше кажучи, ніхто не забороняє нам створити масив із вказівників. Уявімо, що створюється масив із вказівників на цілочислові значення. Елементи такого масиву є вказівниками. Ім'я цього масиву є *вказівником на вказівник* на ціле число.

Теорія

Щоб оголосити вказівник на вказівник, у команді оголошення використовують дві зірочки **. Наприклад, інструкція int** p оголошує вказівник p, значенням якого може бути адреса змінної, яка сама є вказівником на цілочислове значення.

Далі діємо таким чином: створюємо динамічні числові масиви, й адреси цих масивів (їхніх перших елементів) присвоюємо значеннями елементам масиву вказівників. Утворюється специфічна конструкція, яка за всіма зовнішніми ознаками нагадує двомірний масив, тільки в такому масиві в різних рядках може міститися різна кількість елементів. Означений підхід ілюструє програма, наведена в лістингу 3.8.

Лістинг 3.8. Масив числових вказівників

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Індексні змінні:
   int i,j;
   // Розмір масиву вказівників:
   const int size=5;
   // Масив зі значеннями, що визначають розміри
   // числових масивів:
   const int cols[size]={3,7,6,4,2};
   // Динамічний масив вказівників:
   int** nums=new int*[size];
   // Створення динамічних числових масивів
   // і заповнення їх значеннями:
   for(i=0;i<size;i++){
      // Динамічний числовий масив:
      nums[i]=new int[cols[i]];
      cout<<"| ";
      // Заповнення числового масиву:
```

```
        for(j=0;j<cols[i];j++){
            // Випадкове число від 0 до 9:
            nums[i][j]=rand()%10;
            // Відображення значення елемента масиву:
            cout<<nums[i][j]<<" | ";
        }
        cout<<endl;
    }
    // Видалення числових динамічних масивів:
    for(i=0;i<size;i++){
        // Видалення динамічного числового масиву:
        delete [] nums[i];
    }
    // Видалення динамічного масиву вказівників:
    delete [] nums;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У програмі, окрім іншого, створюється цілочислова константа `size`, яка визначає масив вказівників. Масив створюємо за допомогою команди `int** nums=new int*[size]`. Це динамічний масив. Тип його елементів визначається інструкцією `int*` у правій частині виразу. Зірочка `*` після `int`-інструкції свідчить про те, що йдеться про вказівники на цілі числа. Підкреслимо, що вказівники на цілі числа - елементи створеного масиву. Ім'я масиву є вказівником на перший елемент, який у цьому випадку сам є вказівником. Тому ім'я масиву `nums` є вказівником на вказівник на ціле число. Звідси змінну `nums` оголошено з ідентифікатором типу `int**`.

Подробиці



Зірочка * в інструкції оголошення вказівника насправді належить до оголошуваної змінної, а не до ідентифікатора типу. Наприклад, інструкція `int* x,y` оголошує вказівник `x` на цілочислове значення і звичайну (не вказівник) цілочислову змінну `y`. Більше того, замість оголошення `int* x,y` часто використовують оголошення `int *x,y`. Тут ідеться про одну й ту саму інструкцію, але записану трохи по-різному. Другий спосіб підкреслює, що зірочка * "діє" тільки на одну змінну. Тому думати, що `int*` є ідентифікатором для типу даних не зовсім коректно. Разом із тим, якщо інструкції виду `int*` чи `int**` розглядати як ідентифікатори типу, то в багатьох випадках це сприяє кращому розумінню принципів організації програмного коду.

Ще у програмі командою `const int cols[size]={3,7,6,4,2}` оголошується й ініціалізується константний масив `cols` (масив із елементами-константами). Елементи цього масиву визначають розмір динамічних числових масивів, які створюються далі в програмі, і вказівники на які записуються в масив `nums`.

Для створення динамічних числових масивів і заповнення їх значеннями запускається оператор циклу, в якому індексна змінна `i` "пробігає" значення від `0` до `size-1`. У тілі оператора циклу командою `nums[i]=new int[cols[i]]` створюється динамічний числовий масив. Розмір масиву визначається елементом `cols[i]` масиву `cols`. Вказівник на масив присвоюється значенням елементу `nums[i]` масиву вказівників. Після того, як числовий динамічний масив створено, він заповнюється випадковими числами, а значення його елементів виводяться у консольне вікно. Реалізується все це за допомогою оператора циклу, в якому індексна змінна `j` послідовно приймає значення в діапазоні від `0` до `cols[i]-1`. Значення елементу присвоюється командою `nums[i][j]=rand()%10`, а відображається значення елемента в консолі командою `cout<<nums[i][j]<<" | "`.

Подробиці



Масив `nums` є одномірним масивом. Інструкція `nums[i]` дає значення елемента масиву з індексом `i`. Але такий елемент є вказівником, причому вказівником на початковий елемент динамічного числового масиву. Вказівники, як ми пам'ятаємо, можна індексувати. Тому `nums[i][j]` є елементом з індексом `j` у масиві, на який вказує елемент `nums[i]`. Іншими словами, ми організували одномірні масиви так, що створюється повна ілюзія двомірного масиву.

Після того, як динамічні масиви створено і заповнено, потреба в них відпадає, і вони видаляються з пам'яті. Спочатку видаляються числові масиви. Для цього запускається оператор циклу, і в тілі циклу командою `delete [] nums[i]` видаляється кожен із числових динамічних масивів. Після цього командою `delete [] nums` видаляється динамічний масив вказівників.

Із поправкою на те, що в програмі використано генератор випадкових чисел (тому конкретні числові значення під час запуску програми на різних комп'ютерах можуть відрізнятися), результат виконання програми виглядає таким чином:

Результат виконання програми (із лістингу 3.8)

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

Раніше ми вже мали справу з двомірним символьним масивом (див. лістинг 3.6). Однак там уся "символьність" масиву зводилася до того, що елементи масиву були символами. Масив створювався й оброблявся поелементно, і в цьому сенсі він мало чим відрізняється від двомірних масивів інших типів. Водночас символьні масиви (і це ми також знаємо) у C++ є на

особливому рахунку. Деякі особливості символьного двомірного масиву проілюстровано в програмному коді в лістингу 3.9. Якщо точніше, то там створюється одномірний масив із вказівників на вказівники на символьні значення. Ситуація дещо схожа на ту, що була описана в попередньому прикладі.

Отже, розглянемо наведений нижче програмний код:

Лістинг 3.9. Масив символьних вказівників

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Масив символьних вказівників:
    char* str[3]={{"червоний"},{"жовтий"},{"зелений"}};
    // Відображення рядків:
    for(int i=0;i<3;i++){
        cout<<str[i]<<endl;
    }
    // Зчитування значень елементів масиву:
    cout<<str[0][2]<<str[1][1]<<str[1][0]<<str[2][1];
    cout<<str[0][3]<<str[2][5]<<str[0][7]<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 3.9)

```
червоний
жовтий
```

```
зелений
рожевий
```

Масив символьних вказівників створено командою `char* str[3]={{"червоний"},{"жовтий"},{"зелений"}}`. Формально ця команда означає, що створюється масив із трьох елементів, кожен із яких є вказівником на значення типу `char`. У правій частині виразу вказано список значень, якими ініціалізовано елементи масиву `str`. Текстові літерали `"червоний"`, `"жовтий"` і `"зелений"` автоматично записуються в масиви, і під час обробки цих літералів повертаються вказівники на відповідні масиви. У результаті значеннями елементам масиву `str` присвоюються адреси перших елементів масивів, у які були записані текстові літерали. Тому, наприклад, під час виконання команди виду `cout<<str[i]<<endl` в операторі циклу йдеться про виведення у консоль символьного вказівника, і як результат відображається відповідний текстовий літерал. У той самий час ми можемо звертатися до елементів масиву `str` так, як начебто це був двомірний масив. Наприклад, `str[0][2]` є буквою з індексом `2` в літералі з індексом `0` (третя буква в першому літералі - буква `'p'`), а `str[1][1]` - друга буква в другому літералі (буква `'o'`), і так далі. Принцип тут простий. Інструкція `str[i]` є вказівником на символьне значення, а насправді вказівником на перший елемент символьного масиву. Тоді інструкція `str[i][j]` є буквою з індексом `j` у масиві, адреса якого (його першого елемента) міститься в елементі з індексом `i` масиву `str`.

До уваги

Якщо ми спробуємо присвоїти нове значення елементу `str[i][j]` (скажімо, додамо в програмний код команду `srt[0][3]='z'`), отримаємо помилку часу виконання, оскільки тут має місце спроба змінити значення текстового літерала, який є константою.

Задачі для самостійного розв'язання

1. Напишіть програму, в якій створюються два числових масиви однакового розміру. Необхідно обчислити суму попарних добутків елементів цих масивів. Так, якщо через a_k і b_k позначити елементи масивів (індекс $0 \leq k < n$), то необхідно обчислити суму $\sum_{k=0}^{n-1} a_k b_k$.

2. Напишіть програму для створення числового масиву і обчислення суми квадратів елементів цього масиву.

3. Напишіть програму для створення двомірного числового масиву і обчислення суми квадратів його елементів.

4. Напишіть програму для створення квадратної матриці (реалізується через двомірний масив). Матрицю необхідно заповнити числовими значеннями (спосіб заповнення обрати самостійно), а потім виконати *транспонування* матриці: матриця симетрично "відображається" відносно головної діагоналі, у результаті чого елемент матриці a_{ij} стає елементом a_{ji}, і навпаки.

5. Напишіть програму для створення квадратної матриці (реалізується через двомірний масив). Матриця заповнюється випадковими числами, після чого виконується "поворот за годинниковою стрілкою": перший стовпець стає першим рядком, другий стовпець - другим рядком і так далі.

6. Напишіть програму, в якій для двомірної квадратної числової матриці розраховується *слід* - сума діагональних елементів матриці.

7. Напишіть програму, в якій для одномірного числового масиву розраховується найбільший (найменший) елемент і позиція, на якій елемент перебуває в масиві.

8. Напишіть програму для створення одномірного числового масиву. Після заповнення значеннями (наприклад, випадковими числами) масиву потрібно виконати циклічну перестановку елементів. Кількість позицій для циклічної перестановки вводить користувач із клавіатури.

9. Напишіть програму для створення і заповнення натуральними числами двомірного масиву. Заповнення починається з лівого верхнього елемента зліва направо, згори вниз (тобто заповнюється спочатку перший рядок, потім другий і так далі).

10. Напишіть програму для створення і заповнення натуральними числами двомірного масиву. Заповнення починається з лівого верхнього елемента згори вниз, зліва направо (тобто заповнюється спочатку перший стовпець, потім другий і так далі).

11. Напишіть програму для створення квадратної матриці. Елементам на обох діагоналях присвоюються одиничні значення, всі інші елементи нульові.

12. Напишіть програму для обчислення добутку прямокутних матриць (кількість рядків і стовпців у матрицях різна). Перемножуються матриці A (розмірами m на n) і B (розмірами n на l). Результатом є матриця C (розмірами m на l). Робоча формула для обчислення значень матриці C має вигляд $c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$, де $1 \le i \le m$ і $1 \le j \le l$.

13. Напишіть програму, у якій створюється символьний масив для запису тексту. У масив записується текст, після чого необхідно змінити порядок символів у тексті на протилежний: останній символ стає першим, передостанній символ - другим і так далі.

14. Напишіть програму для створення двомірного числового масиву і заповнення його випадковими числами. Для кожного рядка (чи стовпця) двомірного масиву визначається найбільший (чи найменший) елемент, і з таких елементів створюється одномірний числовий масив.

15. Напишіть програму, в якій створюється символьний масив для запису тексту. Підрахувати для записаного в масиві тексту кількість слів і довжину кожного слова. Словами вважати набір символів між пробілами.

16. Напишіть програму для створення динамічного масиву. Розмір масиву - випадкове число (тобто генерується випадкове число, яке визначає розмір масиву). Заповнити масив "симетричними" значеннями: перший і

останній елемент отримують значення 1, другий і передостанній - значення 2 і так далі.

17. Напишіть програму для створення трьох одномірних числових масивів однакового розміру. Перші два масиви заповнюються випадковими числами. Третій масив заповнюється так: порівнюють елементи з такими самими індексами в першому і другому масиві і в третій масив на таку ж позицію записують більше (чи менше) з порівнюваних значень.

18. Напишіть програму для створення двох динамічних масивів (різної довжини). Масиви заповнюють випадковими числами. Потім створюється третій динамічний масив, і його розмір дорівнює сумі розмірів перших двох масивів. Третій масив заповнюється так: спочатку в нього записують значення елементів першого масиву, а потім значення елементів другого масиву.

19. Напишіть програму для створення двох динамічних масивів однакового розміру. Масиви заповнюють випадковими числами. Потім створюють третій динамічний масив, і його розмір удвічі більший за розмір кожного з перших двох масивів. Третій масив заповнюють почерговим записом елементів із перших двох масивів: спочатку записується значення елемента першого масиву, потім значення елемента другого масиву, потім знову першого і знову другого, і так далі.

20. Напишіть програму для створення двомірного числового масиву. Масив заповнюють випадковими числами. На його основі створюється новий масив, шляхом "викреслювання" зі старого одного рядка і одного стовпця. Номери стовпця і рядка, які "викреслюють", визначаються шляхом уведення з клавіатури чи за допомогою генератора випадкових чисел.

Глава 4

Функції

Слухай, кого ти мені, каже, нагадуєш?

з х/ф "Пропала грамота"

Ця глава присвячена в основному функціям. Деякі досить прості приклади використання функцій ми вже зустрічали в попередніх главах. Тут же ми спробуємо показати всю міць і красу застосування функцій у програмних кодах. Предметом нашого розгляду стануть такі питання й теми:

- перевантаження функцій;
- механізми передачі аргументів функціям;
- значення аргументів функції за замовчуванням;
- застосування рекурсії;
- передача вказівників, масивів і текстових значень аргументами функціям;
- повернення результатом функції вказівника й посилання;
- використання вказівників на функції,

і деякі інші аспекти, пов'язані з функціями.

Оголошення й опис функції

Дуже простий приклад використання функцій наведено в лістингу 4.1. Там описано математичні функції для обчислення синуса й косинуса. У програмі ці функції використано для обчислення значень при заданому аргументі, а аналогічні вбудовані функції викликаються для перевірки результату обчислень.

Подробиці



Для обчислення синуса й косинуса (при заданому значенні аргументу x) ми використовуємо такі наближені вирази: $\cos(x) \approx 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots + \frac{(-1)^n x^{2n}}{(2n)!} = \sum_{k=0}^{n} \frac{(-1)^k x^{2k}}{(2k)!}$ для косинуса і $\sin(x) \approx x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots + \frac{(-1)^n x^{2n+1}}{(2n+1)!} =$

$\sum_{k=0}^{n} \frac{(-1)^k x^{2k+1}}{(2k+1)!}$ для синуса. Чим більше значення верхньої межі суми n, тим вища точність обчислень.

Розглянемо наведений нижче програмний код:

Лістинг 4.1. Обчислення синуса й косинуса

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Оголошення функцій:
double myCos(double); // Обчислення косинуса
double mySin(double); // Обчислення синуса
void show(double);    // Допоміжна функція
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Константа для значення числа "пі":
   const double pi=3.141592;
   // Обчислення синуса й косинуса при різних
   // значеннях аргументу:
   show(pi/6);
   show(pi/4);
   show(pi/3);
   show(pi/2);
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
// Опис функції для обчислення косинуса:
double myCos(double x){
   // Верхня межа суми:
```

```
    int n=100;
    // Змінні для запису значення суми і доданку до суми:
    double s=0,q=1;
    // Обчислення суми:
    for(int k=0;k<=n;k++){
        s+=q; // Доданок до суми
        // Доданок для наступної ітерації:
        q*=(-1)*x*x/(2*k+1)/(2*k+2);
    }
    // Результат функції:
    return s;
}
// Опис функції для обчислення синуса:
double mySin(double x){
    // Верхня межа суми:
    int n=100;
    // Змінні для запису значення суми і доданка до суми:
    double s=0,q=x;
    // Обчислення суми:
    for(int k=0;k<=n;k++){
        s+=q; // Доданок до суми
        // Доданок для наступної ітерації:
        q*=(-1)*x*x/(2*k+2)/(2*k+3);
    }
    // Результат функції:
    return s;
}
// Опис допоміжної функції:
void show(double x){
    cout<<"Значення аргументу: "<<x<<endl;
    // Обчислення косинуса:
    cout<<"Косинус: "<<myCos(x)<<" vs. "<<cos(x)<<endl;
```

```
    // Обчислення синуса:
    cout<<"Синус: "<<mySin(x)<<" vs. "<<sin(x)<<endl;
    // Додатковий відступ:
    cout<<endl;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 4.1)

```
Значення аргументу: 0.523599
Косинус: 0.866025 vs. 0.866025
Синус: 0.5 vs. 0.5

Значення аргументу: 0.785398
Косинус: 0.707107 vs. 0.707107
Синус: 0.707107 vs. 0.707107

Значення аргументу: 1.0472
Косинус: 0.5 vs. 0.5
Синус: 0.866025 vs. 0.866025

Значення аргументу: 1.5708
Косинус: 3.26795e-007 vs. 3.26795e-007
Синус: 1 vs. 1
```

У програмі описано функції `myCos()` для обчислення косинуса, `mySin()` для обчислення синуса та допоміжна функція `show()`. Причому функції спочатку оголошуються, а опис функцій виконується безпосередньо після опису головної функції `main()`.

Подробиці



При оголошенні функції вказують її прототип: ідентифікатор типу результату, назва функції і список аргументів, причому назви аргументів можна не

вказувати - лише їхній тип.

Функція повинна бути оголошена до того, як її вперше викликають.

Функція `myCos()` результатом повертає значення типу `double`, і у неї один аргумент (позначений як `x`), теж типу `double`. Локальна цілочислова змінна `n`, оголошена й ініціалізована зі значенням `100` у тілі функції, визначає верхню межу суми, що повертається результатом функції. Ще дві змінні типу `double` призначені для запису значення суми (змінна `s` із початковим значенням `0`) і доданка, який на кожній ітерації додають до суми (змінна `q` із початковим значенням `1`). Обчислюється сума за допомогою оператора циклу, в якому індексна змінна `k` пробігає значення від `0` до `n` включно, і за кожен цикл виконуються команди `s+=q` (додавання нового доданка до суми) і `q*=(-1)*x*x/(2*k+1)/(2*k+2)` (обчислення доданку для наступної ітерації). Після закінчення обчислень значення змінної `s` повертається як результат функції.

До уваги

Для косинуса обчислюється сума $1-\frac{x^2}{2!}+\frac{x^4}{4!}-\frac{x^6}{6!}+\cdots+\frac{(-1)^n x^{2n}}{(2n)!}=\sum_{k=0}^{n} q_k$, де через $q_k=\frac{(-1)^k x^{2k}}{(2k)!}$ позначено доданок до суми (для ітерації з індексом k). Якщо q_{k+1} - доданок на наступній ітерації, то легко помітити, що $\frac{q_{k+1}}{q_k}=\frac{(-1)x^2}{(2k+1)(2k+2)}$. Висновок такий: щоб отримати доданок для наступної ітерації, поточне значення доданка слід помножити на величину $\frac{(-1)x^2}{(2k+1)(2k+2)}$. Такий підхід реалізовано у функції `myCos()`.

Для синуса обчислюється сума $x-\frac{x^3}{3!}+\frac{x^5}{5!}-\frac{x^7}{7!}+\cdots+\frac{(-1)^{n+1} x^{2n+1}}{(2n+1)!}=\sum_{k=0}^{n} q_k$, де позначено $q_k=\frac{(-1)^k x^{2k+1}}{(2k+1)!}$. Відношення $\frac{q_{k+1}}{q_k}=\frac{(-1)x^2}{(2k+2)(2k+3)}$. Тому під час обчислення синуса (функція `mySin()`), розраховуючи доданок для наступної ітерації, значення поточного доданка множимо на $\frac{(-1)x^2}{(2k+2)(2k+3)}$.

Функцію для обчислення синуса `mySin()` описано аналогічно до функції для обчислення косинуса `myCos()`, хіба що з поправкою на змінене початкове

значення для доданка (змінна `q`) і спосіб її обробки (команда `q*=(-1)*x*x/(2*k+2)/(2*k+3)` у тілі оператора циклу).

Функція `show()` не повертає результат, тому ідентифікатором типу результату для неї вказано ключове слово `void`. У функції є аргумент типу `double`, позначений як `x`. Це значення використовується під час обчислення синуса й косинуса.

Функція `show()` відображує значення аргументу, для якого обчислюються синус і косинус, значення синуса й косинуса, обчислені за допомогою функцій `mySin()` і `myCos()`, а також для порівняння наведено значення для синуса й косинуса, обчислені за допомогою вбудованих функцій `sin()` і `cos()`.

До уваги

Для використання вбудованих функцій `sin()` і `cos()` у програмі підключаємо заголовок `<cmath>`.

Код головної функції програми достатньо простий. У тілі функції `main()` визначається дійсна числова константа `pi` зі значенням для числа $\pi \approx 3.141592$, а потім шляхом виклику функції `show()` обчислюється синус і косинус для значень аргументів $\frac{\pi}{6}$, $\frac{\pi}{4}$, $\frac{\pi}{3}$ і $\frac{\pi}{2}$. Як бачимо, збіг зі значеннями, обчисленими за допомогою вбудованих функцій, більш ніж прийнятний.

Перевантаження функцій

Розв'яжемо програмними методами задачу про обчислення прибутку, отримуваного вкладником банку. Вхідними параметрами задачі є:

- вихідна грошова сума, яку вкладник розміщує в банку на депозитному рахунку;
- річна відсоткова ставка, за якою нараховується прибуток на вкладену суму;
- кількість років, на які розміщують депозит;
- кількість періодів у році, коли виконують нарахування відсотків.

Подробиці



Якщо на депозит вносять суму m, і на вклад нараховується r відсотків річних, то через рік фінальна сума (прибуток) складатиме величину $m\left(1+\frac{r}{100}\right)$. Якщо йдеться про декілька років, і за кожен рік діє ставка нарахування відсотка r, то через y років кінцева грошова сума на депозиті складатиме величину $m\left(1+\frac{r}{100}\right)^y$.

Вище малося на увазі, що відсотки нараховують раз на рік. У дійсності може застосовуватися схема, за якою відсотки нараховують декілька разів на рік. Наприклад, якщо відсотки нараховують n разів на рік через однакові проміжки часу, то на один такий інтервал часу припадає ставка відсотка $\frac{r}{n}$, де через r позначена річна відсоткова ставка. При цьому за y років буде ny нарахувань зі ставкою відсотка $\frac{r}{n}$. Як наслідок, для підсумкової суми вкладу маємо $m\left(1+\frac{r}{100\cdot n}\right)^{n\cdot y}$.

Ми хочемо передбачити у програмі такі можливості:

- якщо під час обчислення прибутку вказують лише початкову суму і відсоткову ставку, то автоматично вважається, що депозит розміщують на один рік, і відсотки нараховуються теж один раз на рік;
- якщо під час обчислення прибутку вказують початкову суму, відсоткову ставку і кількість років, то вважаємо, що відсотки нараховуються раз на рік;
- також можна під час обчислення прибутку вказати початкову суму, річну відсоткову ставку, кількість років і кількість періодів нарахування відсотків.

Для всіх перерахованих випадків ми опишемо окрему функцію, яка обчислює кінцеву суму, причому скористаємося таким механізмом, як *перевантаження функцій*.

Теорія

Перевантаження функцій полягає в тому, що в програмі створюється декілька функцій із однаковими назвами. Такі функції повинні відрізнятися прототипом (тип результату, назва функції, кількість і тип аргументів), щоб під час виклику функції можна було однозначно визначити, про яку саме версію функції йдеться. Про функції з однаковими назвами зазвичай говорять як про різні версії однієї функції.

Програмний код із розв'язанням цієї задачі наведено в лістингу 4.2.

Лістинг 4.2. Перевантаження функцій

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція з двома аргументами:
double getMoney(double m,double r){
   return m*(1+r/100);
}
// Функція з трьома аргументами:
double getMoney(double m,double r,int y){
   double s=m;
   for(int k=1;k<=y;k++){
      s*=(1+r/100);
   }
   return s;
}
// Функція з чотирма аргументами:
double getMoney(double m,double r,int y,int n){
   return getMoney(m,r/n,y*n);
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Початкова сума вкладу:
   double money=1000;
   // Відсоткова ставка:
   double rate=5;
   cout<<"Початкова сума: "<<money<<endl;
   cout<<"Річна ставка: "<<rate<<"%\n";
```

```
    // Обчислення прибутку за різні проміжки часу:
    cout<<"Вклад на один рік: "<<getMoney(money,rate)<<endl;
    cout<<"Вклад на 7 років: "
<<getMoney(money,rate,7)<<endl;
    cout<<"Вклад на 7 років\n(нарахування 3 рази на рік): ";
    cout<<getMoney(money,rate,7,3)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче подано результат виконання програми:

Результат виконання програми (із лістингу 4.2)

```
Початкова сума: 1000
Річна ставка: 5%
Вклад на один рік: 1050
Вклад на 7 років: 1407.1
Вклад на 7 років
(нарахування 3 рази на рік): 1414.98
```

Проаналізуємо програмний код. Нас в основному буде цікавити опис функції `getMoney()`. У програмі описано три версії цієї функції, які відрізняються лише кількістю аргументів, які передаються функції. Розглянемо кожний із них окремо.

Найпростіший код у функції `getMoney()` з двома аргументами. У цьому випадку `double`-аргумент `m` визначає початкову суму вкладу, а ще один `double`-аргумент `r` дає значення річної відсоткової ставки. Як результат функції повертається значення виразу `m*(1+r/100)` у повній відповідності до формули обчислення кінцевої суми для річного депозиту.

У версії функції з трьома аргументами перші два аргументи `m` і `r` мають таке ж призначення, як і в попередньому випадку, а третій цілочисловий аргумент `y`

визначає кількість років, на які розміщено депозит. Результат у тілі функції обчислюють за допомогою оператора циклу. Локальна змінна `s` із початковим значенням `m` (перший аргумент функції), яка повертається як результат функції, за кожний цикл множиться на множник `(1+r/100)`, і таких циклів `y`. Як наслідок отримуємо значення, яке відповідає формулі обчислення кінцевої суми депозиту, розміщеного на декілька років (із однократним нарахуванням відсотків у кінці року).

У версії функції `getMoney()` з чотирма аргументами останній цілочисловий аргумент `n` визначає кількість періодів у році для нарахування відсотків. Як результат функції повертається значення виразу `getMoney(m,r/n,y*n)`. Тут ми фактично у тілі функції з чотирма аргументами викликаємо версію цієї ж самої функції з трьома аргументами.

До уваги

Далі ми будемо розглядати такий механізм, як *рекурсія*. Під час рекурсії у тілі функції викликається та ж сама функція. Тут, у цьому прикладі, про рекурсію не йдеться, оскільки в одній версії функції викликається друга версія функції. Технічно це все одно, якби в одній функції викликалася зовсім інша функція. Просто в даному випадку у таких різних по суті функцій однакові назви.

Що ж до виразу `getMoney(m,r/n,y*n)`, який повертається як результат для функції `getMoney()` з чотирма аргументами, то тут ми скористалися тією очевидною обставиною, що кінцева сума за депозитом на y років із початковою сумою вкладу m під r відсотків річних з n-кратним нарахуванням відсотків дорівнює кінцевій сумі вкладу з початковою сумою m, розміщеною під ставку відсотка $\frac{r}{n}$ на ny років з однократним нарахуванням відсотків.

У головній функції програми наведено приклади виклику функції `getMoney()` з різною кількістю аргументів. Кожного разу, коли викликається функція, що має декілька версій, рішення про залучення тієї чи іншої версії функції приймається на основі команди виклику: в основному за кількістю і типом аргументів, а також за типом результату.

Значення аргументів за замовчуванням

Розглянуту вище задачу можна було б розв'язати і дещо по-іншому. Зокрема, у цьому конкретному випадку замість перевантаження функцій ми цілком могли обмежитися створенням усього однієї функції, але з аргументами, що мають *значення за замовчуванням*.

Теорія

При описі функції для деяких або всіх її аргументів можна задати ***значення за замовчуванням***. Ці значення використовуються, якщо під час виклику функції відповідний аргумент не вказаний. Аргументи, що мають значення за замовчуванням, описуються у списку аргументів функції останніми. Значення за замовчуванням для аргументу задають у прототипі функції після імені аргументу через знак рівності.

У лістингу 4.3 наведено програму, у якій розв'язано поставлену раніше задачу. Але тепер ми створюємо всього одну функцію `getMoney()`, а два її останніх аргументи мають значення за замовчуванням.

Лістинг 4.3. Значення аргументів за замовчуванням

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція з аргументами, що мають
// значення за замовчуванням:
double getMoney(double m,double r,int y=1,int n=1){
   double s=m;
   double z=n*y;
   double q=r/n;
   for(int k=1;k<=z;k++){
      s*=(1+q/100);
   }
   return s;
```

```
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Початкова сума вкладу:
    double money=1000;
    // Відсоткова ставка:
    double rate=5;
    cout<<"Початкова сума: "<<money<<endl;
    cout<<"Річна ставка: "<<rate<<"%\n";
    // Обчислення прибутку за різні проміжки часу:
    cout<<"Вклад на один рік: "<<getMoney(money,rate)<<endl;
    cout<<"Вклад на 7 років: "
<<getMoney(money,rate,7)<<endl;
    cout<<"Вклад на 7 років\n(нарахування 3 рази на рік): ";
    cout<<getMoney(money,rate,7,3)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такі ж, як і в попередньому випадку:

Результат виконання програми (із лістингу 4.3)

```
Початкова сума: 1000
Річна ставка: 5%
Вклад на один рік: 1050
Вклад на 7 років: 1407.1
Вклад на 7 років
(нарахування 3 рази на рік): 1414.98
```

Коли функція `getMoney()` викликається з трьома аргументами, це все одно, якби четвертий її аргумент дорівнював би одиниці (значення за

замовчуванням). Якщо функція викликається тільки з двома аргументами, то третій і четвертий її аргументи автоматично вважаються одиничними (значення за замовчуванням для цих аргументів). Щодо коду функції, то локальна змінна `s` із початковим значенням `m` визначає стартовий вклад, змінна `z` зі значенням `n*y` визначає загальну кількість періодів нарахування відсотка, змінна `q` зі значенням `r/n` визначає відсоткову ставку для кожного періоду, за який нараховують відсотки. Результат обчислюється за допомогою оператора циклу, в якому `z` циклів, і за кожний цикл змінна `s` множиться на `(1+q/100)`. Після закінчення обчислень значення змінної `s` повертається як результат функції. Тут, за великим рахунком, ми повторили алгоритм обчислень, використаний у попередньому прикладі для випадку функції з трьома аргументами. Єдине, ми зробили "поправку" на ефективну відсоткову ставку і кількість періодів нарахувань відсоткового прибутку.

До уваги

На перший погляд може здатися, що механізм визначення значень аргументів за замовчуванням більш простий у використанні, ніж перевантаження функцій. У принципі, це так. Однак варто розуміти, що перевантаження функцій - технологія виключно гнучка і ефективна, і не може бути замінена визначенням значень аргументів за замовчуванням.

Також важливо мати на увазі, що і перевантаження функцій, і визначення значень за замовчуванням для аргументів можна використовувати одночасно. У такому випадку функції повинні перевантажуватися так, щоб за контекстом виклику функції завжди можна було однозначно ідентифікувати версію функції і те, які аргументи передано, а які - ні.

Рекурсія

Якщо в описі функції викликається ця сама функція (але, як правило, із іншим аргументом чи аргументами), то така ситуація називається *рекурсією*. Приклад використання рекурсії наведено в лістингу 4.4. Йдеться про ту ж саму задачу щодо обчислення кінцевої суми депозиту, але тепер для простоти ми

розглядаємо не три, а один спосіб нарахування відсотків: депозит розміщують на зазначену кількість років під фіксовану річну ставку відсотка, а нарахування прибутку за відсотками відбувається раз на рік. У програмі описано функцію `getMoney()` із трьома аргументами і під час описання функції використано рекурсію. Розглянемо поданий нижче програмний код:

Лістинг 4.4. Рекурсія

```
#include <iostream>
#include <cstdlib>
using namespace std;
// У функції застосовано рекурсивний виклик:
double getMoney(double m,double r,int y){
   if(y==0){
      return m;
   }
   else{
      return (1+r/100)*getMoney(m,r,y-1);
   }
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Початкова сума вкладу:
   double money=1000;
   // Відсоткова ставка:
   double rate=5;
   cout<<"Початкова сума: "<<money<<endl;
   cout<<"Річна ставка: "<<rate<<"%\n";
   // Обчислення прибутку за різні проміжки часу:
   cout<<"Вклад на 1 рік: "<<getMoney(money,rate,1)<<endl;
```

```
    cout<<"Вклад на 7 років: "
<<getMoney(money,rate,7)<<endl;
    cout<<"Вклад на 10 років: "
<<getMoney(money,rate,10)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми буде таким:

Результат виконання програми (із лістингу 4.4)

```
Початкова сума: 1000
Річна ставка: 5%
Вклад на 1 рік: 1050
Вклад на 7 років: 1407.1
Вклад на 10 років: 1628.89
```

У програмі описано функцію `getMoney()` із трьома аргументами, після чого в головній функції програми функція `getMoney()` викликається з різними значеннями третього аргументу.

Інтерес викликає спосіб описання функції `getMoney()`. Основу коду функції складає умовний оператор, у якому перевіряється умова `y==0` (через `y` позначено третій аргумент функції `getMoney()`). Якщо умова істинна, як результат функції повертається значення `m` (перший аргумент функції `getMoney()`). Тут ми виходимо з того, що на вклад, розміщений у банку на `0` років, відсотки не нараховують. Тому кінцева сума в такий ситуації дорівнює початковій сумі вкладу. Якщо ж умова `y==0` хибна, то як результат функції повертається значення виразу `(1+r/100)*getMoney(m,r,y-1)`. Це і є рекурсивний виклик функції `getMoney()`.

Подробиці



Під час визначення функції ми виходили з таких міркувань. Припустимо, що `getMoney(m,r,y)` дає значення кінцевої суми за депозитом через `y` років. Як можна отримати цю суму? Дуже просто: нарахувати річні відсотки на кінцеву суму вкладу за період `y-1` років. Кінцеву суму вкладу за `y-1` років дає вираз `getMoney(m,r,y-1)`. Нарахування річних відсотків означає множення на величину `(1+r/100)`. У підсумку отримуємо, що значення `getMoney(m,r,y)` і `(1+r/100)*getMoney(m,r,y-1)` повинні збігатися.

Значення функції `getMoney()` із заданими аргументами `m`, `r` і `y` (значення виразу `getMoney(m,r,y)`) обчислюється так. Перевіряється значення аргументу `y`, і якщо воно відмінне від нуля, обчислюється значення виразу `(1+r/100)*getMoney(m,r,y-1)`. Під час обчислення значення виразу знову викликається функція `getMoney()`, але на цей раз третій аргумент на одиницю менший і так далі, поки не буде викликано функцію `getMoney()` із нульовим третім аргументом. Тоді під час виклику функції повертається значення `m` (перший аргумент функції). Це дозволяє обчислити значення попереднього виразу, потім того, що був перед ним, і так далі, аж до самого першого виразу, із якого все починалося.

Механізми передачі аргументів функціям

Щоб зрозуміти природу проблеми, яка буде обговорюватися, розглянемо невеликий приклад, поданий у лістингу 4.5. Суть програми дуже проста:

- описується функція `swap()` з двома символьними (тип `char`) аргументами;
- під час виклику функції значення переданих аргументів виводяться в консольне вікно;

• аргументи обмінюються значеннями, і їхні нові значення також відображуються в консольному вікні.

У програмі оголошуються дві змінні, вони передаються як аргументи функції `swap()` під час її виклику, а потім перевіряються значення цих змінних.

Програмний код, що нас цікавить, такий:

Лістинг 4.5. Передача аргументів за значенням

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача аргументів за значенням:
void swap(char a,char b){
    cout<<"Викликається функція swap()"<<endl;
    // Перевіряються значення аргументів функції:
    cout<<"Перший аргумент: "<<a<<endl;
    cout<<"Другий аргумент: "<<b<<endl;
    // Зміна значень аргументів функції:
    char t=b;
    b=a;
    a=t;
    for(int i=1;i<=20;i++){
       cout<<"-";
    }
    cout<<endl;
    // Перевіряються значення аргументів функції:
    cout<<"Перший аргумент: "<<a<<endl;
    cout<<"Другий аргумент: "<<b<<endl;
    cout<<"Виконання функції swap() закінчено"<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
```

```
    system("chcp 1251>nul");
    // Змінні для передачі аргументами функції:
    char x='A',y='B';
    // Перевіряються значення змінних:
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    // Виклик функції:
    swap(x,y);
    // Перевіряються значення змінних:
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Наші очікування від результатів виконання програми могли б бути такими: після виклику функції `swap()` змінні, які передаються їй як аргументи, обмінюються значеннями. Насправді результат виконання програми такий, як наведено нижче:

Результат виконання програми (із лістингу 4.5)

```
Перша змінна: A
Друга змінна: B
Викликається функція swap()
Перший аргумент: A
Другий аргумент: B
--------------------
Перший аргумент: B
Другий аргумент: A
Виконання функції swap() закінчено
Перша змінна: A
Друга змінна: B
```

Як бачимо, під час перевірки значень аргументів у тілі функції все йде за планом: аргументи обмінялися значеннями. Але ось коли ми перевіряємо значення змінних після виклику функції `swap()` виявляється, що їхні значення не змінилися. Причина полягає у способі передачі аргументів функції. За замовчуванням використовується режим передачі аргументів *за значенням*: передаються не самі змінні, а їхні копії (після закінчення виконання функції копії аргументів видаляються з пам'яті). Тому під час виклику функції `swap()` усі операції виконуються не зі змінними `x` і `y`, які передавалися як аргументи функції, а з їхніми копіями. Саме копії обмінюються своїми значеннями. Значення змінних `x` і `y` лишаються незмінними.

Окрім передачі аргументів за значенням, аргументи передаються ще й *за посиланням*. У цьому випадку у функцію передаються не копії змінних, а самі змінні. Щоб передавати аргументи за посиланням, у описі функції перед назвами аргументів вказують інструкцію `&`. У лістингу 4.6 функцію `swap()` описано так, що аргументи їй передаються за посиланням. Порівняно з програмою з лістингу 4.5 зміни мінімальні (для зручності в поданому далі коді видалено майже всі коментарі, а інструкції `&` в описі функції `swap()` виділено жирним шрифтом):

Лістинг 4.6. Передача аргументів за посиланням

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача аргументів за посиланням:
void swap(char &a,char &b){
   cout<<"Викликається функція swap()"<<endl;
   cout<<"Перший аргумент: "<<a<<endl;
   cout<<"Другий аргумент: "<<b<<endl;
   char t=b;
   b=a;
```

```
    a=t;
    for(int i=1;i<=20;i++){
       cout<<"-";
    }
    cout<<endl;
    cout<<"Перший аргумент: "<<a<<endl;
    cout<<"Другий аргумент: "<<b<<endl;
    cout<<"Виконання функції swap() закінчено"<<endl;
}
int main(){
    system("chcp 1251>nul");
    char x='A',y='B';
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    swap(x,y);
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 4.6)

```
Перша змінна: A
Друга змінна: B
Викликається функція swap()
Перший аргумент: A
Другий аргумент: B
--------------------
Перший аргумент: B
Другий аргумент: A
Виконання функції swap() закінчено
```

```
Перша змінна: B
Друга змінна: A
```

Як бачимо, змінні насправді змінили свої значення після їх передачі аргументами у функцію `swap()`.

Передача вказівника як аргументу функції

Ще один спосіб опису функції `swap()`, але цього разу як аргументи функції передаються вказівники. Розглянемо програму, подану в лістингу 4.7.

Лістинг 4.7. Передача вказівника як аргументу функції

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Як аргументи функції передаються вказівники:
void swap(char* a,char* b){
   cout<<"Викликається функція swap()"<<endl;
   // Перевіряються значення аргументів функції:
   cout<<"Перша змінна: "<<*a<<endl;
   cout<<"Друга змінна: "<<*b<<endl;
   // Зміна значень змінних:
   char t=*b;
   *b=*a;
   *a=t;
   for(int i=1;i<=20;i++){
      cout<<"-";
   }
   cout<<endl;
   // Перевіряються значення змінних:
   cout<<"Перша змінна: "<<*a<<endl;
   cout<<"Друга змінна: "<<*b<<endl;
```

```
    cout<<"Виконання функції swap() закінчено"<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Змінні для передачі в функцію:
    char x='A',y='B';
    // Перевіряються значення змінних:
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    // Виклик функції:
    swap(&x,&y);
    // Перевіряються значення змінних:
    cout<<"Перша змінна: "<<x<<endl;
    cout<<"Друга змінна: "<<y<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 4.7)

```
Перша змінна: A
Друга змінна: B
Викликається функція swap()
Перша змінна: A
Друга змінна: B
--------------------
Перша змінна: B
Друга змінна: A
Виконання функції swap() закінчено
```

```
Перша змінна: B
Друга змінна: A
```

У програмі в описі функції `swap()` тип обох аргументів вказаний як `char*`. Це означає, що як аргументи функції передаються вказівники на символьні значення. Тому коли функція `swap()` викликається у головній функції програми, команда виклику має вигляд `swap(&x,&y)`. Оскільки змінні `x` і `y` оголошено як такі, що відносяться до типу `char`, то `&x` і `&y` (адреси змінних) - це вказівники на значення типу `char`. Іншими словами, якщо в попередніх прикладах у функцію передавалися змінні (або їхні копії), то цього разу як аргументи функції передаються вказівники на змінні.

Отже, як аргументи функції `swap()` передаються вказівники. Але перевіряються значення, які записано за відповідними адресами. І обмін значеннями відбувається не для вказівників, а для значень, які записано за адресами, що є значеннями вказівників. Так, командою `char t=*b` оголошується локальна змінна `t`, і їй присвоюється значення, записане за адресою із вказівника `b` (другий аргумент функції). Далі командою `*b=*a` за адресою із вказівника `b` записується значення, розміщене за адресою із вказівника `a` (перший аргумент функції). Після цього командою `*a=t` за адресою, що міститься у вказівнику `a`, записується значення змінної `t` (саме це значення на самому початку було записано за адресою із вказівника `b`).

Важливо розуміти, що коли ми передаємо як аргументи функції вказівники, вони передаються *за значенням*. Тобто насправді створюється копія аргументу, який передається функції. Але специфіка ситуації в тому, що копія вказівника містить те ж значення, що й вихідний вказівник, а тому копію вказівника встановлено на ту саму комірку пам'яті, що й оригінальний вказівник. У тілі функції, у свою чергу, операції (зі зміни значення) виконуються не із вказівниками, а з комірками, адреси яких записано у вказівниках. Так що механізм передачі аргументів за замовчуванням у цьому прикладі принципової ролі не грає.

Передача масиву як аргументу функції

Нерідко доводиться передавати аргументом функції масив. Як це робити у випадку одномірного масиву, ілюструє програма із лістингу 4.8.

Теорія

Одномірний масив передають у функцію за допомогою двох параметрів: вказівника на перший елемент масиву (ім'я масиву) і цілого числа, що визначає кількість елементів у масиві.

У поданій нижче програмі описано функцію `mean()`, як аргумент якій передається числовий масив, а як результат повертається середнє значення елементів цього масиву. У головній функції програми функція `mean()` викликається для обчислення середнього значення елементів для двох масивів.

Лістинг 4.8. Передача аргументом функції одномірного масиву

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Передача аргументом функції масиву:
double mean(double* m,int n){
   // Локальна змінна для запису суми
   // значень елементів масиву:
   double s=0;
   // Обчислення суми значень елементів масиву:
   for(int k=0;k<n;k++){
      s+=m[k];
   }
   // Результат функції - середнє значення:
   return s/n;
}
```

```
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Перший масив:
   double A[]={1,3,8,-2,4};
   // Другий масив:
   double B[]={4,6,2};
   // Виклик функції з передачею аргументом масиву:
   cout<<"Середнє по масиву A: "<<mean(A,5)<<endl;
   cout<<"Середнє по масиву B: "<<mean(B,3)<<endl;
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 4.8)

```
Середнє по масиву A: 2.8
Середнє по масиву B: 4
```

Функцію `mean()` описано з двома аргументами. Перший аргумент `m` із ідентифікатором типу `double*` формально є вказівником на значення типу `double`, але враховуючи, що ім'я масиву є вказівником на перший елемент масиву, ми цілком можемо ототожнювати аргумент `m` із іменем масиву. Саме так змінна `m` обробляється у тілі функції. Другий цілочисловий аргумент `n` функції визначає розмір масиву.

У тілі функції розраховується сума значень елементів масиву, і отримане значення записується в локальну змінну `s`. Як результат функцією повертається значення `s/n` (сума значень елементів, поділена на кількість елементів - тобто середнє значення).

У головній функції програми командами `double A[]={1,3,8,-2,4}` і `double B[]={4,6,2}` створюються й ініціалізуються два числових масиви.

До уваги

Під час оголошення масивів розмір не вказують. Він визначається автоматично за кількістю елементів, зазначених у списку ініціалізації масиву.

Під час виклику функції `mean()` як аргументи передаються ім'я масиву і розмір масиву: наприклад, `mean(A,5)` або `mean(B,3)`.

Аналогічним чином у функцію передається двомірний масив. У лістингу 4.9 наведено програмний код, у якому описано функцію `show()`. Як аргумент функції передається двомірний масив, значення елементів якого відображаються в консольному вікні.

Теорія

Якщо йдеться про статичний двомірний масив, то він оброблюється як одномірний масив (назвемо його зовнішнім масивом), елементами якого є одномірні масиви (назвемо їх внутрішніми масивами). Тут є один принциповий момент: кожен такий внутрішній масив-елемент має розмір, який є невід'ємною характеристикою типу для елемента зовнішнього масиву. Простіше кажучи, мало вказати, що зовнішній масив складається із внутрішніх масивів. Необхідно конкретизувати розмір цих внутрішніх масивів. При передачі двомірного масиву у функцію насправді як аргументи передають дві характеристики: назву двомірного масиву з явним зазначенням розміру за другим індексом, а також цілочислове значення, що визначає розмір двомірного масиву за першим індексом (розмір зовнішнього масиву).

Розглянемо програмний код:

Лістинг 4.9. Передача двомірного масиву у функцію

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Розмір масиву за другим індексом:
const int n=3;
// Функція для обробки двомірного статичного масиву:
```

```
void show(int M[][n],int p){
    for(int i=0;i<p;i++){
        for(int j=0;j<n;j++){
            cout<<M[i][j]<<" ";
        }
        cout<<endl;
    }
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Перший масив:
    int A[2][n]={{1,2,3},{4,5,6}};
    // Другий масив:
    int
B[][n]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}};
    cout<<"Перший масив:\n";
    show(A,2);
    cout<<"Другий масив:\n";
    show(B,4);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 4.9)

```
Перший масив:
1 2 3
4 5 6
Другий масив:
```

```
11 12 13
14 15 16
17 18 19
20 21 22
```

У програмі для визначення розміру двомірного масиву за другим індексом оголошується цілочислова константа `n`. При цьому, хоча ця константа доступна у функції `show()`, перший аргумент функції все одно описаний виразом `int M[][n]` із явним зазначенням розміру масиву за другим індексом. Таким чином, функція `show()` застосовна для відображення вмісту масивів, розмір яких за другим індексом дорівнює `n`. Другий аргумент функції `show()` ототожнюється з розміром двомірного масиву за першим індексом. У головній функції програми функція `show()` використовується для відображення вмісту масивів `A` і `B`, створених й ініціалізованих командами `int A[2][n]={{1,2,3},{4,5,6}}` й `int B[][n]={{11,12,13},{14,15,16},{17,18,19},{20,21,22}}`. У цих масивів розмір за другим індексом дорівнює `n`, а розміри за першим індексом різні. Причому для масиву `A` розмір за першим індексом вказано явно, а розмір за першим індексом для масиву `B` (оскільки під час його опису перша квадратна дужка пуста) визначається автоматично відповідно до вмісту списку ініціалізації для цього масиву. Під час виклику функції `show()` як перший аргумент передається ім'я масиву, а як другий аргумент - розмір масиву за першим індексом.

Дещо інакше все відбувається, коли у функцію необхідно передати двомірний динамічний масив.

Теорія

Двомірний динамічний масив за фактом є одномірним масивом із вказівників на вказівники. Кожний елемент-вказівник такого масиву містить адресу першого елемента іншого одномірного масиву. Таким чином, передача двомірного динамічного масиву у функцію зводиться до передачі одномірного масиву (два

параметри: вказівник на перший елемент масиву і розмір масиву), плюс ще один аргумент, який визначає розміри внутрішніх масивів. Тому зазвичай обмежуються передачею трьох аргументів: один є вказівником на вказівник, і два цілочислових аргументи.

У лістингу 4.10 наведено програму, у якій, як і в попередньому випадку, описано функцію `show()`, але цього разу вона призначена для відображення вмісту двомірного динамічного масиву.

До уваги

Для виконання форматованого виведення під час роботи з числовими значеннями ми використовуємо функцію `printf()`. Відповідно, у програмі підключається заголовок `<cstdio>` і не підключається заголовок `<iostream>`.

Порівняно з попереднім прикладом (див. лістинг 4.9), тут змінено спосіб опису аргументів функції `show()`, ну і складніше (порівняно зі статичним масивом) виглядає процедура створення двомірного динамічного масиву в головній функції програми:

Лістинг 4.10. Передача двомірного динамічного масиву в функцію

```
#include <cstdio>
#include <cstdlib>
using namespace std;
// Аргументом функції передається двомірний
// динамічний масив:
void show(int** M,int p,int n){
   for(int i=0;i<p;i++){
      for(int j=0;j<n;j++){
         printf("%4d",M[i][j]);
      }
      printf("\n");
   }
}
```

```
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Розміри динамічного масиву й індексні змінні:
   int a=3,b=5,i,j;
   // Створення масиву вказівників:
   int** A=new int*[a];
   // Створення і заповнення внутрішніх масивів:
   for(i=0;i<a;i++){
      A[i]=new int[b];
      for(j=0;j<b;j++){
         // Значення елемента масиву:
         A[i][j]=i*b+j+1;
      }
   }
   printf("Вміст масиву:\n");
   // Відображення вмісту масиву:
   show(A,a,b);
   // Видалення внутрішніх масивів:
   for(i=0;i<a;i++){
      delete [] A[i];
   }
   // Видалення масиву вказівників:
   delete [] A;
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
```

При виконанні програми отримуємо такий результат:

Результат виконання програми (із лістингу 4.10)

```
Вміст масиву:
   1   2   3   4   5
   6   7   8   9  10
  11  12  13  14  15
```

Якщо порівнювати способи передачі у функцію двомірного статичного і динамічного масивів, то з динамічним масивом немає жорсткої прив'язки до розміру масиву за другим індексом. У розглянутому прикладі функція `show()` застосовна для відображення вмісту будь-яких (мається на увазі будь-яких розмірів) двомірних динамічних масивів.

Передача тексту у функцію

Символьний одномірний масив із текстом у принципі передається як аргумент функції так само, як і будь-який інший одномірний масив, із поправкою на те, що розмір масиву явно вказувати не потрібно - він визначається автоматично за розташуванням нуль-символу в масиві (точніше, автоматично визначається розмір тексту в масиві).

Теорія

Під час передачі символьного масиву з текстом у функцію як аргументу функції достатньо передати вказівник на перший елемент масиву (ім'я масиву).

Приклади опису функцій, у які передаються символьні одномірні масиви, наведено в програмі в лістингу 4.11.

Лістинг 4.11. Передача у функцію символьного масиву

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція для визначення довжини рядка:
int getLength(char* str){
   int s=0;
```

```
    for(int i=0;str[i];i++){
        s++;
    }
    return s;
}
// Функція для визначення кількості пробілів у рядку:
int getSpace(char* str){
    int s=0;
    for(int i=0;str[i];i++){
        if(str[i]==' '){
            s++;
        }
    }
    return s;
}
// Функція для відображення рядка і деяких
// додаткових характеристик:
void show(char* str){
    cout<<"Фраза: "<<str<<endl;
    cout<<"Символів: "<<getLength(str)<<endl;
    cout<<"Пробілів: "<<getSpace(str)<<endl;
    for(int k=1;k<=35;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Символьний масив:
    char txt[100]="Вивчаємо мову програмування C++";
```

```
    // Передача аргументом функції символьного масиву:
    show(txt);
    // Передача текстового літерала як аргументу функції:
    show("У C++ є класи й об'єкти");
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання цієї програми:

Результат виконання програми (із лістингу 4.11)

```
Фраза: Вивчаємо мову програмування C++
Символів: 31
Пробілів: 3
-----------------------------------
Фраза: У C++ є класи й об'єкти
Символів: 23
Пробілів: 5
-----------------------------------
```

У програмі описано декілька функцій, призначених для обробки текстів (переданих у функцію через символьний масив чи текстовий літерал). Зокрема, ми описали функцію `getLength()`, що повертає як результат кількість символів у тексті. Функція `getSpace()` повертає як результат кількість пробілів у тексті. Функцію `show()` описано так, що під час виклику відображається текст, переданий їй аргументом, а також через виклик функцій `getLength()` і `getSpace()` відображається інформація про довжину текстового рядка і кількість пробілів у ньому. Усім трьом функціям як аргумент передається вказівник на символьне значення. Під цим вказівником маємо на увазі ім'я символьного масиву. Також функціям як аргумент може бути передано безпосередньо текстовий літерал (оскільки текстовий літерал

обробляється через вказівник на перший символ у тексті). У головній функції програми є команди з викликом функції `show()`.

Вказівник як результат функції

Функція як результат може повертати практично все, що завгодно - за виключенням статичного масиву. Зокрема, результатом функції може бути вказівник. У лістингу 4.12 наведено програму, в якій описано функцію `getMax()`. У функцію передається числовий одномірний масив, а як результат функція повертає вказівник на елемент із найбільшим значенням у масиві. Розглянемо поданий нижче програмний код:

Лістинг 4.12. Функція повертає як результат вказівник

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція як результат повертає
// вказівник на елемент масиву:
int* getMax(int* nums,int n){
   int i=0,k;
   // Визначення індексу найбільшого елемента:
   for(k=0;k<n;k++){
      if(nums[k]>nums[i]){
         i=k;
      }
   }
   // Результат функції - вказівник на елемент:
   return nums+i;
}
// Функція для відображення вмісту масиву:
void show(int* nums,int n){
   for(int i=0;i<n;i++){
      cout<<nums[i]<<" ";
```

```
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Розмір масиву:
    const int size=10;
    // Створення масиву:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Відображення вмісту масиву:
    show(numbers,size);
    // Запис результату функції у вказівник:
    int* maxPnt=getMax(numbers,size);
    // Відображення максимального значення:
    cout<<"Максимальне значення: "<<*maxPnt<<endl;
    // Присвоювання значення елементу:
    *maxPnt=-100;
    // Відображення вмісту масиву:
    show(numbers,size);
    // Значення присвоюється змінній:
    int maxNum=*getMax(numbers,size);
    // Перевірка максимального значення:
    cout<<"Максимальне значення: "<<maxNum<<endl;
    // Присвоювання значення змінній:
    maxNum=-200;
    // Перевірка вмісту масиву:
    show(numbers,size);
    cout<<"Максимальне значення: ";
    // Обчислення нового найбільшого значення:
    cout<<*getMax(numbers,size)<<endl;
```

```
    cout<<"Індекс елемента: ";
    // Обчислення індексу елемента з найбільшим значенням:
    cout<<getMax(numbers,size)-numbers<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У результаті виконання програми отримаємо таке:

Результат виконання програми (із лістингу 4.12)

```
1 5 8 2 4 9 11 9 12 3
Максимальне значення: 12
1 5 8 2 4 9 11 9 -100 3
Максимальне значення: 11
1 5 8 2 4 9 11 9 -100 3
Максимальне значення: 11
Індекс елемента: 6
```

Ідентифікатором типу результату для функції `getMax()` вказано інструкцію `int*`, яка означає, що результатом функції є вказівник на цілочислове значення. У тілі функції цілочислова змінна `i` ініціалізована з початковим нульовим значенням. Далі запускається оператор циклу (індексна змінна `k` пробігає всі можливі значення для індексів масиву), у якому послідовно порівнюються значення елементів масиву зі значенням елемента, що має індекс `i`. Якщо виконано умову `nums[k]>nums[i]`, то значення індексу `k` присвоюється змінній `i`. У результаті після перебору всіх елементів масиву в змінну `i` буде записано індекс елемента з найбільшим значенням (або індекс першого елемента з найбільшим значенням, якщо таких елементів декілька). Як результат функції повертається значення `nums+i`. Тут ми скористалися правилами адресної арифметики і тим, що ім'я масиву є вказівником на його перший елемент.

У функції `main()` створюється цілочислова константа `size`, що визначає розмір масиву, а сам масив створюється командою `int numbers[size]={1,5,8,2,4,9,11,9,12,3}`. Відображення вмісту масиву здійснюється за допомогою спеціально описаної для такої мети функції `show()`.

Під час виконання команди `int* maxPnt=getMax(numbers,size)` у вказівник `maxPnt` записується адреса того елемента в масиві `number`, який має найбільше значення. Дізнатися це значення можемо через вираз `*maxPnt`. При цьому `maxPnt` є адресою (але не індексом!) цього елемента. Щоб дізнатися індекс елемента, необхідно відняти від вказівника на цей елемент вказівник на перший елемент масиву (ім'я масиву).

Після виконання команди `*maxPnt=-100` елемент масиву, що мав до цього найбільше значення, отримає нове значення `-100`.

Якщо нас цікавить значення елемента, а не його адреса, то під час виклику функції `getMax()` можна вказати перед її іменем зірочку `*` (що означає отримання значення за адресою), як робиться в інструкції `int maxNum=*getMax(numbers,size)`. У результаті в змінну `maxNum` записується значення найбільшого елемента, але при цьому інформація про адресу елемента не запам'ятовується. Тому, якщо ми потім присвоїмо змінній `maxNum` нове значення, на вихідному масиві це ніяк не позначиться. Нарешті, для обчислення значення індексу елемента з найбільшим значенням обчислюємо вираз `getMax(numbers,size)-numbers` (різниця вказівників дає ціле число, що визначає кількість комірок між відповідними областями пам'яті - для елементів масиву отримане таким чином значення збігається з індексом елемента).

Посилання як результат функції

Функція як результат може повертати посилання. Якщо функція повертає посилання, то її результатом є не просто значення деякого виразу або змінної, а фактично сама змінна.

До уваги

Зрозуміло, що в таких випадках мова не може йти про локальну змінну - нема сенсу давати доступ до змінної, яка існує, тільки поки виконується функція.

Невелику варіацію на тему попереднього прикладу, але цього разу з поверненням функцією посилання замість вказівника, наведено в лістингу 4.13.

Лістинг 4.13. Функція повертає як результат посилання

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція як результат повертає
// посилання на елемент масиву:
int &getMax(int* nums,int n){
   int i=0,k;
   // Визначення індексу найбільшого елемента:
   for(k=0;k<n;k++){
      if(nums[k]>nums[i]){
         i=k;
      }
   }
   // Результат функції - посилання на елемент:
   return nums[i];
}
// Функція для відображення вмісту масиву:
void show(int* nums,int n){
   for(int i=0;i<n;i++){
      cout<<nums[i]<<" ";
```

```
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Розмір масиву:
    const int size=10;
    // Створення масиву:
    int numbers[size]={1,5,8,2,4,9,11,9,12,3};
    // Відображення вмісту масиву:
    show(numbers,size);
    // Запис результату функції у змінну:
    int maxNum=getMax(numbers,size);
    // Відображення максимального значення:
    cout<<"Максимальне значення: "<<maxNum<<endl;
    // Присвоювання значення змінній:
    maxNum=-100;
    // Відображення вмісту масиву:
    show(numbers,size);
    // Результат функції записується в посилання:
    int &maxRef=getMax(numbers,size);
    // Перевірка максимального значення:
    cout<<"Максимальне значення: "<<maxRef<<endl;
    // Присвоювання значення посиланню:
    maxRef=-200;
    // Перевірка вмісту масиву:
    show(numbers,size);
    cout<<"Максимальне значення: ";
    // Обчислення нового найбільшого значення:
    cout<<getMax(numbers,size)<<endl;
```

```
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 4.13)

```
1 5 8 2 4 9 11 9 12 3
Максимальне значення: 12
1 5 8 2 4 9 11 9 12 3
Максимальне значення: 12
1 5 8 2 4 9 11 9 -200 3
Максимальне значення: 11
```

Поглянемо, що принципово змінилося в програмному коді (порівняно з лістингом 4.12.). У першу чергу, звичайно, це результат функції `getMax()`: тепер функція повертає не вказівник, а посилання, про що свідчить інструкція `&` перед іменем функції в її описі. Оскільки функцію описано з ідентифікатором `int`, то йдеться про посилання на цілочислове значення. У тілі функції, для числового масиву `nums`, переданого у функцію, повертається посилання на елемент з найбільшим значенням. Формально команда повернення функцією результату виглядає як `return nums[i]`, що в принципі означає повернення результатом значення елемента з індексом `i` з масиву `nums`. Але оскільки в цьому випадку в описі функції перед її іменем стоїть інструкція `&`, то повертається не просто значення елемента, а *посилання* на цей елемент. Із прикладної точки зору це твердження означає, що через результат функції `getMax()` ми можемо не лише прочитати значення найбільшого елемента, а й присвоїти йому нове значення.

У функції `main()` створюється цілочисловий масив `number` розміром `size`, і саме з цим масивом виконуються операції з залученням функції

getMax(). Так, під час виконання команди int maxNum=getMax(numbers,size) у змінну maxNum записується значення найбільшого елемента масиву. Подальша зміна значення змінної maxNum (як, наприклад, під час виконання команди maxNum=-100) на масиві numbers не позначається - значення його елементів лишаються незмінними. Але якщо ми запишемо результат функції не у звичайну змінну, а в посилання (як приклад може бути команда int &maxRef=getMax(numbers,size)), то це посилання стане альтернативною назвою для відповідного елемента масиву (того елемента, на який функція повертає посилання). Тому, скажімо, виконання команди maxRef=-200 означає, що відповідний елемент масиву отримує нове значення.

Динамічний масив як результат функції

Теоретично можна описати функцію, яка як результат повертає динамічний масив. Точніше, вона повертає вказівник на перший елемент масиву, а сам масив створюється під час виклику функції. Правда, це досить небезпечний спосіб програмування (чому - пояснюється далі). Разом з тим, деякі специфічні моменти загального характеру він дозволяє продемонструвати, тому розглянемо його.

У лістингу 4.14 наведено програму, у якій описано дві функції, кожна з яких повертає як результат динамічний масив: у випадку функції fibs() – це масив, заповнений числами Фібоначчі, а функція myrand() повертає цілочисловий масив, заповнений випадковими числами в діапазоні значень від 0 до 9.

До уваги

У послідовності чисел Фібоначчі перші два числа дорівнюють одиниці, а кожне наступне число дорівнює сумі двох попередніх. У результаті отримуємо: 1, 1, 2, 3, 5, 8, 13, 21, 34, 55 і так далі.

Кожній із цих функцій під час виклику аргументом передається ціле число, яке визначає розмір масиву. Одна від одної функції принципово відрізняються тільки способом заповнення елементів, а в іншому вони практично ідентичні.

Розглянемо наступний програмний код:

Лістинг 4.14. Функція повертає як результат динамічний масив

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Результат функції - динамічний масив
// із числами Фібоначчі:
int* fibs(int n){
   int* nums=new int[n];
   for(int i=0;i<n;i++){
      if(i==0||i==1){
         nums[i]=1;
      }
      else{
         nums[i]=nums[i-1]+nums[i-2];
      }
   }
   return nums;
}
// Результат функції - динамічний масив
// із випадковими числами:
int* myrand(int m){
   int* nums=new int[m];
   for(int i=0;i<m;i++){
      nums[i]=rand()%10;
   }
   return nums;
```

```
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Змінні:
   int n=10,m=15,i;
   // Вказівник на цілочислове значення:
   int* f;
   // Створюється динамічний масив:
   f=fibs(n);
   // Відображення значень елементів
   // динамічного масиву:
   for(i=0;i<n;i++){
      cout<<f[i]<<" ";
   }
   cout<<endl;
   // Видалення динамічного масиву:
   delete [] f;
   // Новий динамічний масив:
   f=myrand(m);
   // Відображення вмісту масиву:
   for(i=0;i<m;i++){
      cout<<f[i]<<" ";
   }
   cout<<endl;
   // Видалення динамічного масиву:
   delete [] f;
   // Затримка консольного вікна:
   system("pause>nul");
```

```
    return 0;
}
```

Результат виконання програми, із поправкою на те, що другий масив заповнюється випадковими числами, може виглядати так:

Результат виконання програми (із лістингу 4.14)

```
1 1 2 3 5 8 13 21 34 55
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Проаналізуємо найцікавіші місця програми. Скажімо, функція `fibs()` формально повертає вказівник на ціле число (її ідентифікатор типу визначено як `int*`). У тілі функції командою `int* nums=new int[n]` створюється динамічний масив (через `n` позначено цілочисловий аргумент функції). Масив заповнюється за допомогою оператора циклу. У ньому, у свою чергу, є умовний оператор, де перевіряється значення індексу елемента (в умові `i==0||i==1` використано оператор `||` *логічного або*, який повертає значення `true`, якщо хоча б один із його операндів дорівнює `true`). Для нульового й одиничного індексу відповідному елементу присвоюється одиничне значення. Для елементів із іншими індексами значення обчислюється як сума значень двох попередніх елементів у масиві (команда `nums[i]=nums[i-1]+nums[i-2]`). Після заповнення масиву значення змінної-вказівника `nums` (посилання на перший елемент динамічного масиву) повертається як результат функції.

Аналогічно організовано функцію `myrand()`, тільки у неї значення елементам динамічного масиву присвоюється з використанням функції генерування випадкових чисел `rand()`.

У функції `main()` оголошується вказівник `f` на цілочислове значення. Потім командою `f=fibs(n)` у вказівник `f` записується адреса першого елемента масиву, який створюється під час виклику функції `fibs()`. Після цього `f` можна розглядати як масив, заповнений числами Фібоначчі. Змінній-вказівнику `f` можна присвоїти й інше значення - наприклад, командою `f=myrand(m)` у `f`

записується адреса першого елемента заповненого випадковими числами масиву, який створюється під час виклику функції `myrand()`. Але перед присвоюванням нового значення вказівнику `f` необхідно виконати команду `delete [] f`, яка видаляє масив із числами Фібоначчі. Якщо цього не зробити, то адреса масиву "загубиться" (її не буде записано в жоден вказівник), і в результаті пам'ять залишиться незвільненою. Власне, в цьому полягає важливий аргумент проти використання підходу, за якого функція повертає результатом динамічний масив. Такий масив створюється під час виклику функції, але після закінчення її роботи не видаляється з пам'яті. Як наслідок може статись, що масив не буде своєчасно видалений з пам'яті.

Описану вище процедуру (із поверненням динамічного масиву результатом функції) зі статичним масивом провести не вийде. Річ у тім, що всі статичні змінні (у тому числі й масиви), які створюються під час виклику функції, після закінчення її виконання автоматично видаляються. Однак можна застосувати іншу тактику, і замість того, щоб намагатися описувати функцію, яка повертає масив, передати у функцію масив через аргументи, а потім внести до цього масиву потрібні зміни. У лістингу 4.15 наведено програму, яка формально нагадує попередню (із лістингу 4.14), тільки тепер масиви, із яким виконуються маніпуляції, передаємо у відповідну функцію через аргументи. Розглянемо поданий нижче програмний код:

Лістинг 4.15. Заповнення статичного масиву

```
#include <iostream>
#include <cstdlib>
using namespace std;
// У функцію передається масив для заповнення
// числами Фібоначчі:
void fibs(int* nums,int n){
    for(int i=0;i<n;i++){
        if(i==0||i==1){
```

```
            nums[i]=1;
        }
        else{
            nums[i]=nums[i-1]+nums[i-2];
        }
    }
}
// У функцію передається масив для заповнення
// випадковими числами:
void myrand(int* nums,int m){
    for(int i=0;i<m;i++){
        nums[i]=rand()%10;
    }
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Ініціалізація генератора випадкових чисел:
    srand(2);
    // Розмір масиву:
    const int n=15;
    // Масив:
    int f[n];
    // Заповнення масиву числами Фібоначчі:
    fibs(f,n);
    // Відображення значень елементів масиву:
    for(int i=0;i<n;i++){
        cout<<f[i]<<" ";
    }
    cout<<endl;
    // Заповнення масиву випадковими числами:
```

```
    myrand(f,n);
    // Відображення вмісту масиву:
    for(int i=0;i<n;i++){
       cout<<f[i]<<" ";
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання функції може бути таким:

Результат виконання програми (із лістингу 4.15)

```
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
5 6 8 5 4 0 0 1 5 6 7 9 2 6 3
```

Тепер функції `fibs()` і `myrand()` не повертають результат, а масив, який треба заповнити відповідно числами Фібоначчі й випадковими числами, передається у функцію (за допомогою двох аргументів: імені масиву та його розміру).

У тілі функції створюється статичний цілочисловий масив `f` визначеного наперед розміру. Потім, до присвоювання значень елементам масиву, ім'я масиву передається як аргумент у функцію `fibs()`. Це коректна команда, оскільки `f` є вказівником на перший елемент масиву, і після створення масиву, навіть якщо він ще не заповнений, у вказівника `f` уже є значення. В іншому команди програми повинні бути зрозумілими читачу.

Вказівник на функцію

Ім'я функції є вказівником на функцію. Цю просту обставину можна використати під час розв'язання значної кількості прикладних задач. Далі розглянемо деякі приклади використання вказівників на функції.

Теорія

Змінну, що є ***вказівником на функцію***, описують таким чином:

- вказують ключове слово, що визначає тип результату, який повертає функція, на яку може посилатися вказівник;
- вказують назву для змінної-вказівника;
- перед іменем змінної-вказівника ставлять зірочку *, а вся конструкція із зірочки й імені вказівника береться у круглі дужки;
- після імені вказівника у круглих дужках перераховують ідентифікатори типів аргументів, які передаються функції, на яку може посилатися вказівник.

У лістингу 4.16 наведено дуже простий приклад використання вказівників на функцію.

Лістинг 4.16. Вказівник на функцію

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функції з двома аргументами (тип double й int),
// що повертають результат типу double:
double f(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=(1+x);
   }
   return s;
}
double g(double x,int n){
   double s=1;
   for(int k=1;k<=n;k++){
      s*=x/k;
   }
   return s;
```

```
}
// Функції з одним аргументом (тип int),
// що повертають результат типу char:
char h(int n){
    return 'A'+n;
}
char u(int n){
    return 'Z'-n;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Змінні для передачі аргументами:
    double x=2;
    int n=3;
    // Вказівники на функції:
    double (*p)(double,int);
    char (*q)(int);
    double (*r)(double);
    // Використання вказівників на функції:
    p=f;
    cout<<"| "<<p(x,n)<<" | ";
    p=g;
    cout<<p(x,n)<<" | ";
    q=h;
    cout<<q(n)<<" | ";
    q=u;
    cout<<q(n)<<" | ";
    r=exp;
    cout<<r(x/2)<<" | ";
    r=log;
```

```
    cout<<r(x)<<" |\n";
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче показано результат виконання програми:

Результат виконання програми (із лістингу 4.16)

```
| 27 | 1.33333 | D | W | 2.71828 | 0.693147 |
```

У програмі описано чотири допоміжні функції:

- функція `f()` для аргументів `x` і `n` повертає значення $(1+x)^n$;
- функція `g()` для аргументів `x` і `n` повертає значення $\frac{x^n}{n!}$;
- функція `h()` для аргументу `n` повертає символ, зміщений у кодовій таблиці символів від символу `'A'` на `n` позицій уперед;
- функція `u()` для аргументу `n` повертає символ, зміщений у кодовій таблиці символів від символу `'Z'` на `n` позицій назад.

У головній функції програми оголошуються три вказівники на функції:

- Вказівник `p` оголошено командою `double (*p)(double,int)`. Як значення цьому вказівнику може присвоюватися ім'я функції, у якої два аргументи (перший типу `double` і другий типу `int`) і яка як результат повертає значення типу `double`. Наприклад, після виконання команди `p=f` вказівник `p` є альтернативним способом звернення до функції `f()`. Тому команда `p(x,n)` означає виклик функції `f()` із аргументами `x` і `n`. Після виконання команди `p=g`, вираз `p(x,n)` означає виклик функції `g()` із аргументами `x` і `n`.
- Вказівник `q` оголошено командою `char (*q)(int)`. Як значення цьому вказівнику може присвоюватися ім'я функції, у якої один аргумент типу `int` і яка як результат повертає значення типу `char`. Наприклад, після виконання команди `q=h` вказівник `q` є альтернативним способом звернення до функції

h(). Команда q(n) означає виклик функції h() із аргументом n. Після виконання команди q=u, вираз q(n) означає виклик функції u() з аргументом n.

• Вказівник r оголошено командою double (*r)(double). Як значення цьому вказівнику може присвоюватися ім'я функції, у якої один аргумент типу double і яка як результат повертає значення типу double. Після виконання команди r=exp (функція для обчислення експоненти) вказівник r є альтернативним способом звернення до вбудованої математичної функції exp(). Команда r(x/2) означає виклик функції exp() із аргументом x/2. Після виконання команди r=log (функція для обчислення натурального логарифма) вираз r(x) означає виклик функції log() із аргументом x.

До уваги

Для використання вбудованих математичних функцій у програмі ми підключаємо заголовок <cmath>.

Щодо обчислюваних значень, то має сенс звернути увагу на те, що при $x = 2$ і $n = 3$ мають місце такі співвідношення: $(1 + x)^n = 3^3 = 27$, $\frac{x^n}{n!} = \frac{2^3}{3!} = \frac{8}{6} = \frac{4}{3} \approx 1.33333$, $\exp\left(\frac{x}{2}\right) = \exp(1) \approx 2.71828$ і $\ln(x) = \ln(2) \approx 0.693147$. Третя буква після букви 'A' - буква 'D', а третя буква перед буквою 'Z' - буква 'W'.

Ще один приклад використання вказівників на функції наведено в лістингу 4.17. Там ми маємо справу з програмою, в якій у числовому вигляді обчислюється визначений інтеграл. Для обчислення інтеграла створюється спеціальна функція, аргументами якій під час виклику передаються вказівник на підінтегральну функцію і числові значення, що визначають межі області інтегрування.

Подробиці

Щоб не ускладнювати собі життя, будемо виходити з того, що визначеним інтегралом $\int_a^b f(x)dx$ від підінтегральної функції $f(x)$ на інтервалі значень від a до b є площа під кривою, що визначається залежністю $f(x)$. Проте

насправді формальне визначення інтеграла в цьому випадку не дуже важливе. Важливо те, що ми будемо обчислювати інтеграл (у наближеному числовому вигляді) за такою формулою (формула трапецій): $\int_a^b f(x)dx \approx \frac{f(a)+f(b)}{2}\Delta x + \Delta x \sum_{k=1}^{n-1} f(a+k\Delta x)$, де введено позначення $\Delta x = \frac{b-a}{n}$, а через n позначено деяке ціле число (чим воно більше, тим вища точність обчислень). Саме наведена вище формула (при заданій функції $f(x)$ і межах a і b) використовується в програмі для обчислення інтеграла.

Розглянемо наведену нижче програму:

Лістинг 4.17. Передача аргументом вказівника на функцію при обчисленні інтеграла

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція для обчислення інтеграла. Першим аргументом
// передається вказівник на функцію:
double integrate(double (*F)(double),double a,double b){
   int n=1000;
   double dx=(b-a)/n;
   double s=(F(a)+F(b))*dx/2;
   for(int k=1;k<=n-1;k++){
      s+=F(a+dx*k)*dx;
   }
   return s;
}
// Підінтегральні функції:
double f(double x){
   return x*(1-x);
}
double g(double x){
   return 1/x;
}
```

```
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Перший інтеграл:
    cout<<integrate(f,0,1)<<endl;
    // Другий інтеграл:
    cout<<integrate(g,1,2)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У результаті виконання програми отримуємо:

Результат виконання програми (із лістингу 4.17)

```
0.166667
0.693147
```

Функція для обчислення інтеграла називається `integrate()`. Вона описана так, що результатом повертає значення типу `double` (значення інтеграла). Її перший аргумент описано як `double (*F)(double)`. Цей аргумент - вказівник на функцію, у якої один `double`-аргумент і яка повертає значення типу `double`. Тому в тілі функції аргумент `F` оброблюється як ім'я функції. Два інших `double`-аргументи `a` і `b` визначають межі області інтегрування.

Локальна змінна `n` визначає кількість інтервалів, на які розбивається область інтегрування, а змінна `dx` зі значенням `(b-a)/n` визначає довжину кожного такого інтервалу. Інтегральна сума записується в змінну `s`, початкове значення якої задано виразом `(F(a)+F(b))*dx/2`. Далі запускається оператор циклу, у якому за кожну чергову ітерацію до значення змінної `s` додається доданок

`F(a+dx*k)*dx`. Після закінчення оператора циклу значення змінної `s` повертається як результат функції.

Також описано дві підінтегральні функції. Функція `f()` визначає залежність виду $f(x) = x(1-x)$, а функція `g()` визначає залежність виду $g(x) = \frac{1}{x}$. У головній функції програми командами `integrate(f,0,1)` і `integrate(g,1,2)` відповідно обчислюються інтеграли $\int_0^1 x(1-x)dx = \frac{1}{6} \approx 0.166667$ і $\int_1^2 \frac{dx}{x} = \ln(2) \approx 0.693147$.

Задачі для самостійного розв'язання

1. Напишіть програму з функцією для обчислення факторіала числа (факторіал числа $n! = 1 \cdot 2 \cdot 3 \cdot \ldots \cdot n$ - добуток натуральних чисел від 1 до цього числа). Запропонуйте різні способи організації коду (наприклад, із рекурсією і без рекурсії).

2. Напишіть програму з функцією для обчислення подвійного факторіала числа (подвійний факторіал числа $n!! = n \cdot (n-2) \cdot (n-4) \cdot \ldots$ - добуток натуральних чисел "через одне число" - останній множник 2 для парних чисел і 1 для непарних чисел). Запропонуйте різні способи організації коду (наприклад, із рекурсією і без рекурсії).

3. Напишіть програму з функцією для обчислення чисел Фібоначчі. Як аргумент функції передається номер числа в послідовності, а як результат повертається саме число. Розглянути спосіб опису функції з використанням рекурсії і без використання рекурсії.

4. Напишіть програму з функцією для обчислення біноміальних коефіцієнтів $C_n^k = \frac{n!}{k!(n-k)!}$. Параметри k і n передаються аргументами функції.

5. Напишіть програму з функцією для обчислення суми натуральних чисел. У функції використайте рекурсію. Перевантажте функцію так, щоб другий аргумент визначав степінь, до якої підносяться доданки під час підсумовування.

Наприклад, якщо як аргументи функції передано значення n і k, то як результат повертається сума $1^k + 2^k + 3^k + \cdots + n^k$ (при $k = 1$ виходить сума натуральних чисел).

6. Напишіть програму з функцією для розрахунку експоненти $\exp(x) \approx 1 + x + \frac{x^2}{2!} + \cdots \frac{x^n}{n!}$. У функції повинно бути два аргументи (параметри x і n), а при її описанні використати рекурсію. Скористатися тим, що якщо $S_n(x) = 1 + x + \frac{x^2}{2!} + \cdots \frac{x^n}{n!}$, то має місце рекурентне співвідношення $S_n(x) = S_{n-1}(x) + \frac{x^n}{n!}$ і крім того $S_0(x) = 1$.

7. Напишіть програму з перевантаженою функцією. Якщо функції передається один числовий аргумент, то вона повертає як результат значення цього аргументу. Якщо функції передається два числових аргументи, то вона повертає як результат суму квадратів цих значень. Якщо функції передається три числових аргументи, то вона повертає як результат суму кубів їхніх значень.

8. Напишіть програму з функцією, у якої такі аргументи: вказівник на функцію і масив числових значень. Під час виклику функції до кожного з елементів масиву застосовується функція, передана першим аргументом (через вказівник). Передбачити перевантаження функції таким чином, щоб як перший аргумент (замість вказівника на функцію) можна було передавати число (у такому випадку всі елементи масиву-аргументу множаться на це число).

9. Напишіть програму з функцією, призначеною для обчислення похідної від деякої функції. Як перший аргумент передається вказівник на диференційовану функцію, а другий аргумент - точка, у якій обчислюється похідна. Для обчислення похідної від певної функції $f(x)$ використайте наближену формулу $f'(x) \approx \frac{f(x+h)-f(x)}{h}$, де через h позначено малий приріст по аргументу (обирається довільно). Передбачити перевантаження функції таке, щоб замість другого числового аргументу можна було передавати масив значень. У такому випадку як аргумент функції передається ще один додатковий масив, у який

записуються значення похідної у кожній із точок, що визначаються елементами першого масиву.

10. Напишіть програму з функцією для обчислення значення полінома в точці. Як аргументи функції передаються масив із числовими значеннями і точка, в якій обчислюється значення полінома. Якщо через a_k позначити елементи масиву, переданого аргументом функції (індекс $k = 0,1,2,\dots,n$), а через x позначити точку, в якій обчислюється значення полінома, то як результат функції повинна повертатися сума $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Перевантажити функцію так, щоб за відсутності аргументу, що визначає точку для обчислення значення полінома, функція в консольне вікно виводила коефіцієнти полінома (значення елементів масиву).

11. Напишіть програму з функцією, аргументом якій передається числовий масив. Під час виклику функції елементи масиву сортуються в порядку зростання, а як результат функція повертає вказівник на останній елемент масиву.

12. Напишіть програму з функцією, аргументом якій передається числовий масив. Під час виклику функції елементи масиву сортуються в порядку спадання, а як результат функція повертає посилання на останній елемент масиву.

13. Напишіть програму з функцією, аргументом якій передається символьний масив з текстом. Як результат функції повертається довжина найдовшого слова в тексті, що міститься в масиві.

14. Напишіть програму з функцією, аргументом якій передається символьний масив із текстом. Під час виклику функції текст, записаний у масив, відображується у зворотному порядку. Спробуйте реалізувати таку функцію через рекурсію.

15. Напишіть програму з функцією, аргументом якій передаються символьний масив і окремий символ. Як результат функції повертається кількість букв, що визначається символьним аргументом, в тексті, який міститься в символьному масиві.

16. Напишіть програму з функцією, аргументом якій передається двомірний числовий масив. Як результат функція повертає посилання на елемент масиву з найбільшим значенням.

17. Напишіть програму з функцією, аргументом якій передається двомірний числовий масив. Як результат функція повертає вказівник на елемент масиву з найменшим значенням.

18. Напишіть програму з функцією, аргументом якій передається двомірний числовий масив. Як результат функція повертає середнє арифметичне для елементів масиву (сума всіх елементів, поділена на їхню кількість).

19. Напишіть програму з функцією, аргументом якій передається символ і двомірний символьний масив (попередньо заповнений випадковими символами). Як результат функція повертає кількість входжень символу у двомірний масив.

20. Напишіть програму з функцією, аргументом якій передається символ і двомірний символьний масив, у кожний рядок якого записано текст. Як результат функцією повертається кількість входжень символу у двомірний масив (на відміну від попередньої задачі, у цьому випадку перевіряти потрібно не всі елементи двомірного символьного масиву, а лише ті, що містять текст).

Глава 5

Класи й об'єкти

- Ну що, ти бачиш Туреччину?

- Ні, поки що не бачу.

- А зараз?

з х/ф "Пропала грамота"

У цій главі ми познайомимося з азами об'єктно-орієнтованого програмування (скорочено ООП), зокрема:

- навчимося описувати класи і створювати об'єкти;
- розглянемо режими доступу до полів і методів класу;
- навчимося використовувати конструктори й деструктори;
- дізнаємося, що таке перевантаження методів;
- познайомимося з успадкуванням;
- дізнаємося, як виконується перевантаження операторів,

а також зупинимося на деяких суміжних питаннях, пов'язаних із використанням класів і об'єктів.

Знайомство з класами й об'єктами

Розглянемо вже знайому нам задачу про розміщення на депозиті певної грошової суми, на яку нараховуються відсотки, але цього разу звернемося за допомогою до *класів і об'єктів*.

Подробиці



Нагадаємо, що якщо на депозиті розміщується сума m на час (у роках) t під r відсотків річних, то кінцева сума за таким вкладом обчислюється за формулою $m\left(1+\frac{r}{100}\right)^t$.

Розглянемо програмний код, наведений у лістингу 5.1.

Лістинг 5.1. Опис класу і створення об'єкта

```
#include <iostream>
```

```
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyMoney{
public:
   // Поля:
   string name;
   double money;
   double rate;
   int time;
   // Методи:
   double getMoney(){
      double s=money;
      for(int k=1;k<=time;k++){
         s*=(1+rate/100);
      }
      return s;
   }
   void showAll(){
      cout<<"Ім'я: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
      cout<<"Період (років): "<<time<<endl;
      cout<<"Кінцева сума: "<<getMoney()<<endl;
   }
};
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Створення об'єкта:
```

```
    MyMoney obj;
    // Присвоювання значень полям:
    obj.name="Петренко Петро Петрович";
    obj.money=1000;
    obj.rate=8;
    obj.time=5;
    // Виклик методу:
    obj.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Найбільш важлива частина програми пов'язана з описом класу, який називається `MyMoney`.

Теорія

Клас є загальною схемою чи шаблоном, на основі якого потім створюються ***об'єкти***. Клас містить оголошення ***полів*** і ***методів***, які називаються ***членами*** класу. Поле класу аналогічне змінній, а метод - аналог функції. Під час створення на основі класу об'єктів, останні отримують персональний набір полів і методів - строго у відповідності з описом класу. Поля об'єктів мають значення, а методи виконують деякі дії та можуть повертати результат. Фактично, поля й методи - це відповідно змінні й функції, але тільки строго "прикріплені" до певного об'єкту. Різні об'єкти, створені на основі одного класу, мають однакові набори полів і методів, при цьому значення полів у кожного об'єкта свої, а методи мають автоматичний доступ до об'єктів, із яких викликаються. Немає сенсу говорити про значення поля чи результат виклику методу, якщо не ідентифіковано об'єкт, до якого вони належать.

Опис класу починається з ключового слова `class`, після чого вказують ім'я класу, а потім у фігурних дужках описують тіло класу (в кінці опису стоїть крапка з комою).

У тілі класу описуються *поля й методи класу*. Але перед початком опису йде ключове слово `public` (після якого стоїть двокрапка). Це ключове слово

ідентифікує блок відкритих членів класу - таких, доступ до яких є не тільки в межах класу, але й із зовнішнього коду.

Полів у класу `MyMoney` чотири:

- поле `name` типу `string` (текстове значення) призначено для запису імені вкладника;
- поле `money` типу `double` призначено для запису початкової суми вкладу;
- поле `rate` типу `double` призначено для запису показника річної відсоткової ставки;
- цілочислове (тип `int`) поле `time` призначено для запису кількості років, на які розміщується депозит.

До уваги

Текстове поле `name` у даному випадку реалізовано у вигляді об'єкта вбудованого класу `string`, призначеного для роботи з текстом. Для використання класу `string` у програмі підключаємо заголовок `<string>`.

Окрім полів, у класу `MyMoney` є ще два методи. Метод `getMoney()` призначено для обчислення кінцевої суми по вкладу. Метод повертає значення типу `double` (кінцева сума по вкладу) і у нього немає аргументів.

Подробиці

Для обчислення кінцевої суми по вкладу необхідно знати три параметри: початкову суму вкладу, річну ставку відсотка і кількість років, на які розміщується депозит. Значення цих параметрів метод `getMoney()` отримує через поля `money`, `rate` і `time` об'єкта, з якого викликається. Тому немає необхідності передавати методу `getMoney()` аргументи.

У тілі метода `getMoney()` оголошується локальна змінна `s` зі значенням `money` (тобто під час виклику методу локальній змінній `s` присвоюється значення поля `money` того об'єкта, з якого викликається метод). Далі запускається оператор циклу, в якому індексна змінна пробігає значення в діапазоні від `1` до значення поля `time` (знову ж, тут мається на увазі поле об'єкта, з якого викликається метод). За кожний цикл виконується команда

`s*=(1+rate/100)`, якою поточне значення змінної `s` множиться на значення `(1+rate/100)` (тут `rate` - поле об'єкта). Після закінчення виконання оператора циклу значення змінної `s` повертається як результат методу `getMoney()`.

До уваги

Таким чином, коли метод `getMoney()` викликається з об'єкта, то він звертається до полів цього об'єкта для отримання тих значень, які необхідні під час виконання розрахунків. Тому якщо метод `getMoney()` викликається з різних об'єктів, то в загальному випадку отримуємо і різні результати.

Метод `showAll()` не повертає результат (ідентифікатор типу `void` на початку опису методу), і у нього немає аргументів. Під час виклику методу в консольному вікні відображаються значення всіх чотирьох полів об'єкта, з якого викликається метод.

Власне, на цьому опис класу `MyMoney` закінчується. Далі розглянемо команди в головній функції програми. Там, по-перше, створюється об'єкт `obj` класу `MyMoney`. Робиться це дуже просто - за допомогою команди `MyMoney obj`. Фактично, об'єкт створюється так само, як оголошуються звичайні змінні, тільки під час створення об'єкта як ідентифікатор типу вказують ім'я класу. Проте створення об'єкта ще не означає, що його поля отримали значення. Під них лише виділено місце, а значення доводиться присвоювати "вручну". Звернення до полів об'єкта виконується так: вказується ім'я об'єкта і через крапку після імені об'єкта вказується назва поля. У всьому іншому з полями допускається маніпулювати так само, як і зі звичайними змінними. Зокрема, командами `obj.name="Петренко Петро Петрович"`, `obj.money=1000`, `obj.rate=8` і `obj.time=5` полям об'єкта `obj` присвоюються значення. Потім командою `obj.showAll()` із об'єкта `obj` викликається метод `showAll()`. Отже, результат виконання програми буде такий:

Результат виконання програми (із лістингу 5.1)

```
Ім'я: Петренко Петро Петрович
Вклад: 1000
Ставка (%): 8
Період (років): 5
Кінцева сума: 1469.33
```

Як бачимо, використання класу й об'єкта багато в чому виявилося зручним - може лише за винятком процедури присвоювання значень полям об'єкта. Проте, як ми з'ясуємо далі, у цієї проблеми є рішення (і навіть декілька).

Відкриті та закриті члени класу

Внесемо деякі зміни у програмний код, розглянутий у попередньому прикладі. Зокрема, для присвоювання значень полям створимо спеціальний метод, а самі поля й метод для обчислення кінцевої суми за депозитом зробимо *закритими*.

Теорія

До закритих полів і методів можна звертатися тільки в програмному коді в тілі класу. Закриті поля й методи описуються в блоці, поміченому ключовим словом `private` (або взагалі без ключового слова: за замовчуванням, якщо члени класу явно не ідентифіковані ключовим словом `public` як відкриті, їх вважають закритими).

У попередньому прикладі ми могли в головній функції в явному вигляді присвоювати значення полям об'єкта саме тому, що поля були описані як відкриті. У наступному прикладі, як зазначалося, ми всі поля зробимо закритими, тому присвоїти їм значення явним чином після того, як створено об'єкт, уже не вийде. Для цієї мети ми описуємо метод `setAll()` із чотирма аргументами, що визначають значення, які слід присвоїти полям об'єкта. Зрозуміло, що сам метод `setAll()` повинен бути відкритим. Розглянемо програмний код, наведений у лістингу 5.2.

Лістинг 5.2. Закриті та відкриті члени класу

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyMoney{
private:   // Закриті члени класу
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Відкриті члени класу
   void showAll(){
      cout<<"Ім'я: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
      cout<<"Період (років): "<<time<<endl;
      cout<<"Кінцева сума: "<<getMoney()<<endl;
   }
   void setAll(string n,double m,double r,int t){
      name=n;
      money=m;
      rate=r;
```

```
        time=t;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єктів:
    MyMoney objA,objB;
    // Присвоювання значень полям:
    objA.setAll("Петренко Петро Петрович",1000,8,5);
    objB.setAll("Федоренко Федір Федорович",1200,7,4);
    // Відображення значень полів:
    objA.showAll();
    cout<<endl;
    objB.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 5.2)

```
Ім'я: Петренко Петро Петрович
Вклад: 1000
Ставка (%): 8
Період (років): 5
Кінцева сума: 1469.33

Ім'я: Федоренко Федір Федорович
Вклад: 1200
Ставка (%): 7
```

```
Період (років): 4
Кінцева сума: 1572.96
```

У тілі метода `setAll()` значення аргументів присвоюються полям об'єкта, з якого метод викликають. Тому для заповнення полів значеннями з відповідного об'єкта слід викликати метод `setAll()` і передати йому аргументами значення для полів `name`, `money`, `rate` і `time`.

У головній функції тепер створюємо два об'єкти (`objA` та `objB`), для чого використано команду `MyMoney objA,objB`. Для присвоювання значень полям об'єкта `objA` використовуємо команду `objA.setAll("Петренко Петро Петрович",1000,8,5)`. Команду `objB.setAll("Федоренко Федір Федорович",1200,7,4)` використано для присвоювання значень полям об'єкта `objB`. Перевірити значення полів кожного з об'єктів допомагають команди `objA.showAll()` і `objB.showAll()`.

До уваги

У методі `selAll()` перший аргумент описано як такий, що належить до типу `string` (насправді це об'єкт класу `string`). Текстовий літерал реалізується за допомогою символьного масиву. Тобто в принципі типи різні. Проте за рахунок автоматичного перетворення типів ми можемо передавати текстові літерали як аргумент, який насправді повинен належати до типу `string`.

Перевантаження методів

Аналізуючи попередню програму, легко помітити, що спосіб присвоювання значень полям за допомогою спеціального методу хоч і спрощує процес використання об'єктів, але деякі "тонкі" місця в ньому все одно залишаються. Скажімо, може знадобитися змінити значення не всіх полів, а тільки деяких. У цьому випадку в нагоді може стати *перевантаження методів*.

Теорія

Під час ***перевантаження методів***, як і під час перевантаження функцій, описується декілька версій методу з однією і тією ж назвою. Такі "версії" повинні чимось

відрізнятися: кількістю аргументів, їхніми типами, типом результату, який повертається методом.

У лістингу 5.3 наведено версію програми, призначеної для обчислення кінцевої суми за депозитами, у якій, як і раніше, описується клас `MyMoney`, проте тепер у цьому класі перевантажується метод `setAll()`, завдяки чому для зміни значень полів об'єкта класу `MyMoney` немає необхідності вказувати значення всіх аргументів - можна обмежитися тільки деякими з них.

Лістинг 5.3. Перевантаження методів

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyMoney{
private:   // Закриті члени класу
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Відкриті члени класу
   void showAll(){
      cout<<"Ім'я: "<<name<<endl;
      cout<<"Вклад: "<<money<<endl;
      cout<<"Ставка (%): "<<rate<<endl;
```

```
    cout<<"Період (років): "<<time<<endl;
    cout<<"Кінцева сума: "<<getMoney()<<endl;
}
// Версія з чотирма аргументами:
void setAll(string n,double m,double r,int t){
    name=n;
    money=m;
    rate=r;
    time=t;
}
// Версія з трьома аргументами:
void setAll(double m,double r,int t){
    money=m;
    rate=r;
    time=t;
}
// Версія з текстовим аргументом:
void setAll(string n){
    name=n;
}
// Версія з цілочисловим аргументом:
void setAll(int t){
    time=t;
}
// Версія з двома аргументами типу double:
void setAll(double m,double r){
    money=m;
    rate=r;
}
// Версія з двома аргументами (типу double і типу bool):
void setAll(double x,bool s=true){
    if(s){
```

```
            money=x;
        }
        else{
            rate=x;
        }
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єкта:
    MyMoney obj;
    // Присвоювання значень полям:
    obj.setAll("Петренко Петро Петрович",1000,8,5);
    obj.showAll();
    cout<<endl;
    // Зміна імені:
    obj.setAll("Федоренко Федір Федорович");
    obj.showAll();
    cout<<endl;
    // Зміна терміну вкладу:
    obj.setAll(10);
    obj.showAll();
    cout<<endl;
    // Зміна початкової суми:
    obj.setAll(1200.0);
    obj.showAll();
    cout<<endl;
    // Зміна початкової суми:
    obj.setAll(1500,true);
    obj.showAll();
```

```
    cout<<endl;
    // Зміна відсоткової ставки:
    obj.setAll(6,false);
    obj.showAll();
    cout<<endl;
    // Зміна початкової суми, відсоткової ставки і
    // терміну розміщення депозиту:
    obj.setAll(1000,8,5);
    obj.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 5.3)

```
Ім'я: Петренко Петро Петрович
Вклад: 1000
Ставка (%): 8
Період (років): 5
Кінцева сума: 1469.33

Ім'я: Федоренко Федір Федорович
Вклад: 1000
Ставка (%): 8
Період (років): 5
Кінцева сума: 1469.33

Ім'я: Федоренко Федір Федорович
Вклад: 1000
Ставка (%): 8
Період (років): 10
```

```
Кінцева сума: 2158.92

Ім'я: Федоренко Федір Федорович
Вклад: 1200
Ставка (%): 8
Період (років): 10
Кінцева сума: 2590.71

Ім'я: Федоренко Федір Федорович
Вклад: 1500
Ставка (%): 8
Період (років): 10
Кінцева сума: 3238.39

Ім'я: Федоренко Федір Федорович
Вклад: 1500
Ставка (%): 6
Період (років): 10
Кінцева сума: 2686.27

Ім'я: Федоренко Федір Федорович
Вклад: 1000
Ставка (%): 8
Період (років): 5
Кінцева сума: 1469.33
```

Порівняно з попереднім прикладом (див. лістинг 5.2), ми додали декілька додаткових версій методу `setAll()`, і змінили у зв'язку з цим код головної функції програми - таким чином, що там тестуються різні способи виклику методу `getAll()`. Більш конкретно, окрім версії методу `setAll()` із чотирма аргументами, є ще й такі способи передачі аргументів методу:

• Якщо метод викликано з трьома числовими аргументами, то ці аргументи визначають значення полів `money`, `rate` і `time`.

• Під час передачі методу як аргументу текстового значення воно присвоюється полю `name` об'єкта. Тут незайвим буде зауважити, що насправді текстовий літерал реалізується у вигляді символьного масиву. Однак шляхом автоматичного перетворення типу текстовий літерал приводиться до текстового типу `string`.

• Під час виклику методу з одним цілочисловим аргументом він задає нове значення для поля `time`.

• Під час виклику методу з двома числовими аргументами, вони визначають значення полів `money` і `rate` відповідно.

• Також в класі `MyMoney` описано версію методу `getAll()` з двома аргументами: перший типу `double` і другий типу `bool` (логічний тип). Причому другий аргумент має значення за замовчуванням `true`. Якщо значення другого аргументу дорівнює `true`, тоді перший числовий аргумент є значенням для поля `money`. Якщо другий аргумент дорівнює `false`, то перший числовий аргумент є значенням для поля `rate`.



До уваги

Таким чином, якщо метод `setAll()` викликається з одним цілочисловим аргументом, то в такому випадку значення присвоюється полю `time`. Якщо метод `setAll()` викликається з одним аргументом типу `double`, то значення присвоюється полю `money`. Тому, наприклад, у команді `obj.setAll(1200.0)` числовий аргумент `1200.0` вказано з десятковою крапкою, оскільки в протилежному випадку він би був цілочисловим, і значення присвоювалося би полю `time`. Оскільки аргумент у методі відноситься до типу `double`, то викликається версія методу з двома аргументами, перший із яких типу `double`, а для другого логічного аргументу використовується значення за замовчуванням `true`. А ось в інструкції `obj.setAll(1500,true)` значення другого логічного аргументу вказано явно

(хоча воно і збігається зі значенням за замовчуванням), а перший аргумент є цілим числом. У такій ситуації перший цілочисловий аргумент автоматично приводиться до типу `double`. Під це ж правило підпадає команда `obj.setAll(6,false)`. Однак оскільки в ній значення другого логічного аргументу дорівнює `false`, то числове значення присвоюється полю `rate`.

Знайомство з конструкторами і деструкторами

У програмному коді, що розглядається далі, клас `MyMoney` поповнює свій "арсенал" за рахунок *конструктора* і *деструктора*.

Теорія

Конструктор - метод, який автоматично викликається під час створення об'єкта. Назва конструктора збігається з назвою класу, конструктор не повертає результат і описується без ідентифікатора типу результату. У конструктора можуть бути аргументи, і конструктор можна перевантажувати (у класі допускається описувати декілька версій конструктора).

Деструктор - метод, який автоматично викликається під час видалення об'єкта з пам'яті. Деструктор не повертає результат, описується без ідентифікатора типу результату, у нього немає аргументів, і він не перевантажується (у класі може бути тільки один деструктор). Ім'я деструктора формується об'єднанням ~ і назви класу.

Програмний код, що нас цікавить, наведено у лістингу 5.4. Для зменшення обсягу коду ми в класі `MyMoney` описуємо тільки одну версію методу `setAll()` (версія з чотирма аргументами).

Лістинг 5.4. Конструктори і деструктори

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
```

```
class MyMoney{
private:   // Закриті члени класу
   string name;
   double money;
   double rate;
   int time;
   double getMoney(){
      double s=money;
         for(int k=1;k<=time;k++){
            s*=(1+rate/100);
         }
      return s;
   }
public:   // Відкриті члени класу
   // Конструктор без аргументів:
   MyMoney(){
      name="Пан Коцький";
      money=100;
      rate=5;
      time=1;
      cout<<"Створено новий об'єкт:\n";
      showAll();
   }
   // Конструктор з чотирма аргументами:
   MyMoney(string n,double m,double r,int t){
      setAll(n,m,r,t);
      cout<<"Створено новий об'єкт:\n";
      showAll();
   }
   // Деструктор:
   ~MyMoney(){
      cout<<"Об'єкт для \""<<name<<"\" видалено\n";
```

```
        for(int k=1;k<=35;k++){
            cout<<"*";
        }
        cout<<endl;
    }
    // Методи класу:
    void showAll(){
        cout<<"Ім'я: "<<name<<endl;
        cout<<"Вклад: "<<money<<endl;
        cout<<"Ставка (%): "<<rate<<endl;
        cout<<"Період (років): "<<time<<endl;
        cout<<"Кінцева сума: "<<getMoney()<<endl;
        for(int k=1;k<=35;k++){
            cout<<"-";
        }
        cout<<endl;
    }
    void setAll(string n,double m,double r,int t){
        name=n;
        money=m;
        rate=r;
        time=t;
    }
};
// Функція:
void foxfellow(){
    // Створення локального об'єкта:
    MyMoney objD("Лис Микита",200,3,2);
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
```

```
    system("chcp 1251>nul");
    // Створення об'єктів:
    MyMoney objA;
    MyMoney objB("Дональд Дак",1500,8,7);
    // Виклик функції:
    foxfellow();
    // Створення динамічного об'єкта:
    MyMoney* objC=new MyMoney("Вінні Пух",1200,6,9);
    cout<<"Всі об'єкти створено\n";
    // Видалення динамічного об'єкта:
    delete objC;
    cout<<"Виконання програми закінчено\n";
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 5.4)

```
Створено новий об'єкт:
Ім'я: Пан Коцький
Вклад: 100
Ставка (%): 5
Період (років): 1
Кінцева сума: 105
-----------------------------------
Створено новий об'єкт:
Ім'я: Дональд Дак
Вклад: 1500
Ставка (%): 8
Період (років): 7
```

```
Кінцева сума: 2570.74
----------------------------------
Створено новий об'єкт:
Ім'я: Лис Микита
Вклад: 200
Ставка (%): 3
Період (років): 2
Кінцева сума: 212.18
----------------------------------
Об'єкт для "Лис Микита" видалено
**********************************
Створено новий об'єкт:
Ім'я: Вінні Пух
Вклад: 1200
Ставка (%): 6
Період (років): 9
Кінцева сума: 2027.37
----------------------------------
Всі об'єкти створено
Об'єкт для "Вінні Пух" видалено
**********************************
Виконання програми закінчено

Об'єкт для "Дональд Дак" видалено
**********************************
Об'єкт для "Пан Коцький" видалено
**********************************
```

Нас у програмному коді цікавлять у першу чергу дві версії конструктора і деструктор, описані в класі `MyMoney`. У класі описано дві версії конструктора: без аргументів і з чотирма аргументами. У версії конструктора без аргументів командами `name="Пан Коцький"`, `money=100`, `rate=5` і `time=1` полям

об'єкта присвоюються значення. Потім командою `cout<<"Створено новий об'єкт:\n"` у консольне вікно виводиться відповідне повідомлення і, нарешті, командою `showAll()` зі створюваного об'єкта викликається метод, завдяки чому відомості про значення полів об'єкта з'являються в консольному вікні. Ще раз підкреслимо, що всі ці дії виконуються під час створення об'єкта, якщо для його створення застосовують конструктор без аргументів.

До уваги

Ми дещо вдосконалили метод `showAll()`, так що тепер під час його виклику, після того, як відображено інформацію про об'єкт, з'являється сформована з дефісів імпровізована горизонтальна лінія. Передбачається, що це дозволить краще візуалізувати результат виконання програми.

Також у класі `MyMoney` описано конструктор із чотирма аргументами. У такому випадку викликається метод `setAll()`, і йому передаються ті ж аргументи, що були передані конструктору. Таким чином, аргументи конструктора в цьому випадку визначають значення, які присвоюються полям створюваного об'єкта.

Після заповнення полів командою `cout<<"Створено новий об'єкт:\n"` відображається повідомлення про створення об'єкта, після чого викликається метод `showAll()`.

У деструкторі командою `cout<<"Об'єкт для \""<<name<<"\" видалено\n"` виводиться у консоль повідомлення про видалення об'єкта (у повідомленні в подвійних лапках указують значення поля `name` об'єкта, що видаляється). Після цього "підводиться лінія" із символів * (для чого використано оператор циклу).

Подробиці

Щоб відобразити подвійні лапки в тексті, перед ними ставиться похила риска. Простіше кажучи, якщо в текст необхідно вставити подвійні лапки, то використовується інструкція `\"`. Застосування в такій ситуації самих лише

лапок (без похилої риски) призведе до помилки, оскільки подвійні лапки використовуються для ідентифікації текстових літералів.

Крім класу `MyMoney` у програмі описано функцію `foxfellow()`. Функція не повертає результат, і в неї немає аргументів. Код функції дуже простий: у тілі функції командою `MyMoney objD("Лис Микита",200,3,2)` створюється об'єкт класу `MyMoney`.

У функції `main()` створюються декілька об'єктів класу `MyMoney` і викликається функція `foxfellow()`. Правда, об'єкти створюються по-різному. Один із об'єктів створюється командою `MyMoney objA`. Тут ідеться про статичний об'єкт `objA`, під час створення якого викликається конструктор без аргументів. А ось під час створення об'єкта `objB` використано команду `MyMoney objB("Дональд Дак",1500,8,7)`, у якій після імені створюваного об'єкта у круглих дужках вказано значення, що передаються конструктору. Як нескладно здогадатися, об'єкт `objB` створюється з викликом конструктора з чотирма аргументами.

Подробиці



Рішення про те, яку саме версію конструктора слід викликати під час створення об'єкта, приймається виходячи зі списку аргументів, які передаються конструктору (аргументи, вказані після імені об'єкта в команді, якою створюється об'єкт).

Також треба мати на увазі, що поки в класі не описано жодного конструктора, то під час створення об'єкта використовується так званий ***конструктор за замовчуванням***, у якого немає аргументів, і який жодних додаткових дій з об'єктом не виконує. Якщо в класі описано хоча б один конструктор, то конструктор за замовчуванням більше не доступний.

Варто зауважити, що як під час створення об'єкта `objA`, так і під час створення об'єкта `objB` внаслідок виклику конструктора (різного для різних об'єктів) у самому конструкторі викликається метод `showAll()`. Тому саме тільки створення об'єкта спричинює появу в консольному вікні повідомлення відповідного змісту з інформацією про значення полів створеного об'єкта. Це ж

зауваження стосується об'єкта, який створюється під час виклику функції `foxfellow()`. Однак тепер ідеться про локальний об'єкт. Локальний об'єкт, як і локальна змінна, існує, поки виконується функція. Тому коли викликається функція `foxfellow()`, і починає виконуватися її код, створюється об'єкт `objD`, на чому виконання функції закінчується. Але під час закінчення виконання функції з пам'яті видаляється локальний об'єкт `objD`. Як наслідок, виклик функції `foxfellow()` спричинює виклик конструктора класу `MyMoney` (із чотирма аргументами) і практично відразу призводить до виклику деструктора. Під час виклику деструктора з'являється повідомлення про видалення об'єкта.

Також у програмі створюється динамічний об'єкт, для чого ми використовуємо команду `MyMoney* objC=new MyMoney("Вінні Пух",1200,6,9)`. Тут інструкція `MyMoney* objC` оголошує вказівник `objC` на об'єкт класу `MyMoney`. Інструкція `new MyMoney("Вінні Пух",1200,6,9)` створює динамічний об'єкт (шляхом динамічного виділення пам'яті під нього). Адреса об'єкта записується у вказівник `objC`. Під час створення динамічного об'єкта викликається конструктор із чотирма аргументами.

Після створення динамічного об'єкта, командою `cout<<"Всі об'єкти створено\n"` відображається повідомлення про закінчення процесу створення об'єктів, і командою `delete objC` динамічний об'єкт видаляється. Під час видалення об'єкта викликається деструктор. Потім командою `cout<<"Виконання програми закінчено\n"` відображається повідомлення про закінчення виконання програми (і виконується додатковий пропуск рядка за допомогою команди `cout<<endl`).

Однак насправді на цьому виконання програми не закінчується. Після того, як користувач натисне клавішу <Enter> (або іншу клавішу), у консольному вікні з'являться повідомлення про видалення об'єктів зі значеннями

"Дональд Дак" і "Пан Коцький" поля name. Правда, відбудеться це дуже швидко, і консольне вікно відразу після цього закриється.

До уваги

Щоб побачити результат виконання програми "у всій красі" можна порекомендувати (під час роботи з операційною системою Windows) скористатися командою **Виконати** в головному меню і запустити через цю утиліту відкомпільований exe-файл проекту. Також можна під час роботи з середовищем Visual Studio Express запустити програму за допомогою команди **Start Without Debugging** з меню **Debug** (або запустити програму, натиснувши <Ctrl>+<F5>).

Причина в тому, що під час закінчення виконання програми (тобто під час закінчення виконання функції main()) із пам'яті видаляються об'єкти, які було створено у функції main(). Об'єкти видаляються в порядку, зворотному до того, у якому вони створювалися. Тому спочатку видаляється об'єкт objB, а після цього видаляється об'єкт objA. Зрозуміло, що кожного разу викликається деструктор.

Перевантаження операторів

Наведений далі приклад є черговою варіацією на фінансову тему, доповнену цього разу *перевантаженням операторів*.

Теорія

Для класів користувача можна перевизначати (або, точніше, задавати) правила обчислення виразів із основними операторами. Це називається ***перевантаженням операторів***. Для перевантаження певного оператора описується спеціальна ***операторна функція*** або ***операторний метод***. Операторна функція або метод описуються, як і звичайна функція чи метод, проте з деякими особливостями. Так, назву операторної функції або методу отримують об'єднанням ключового слова operator і символу оператора. Тип результату операторної функції або методу - тип значення, яке повертається під час застосування оператора до об'єкта класу. Аргументи операторної функції визначаються операндами виразу, для якого задається дія оператора. Якщо йдеться про операторний метод, то об'єкт виклику методу ототожнюється з першим операндом виразу.

У лістингу 5.5 наведено програму, основу якої складає клас MyMoney. Для нього визначено декілька операторних методів і декілька операторних функцій. Зокрема, за допомогою операторних функцій визначаються такі дії з об'єктами класу MyMoney:

- Віднімання об'єктів: якщо від одного об'єкта класу MyMoney відняти інший об'єкт класу MyMoney, у результаті отримаємо різницю кінцевих сум за відповідними вкладами.
- Префіксна форма оператора декремента: під час застосування оператора декремента до об'єкта значення поля money об'єкта зменшується на величину 1000 (але залишається зі значенням не меншим за нуль).
- Постфіксна форма оператора декремента: під час застосування оператора декремента до об'єкта значення поля time об'єкта зменшується на 1 (але не може бути меншим за нуль).

За допомогою операторних методів задаємо такі дії з об'єктами класу MyMoney:

- Під час додавання двох об'єктів класу MyMoney отримуємо новий об'єкт цього ж класу, у якого значення полів обчислюються на основі значень полів вихідних об'єктів (тих, що додаються).
- Префіксна форма оператора інкремента: під час застосування оператора інкремента до об'єкта значення поля money об'єкта збільшується на величину 1000.
- Постфіксна форма оператора інкремента: під час застосування оператора інкремента до об'єкта значення поля time об'єкта збільшується на 1.

Подробиці



У операторів інкремента ++ і декремента -- є префіксна і постфіксна форми. У префіксній формі оператор вказують перед операндом. У постфіксній формі оператор вказують після операнда. Префіксну й постфіксну форми операторів інкремента й декремента можна перевантажувати по-різному. Діє таке правило: під час перевантаження

постфіксної форми оператора в операторному методі або функції оголошується формальний не використовуваний додатковий цілочисловий аргумент.

Для спрощення ситуації і зменшення обсягу програмного коду в програмний код класу MyMoney внесено деякі зміни. Зокрема, всі поля й методи в класі оголошено відкритими. Не використовується деструктор. Спрощено код конструкторів (тепер під час створення об'єктів повідомлення не відображаються). Отже, отримаємо такий програмний код:

Лістинг 5.5. Операторні методи й функції

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyMoney{
public:
   string name;
   double money;
   double rate;
   int time;
   // Конструктор без аргументів:
   MyMoney(){
      name="";
      money=0;
      rate=0;
      time=0;
   }
   // Конструктор із чотирма аргументами:
   MyMoney(string n,double m,double r,int t){
      setAll(n,m,r,t);
   }
```

```
// Методи класу:
double getMoney(){
   double s=money;
   for(int k=1;k<=time;k++){
      s*=(1+rate/100);
   }
   return s;
}
void showAll(){
   cout<<"Ім'я: "<<name<<endl;
   cout<<"Вклад: "<<money<<endl;
   cout<<"Ставка (%): "<<rate<<endl;
   cout<<"Період (років): "<<time<<endl;
   cout<<"Кінцева сума: "<<getMoney()<<endl;
   for(int k=1;k<=35;k++){
      cout<<"-";
   }
   cout<<endl;
}
void setAll(string n,double m,double r,int t){
   name=n;
   money=m;
   rate=r;
   time=t;
}
// Префіксна форма оператора інкремента:
MyMoney operator++(){
   money=money+1000;
   return *this;
}
// Постфіксна форма оператора інкремента:
MyMoney operator++(int){
```

```
        time++;
        return *this;
    }
    // Операторний метод для додавання двох об'єктів:
    MyMoney operator+(MyMoney obj){
        MyMoney tmp;
        tmp.name="Лис Микита";
        tmp.money=money+obj.money;
        tmp.rate=(rate>obj.rate)?rate:obj.rate;
        tmp.time=(time+obj.time)/2;
        return tmp;
    }
};
// Операторна функція для віднімання об'єктів:
double operator-(MyMoney objX,MyMoney objY){
    return objX.getMoney()-objY.getMoney();
}
// Префіксна форма оператора декремента:
MyMoney operator--(MyMoney &obj){
    if(obj.money>1000){
        obj.money-=1000;
    }
    else{
        obj.money=0;
    }
    return obj;
}
// Постфіксна форма оператора декремента:
MyMoney operator--(MyMoney &obj,int){
    if(obj.time>0){
        obj.time--;
    }
```

```
    else{
        obj.time=0;
    }
    return obj;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єкта:
    MyMoney objA("Пан Коцький",1200,7,1);
    objA.showAll();
    // Зменшення значення поля time:
    objA--;
    objA.showAll();
    // Зменшення значення поля time:
    objA--;
    objA.showAll();
    // Збільшення значення поля time:
    objA++;
    objA.showAll();
    // Зменшення значення поля money:
    --objA;
    objA.showAll();
    // Зменшення значення поля money:
    --objA;
    objA.showAll();
    // Збільшення значення поля money:
    ++objA;
    objA.showAll();
    // Створення об'єкта:
    MyMoney objB("Вінні Пух",1100,8,5);
```

```
    objB.showAll();
    // Створення об'єкта:
    MyMoney objC;
    // Обчислення суми об'єктів:
    objC=objA+objB;
    objC.showAll();
    // Обчислення різниці об'єктів:
    cout<<"Різниця в доходах: "<<objC-objB<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 5.5)

```
Ім'я: Пан Коцький
Вклад: 1200
Ставка (%): 7
Період (років): 1
Кінцева сума: 1284
-----------------------------------
Ім'я: Пан Коцький
Вклад: 1200
Ставка (%): 7
Період (років): 0
Кінцева сума: 1200
-----------------------------------
Ім'я: Пан Коцький
Вклад: 1200
Ставка (%): 7
Період (років): 0
Кінцева сума: 1200
```

```
----------------------------------
Ім'я: Пан Коцький
Вклад: 1200
Ставка (%): 7
Період (років): 1
Кінцева сума: 1284
----------------------------------
Ім'я: Пан Коцький
Вклад: 200
Ставка (%): 7
Період (років): 1
Кінцева сума: 214
----------------------------------
Ім'я: Пан Коцький
Вклад: 0
Ставка (%): 7
Період (років): 1
Кінцева сума: 0
----------------------------------
Ім'я: Пан Коцький
Вклад: 1000
Ставка (%): 7
Період (років): 1
Кінцева сума: 1070
----------------------------------
Ім'я: Вінні Пух
Вклад: 1100
Ставка (%): 8
Період (років): 5
Кінцева сума: 1616.26
----------------------------------
Ім'я: Лис Микита
```

```
Вклад: 2100
Ставка (%): 8
Період (років): 3
Кінцева сума: 2645.4
---------------------------------
Різниця в доходах: 1029.13
```

Проаналізуємо ту частину програми, яка пов'язана з описом і використанням операторних методів і операторних функцій. Почнемо з аналізу описання операторних функцій.

Операторна функція для віднімання об'єктів називається `operator-()`, повертає результат типу `double`, і в ній оголошено два аргументи `objX` і `objY` - обидва об'єкти класу `MyMoney`. Таке оголошення означає, що якщо ми від одного об'єкта класу `MyMoney` віднімемо інший об'єкт класу `MyMoney`, то результатом буде число типу `double`. Перший аргумент `objX` відповідає першому операнду (об'єкт, із якого віднімається значення), а інший аргумент `objY` відповідає другому операнду (об'єкт, який віднімається). У тілі функції всього одна команда `return objX.getMoney()-objY.getMoney()`, якою результат функції розраховується як різниця значень, що повертаються методом `getMoney()` для кожного з об'єктів. Очевидно, що тут ідеться про обчислення різниці значень кінцевих сум за депозитами.

Операторна функція для префіксної форми оператора декремента називається `operator--()`. У функції один аргумент (позначено як `obj`) типу `MyMoney`, причому цей аргумент передається за посиланням. Як результат функція повертає також об'єкт класу `MyMoney`. У тілі функції в умовному операторі перевіряється умова `obj.money>1000`. Умова полягає в тому, що значення поля `money` того об'єкта, до якого застосовують операцію інкремента, перевищує значення `1000`. Якщо це так, то командою `obj.money-=1000` значення цього поля зменшується на величину `1000`. У

протилежному випадку командою `obj.money=0` полю присвоюється нульове значення. Як результат функції повертається об'єкт `obj`.

До уваги

В операторній функції `operator--()` змінюється аргумент функції (об'єкт `obj`). Саме тому аргумент у функцію передається за посиланням, а не за значенням.

Функцію для постфіксної форми оператора декремента, порівняно з префіксною формою, оголошено з додатковим цілочисловим аргументом. Причому оскільки роль аргументу чисто формальна, то ім'я аргументу можна не вказувати, а тільки його тип. У тілі функції запускається умовний оператор із умовою `obj.time>0`. Якщо цілочислове поле `time` невід'ємне, то командою `obj.time--` його значення зменшується на одиницю. Якщо умову `obj.time>0` не виконано, то командою `obj.time=0` полю присвоюється нульове значення.

Як зазначалося раніше, під час перевантаження операторів зазвичай є альтернатива: описувати операторні функції або операторні методи. У цьому прикладі частина операцій реалізується через операторні методи. Так, операторний метод `operator++()` для оператора інкремента описано без аргументів і як такий, що повертає як значення об'єкт класу `MyMoney`. Відсутність аргументів у операторного методу пояснюється просто: єдиний операнд у цьому випадку ототожнюється з об'єктом, із якого викликається метод. Можна сказати й інакше: об'єкт, до якого застосовується операнд, є тим об'єктом, із якого викликають операторний метод для обробки відповідної команди.

У тілі методу командою `money=money+1000` змінюється значення поля `money`, після чого командою `return *this` об'єкт, із якого викликається метод, повертається як результат виразу. Тут ми скористалися стандартним ключовим словом `this`, яке є вказівником на об'єкт, із якого викликано метод. Відповідно, `*this` позначає сам об'єкт.

Подробиці



Незайвим буде уточнити спосіб, яким функції та методи (операторні й не тільки) повертають результат. Отже, ситуація така. Якщо метод або функція повертає результат, то під значення, що повертається, виділяється область пам'яті. У цю область пам'яті записується копія того значення, яке повертається як результат метода чи функції. Тому якщо метод або функція повертає як результат деякий об'єкт, то насправді повертається копія цього об'єкта.

Аналогічним чином описується операторний метод для постфіксної форми оператора інкремента. Тільки тепер у метода є формальний цілочисловий аргумент (точніше, указано лише ідентифікатор типу аргументу) - ознака того, що йдеться про постфіксну форму оператора. У тілі оператора командою `time++` значення поля `time` збільшується на одиницю, після чого командою `return *this` результатом повертається об'єкт, із якого викликається операторний метод (об'єкт, до якого застосовується операція інкремента).

Операторний метод `operator+()` для додавання двох об'єктів описано з аргументом `obj` - об'єктом класу `MyMoney`, і як результат метод повертає об'єкт того ж класу. Таким чином, перший операнд у відповідному виразі ототожнюється з об'єктом, із якого викликається метод, а другий операнд - із об'єктом `obj`, який передано як аргумент операторному методу. У тілі методу командою `MyMoney tmp` створюється локальний об'єкт `tmp` класу `MyMoney`. Після створення об'єкта його полям присвоюються значення. Так, командою `tmp.name="Лис Микита"` присвоюється значення полю `name` (очевидно, це значення не залежить від значень полів об'єктів, задіяних в операції додавання). Значення поля `money` для об'єкта `tmp` обчислюється за допомогою команди `tmp.money=money+obj.money`. Отже, поле `money` об'єкта `tmp` є сумою значень полів `money` об'єктів, що додаються. Значення поля `rate` об'єкта `tmp` обчислюється за допомогою команди `tmp.rate=(rate>obj.rate)?rate:obj.rate`. Тут ми скористалися тернарним оператором `?:`. Результат виразу

(rate>obj.rate)?rate:obj.rate обчислюється так: перевіряється умова rate>obj.rate, і якщо вона істинна, то як результат повертається значення rate (поле об'єкта для першого операнда). Якщо умова rate>obj.rate хибна, як результатом повертається obj.rate (поле rate об'єкта obj для другого операнда). Таким чином, як значення полю rate об'єкта tmp присвоюється найбільше зі значень полів rate об'єктів, що додаються.

Нарешті, командою tmp.time=(time+obj.time)/2 значення поля time обчислюється як середньоарифметичне для значень полів time об'єктів, що додаються. Слід урахувати, що оскільки поле time цілочислове, то операція ділення в наведеній вище команді виконується на множині цілих чисел (ділення націло - дробова частина відкидається).

Після присвоювання значень полям об'єкта tmp цей об'єкт повертається як результат операторного методу.

У функції main() перевіряються можливості описаних у програмі операторних функцій і методів. Для початку командою MyMoney objA("Пан Коцький",1200,7,1) створюється об'єкт objA класу MyMoney. На цьому об'єкті тестується виконання унарних операторів інкремента ++ і декремента --. Кожного разу після зміни параметрів об'єкта командою objA.showAll() здійснюється перевірка значень його полів.

Для тестування бінарних операцій (додавання та віднімання об'єктів) командою MyMoney objB("Вінні Пух",1100,8,5) створюється ще один об'єкт (другий операнд), а для запису результату обчислення суми створюємо об'єкт за допомогою команди MyMoney objC. Після цього виконується команда objC=objA+objB, а значення полів об'єкта-результату перевіряються за допомогою інструкції objC.showAll(). Також у програмі обчислюється різниця об'єктів objC-objB (результат - число типу double).

Знайомство з успадкуванням

У наступному прикладі на основі класу MyMoney шляхом *успадкування* створюється клас BigMoney.

Теорія

Успадкування - механізм, що дозволяє створювати нові класи на основі тих, які вже існують. Клас, який є основою для створення другого класу, називається ***базовим***. Клас, який створюється на основі базового класу, називається ***похідним*** класом.

Щоб створити похідний клас на основі базового класу в описі похідного класу після його імені вказують базовий клас. Між іменем похідного й базового класу ставлять двокрапку й ідентифікатор, що визначає характер (тип) успадкування. Як ідентифікатор можуть використовуватися ключові слова public (відкрите успадкування), private (закрите успадкування) і protected (захищене успадкування). Незалежно від типу успадкування, у похідному класі закриті члени базового класу не спадкуються.

Головна відмінність класу BigMoney від класу MyMoney полягає в тому, що в класі BigMoney з'являється цілочислове поле periods, що визначає кількість річних нарахувань відсотків за депозитом. Відповідно, корективи вносяться до способу обчислення кінцевої суми на депозиті і до способу відображення інформації про об'єкт.

Далі наведено програму, у якій використано успадкування класів:

Лістинг 5.6. Успадкування класів

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис базового класу:
class MyMoney{
public:
   // Поля базового класу:
```

```
    string name;
    double money;
    double rate;
    int time;
    // Методи базового класу:
    double getMoney(){
        double s=money;
        for(int k=1;k<=time;k++){
            s*=(1+rate/100);
        }
        return s;
    }
    void showAll(){
        cout<<"Ім'я: "<<name<<endl;
        cout<<"Вклад: "<<money<<endl;
        cout<<"Ставка (%): "<<rate<<endl;
        cout<<"Період (років): "<<time<<endl;
        cout<<"Кінцева сума: "<<getMoney()<<endl;
    }
    void setAll(string n,double m,double r,int t){
        name=n;
        money=m;
        rate=r;
        time=t;
    }
    // Конструктор базового класу (чотири аргументи):
    MyMoney(string n,double m,double r,int t){
        setAll(n,m,r,t);
    }
    // Конструктор базового класу (без аргументів):
    MyMoney(){
        setAll("",0,0,0);
```

```
    }
};
// Похідний клас:
class BigMoney: public MyMoney{
public:
    // Поля похідного класу:
    int periods;
    // Перевизначення методів:
    double getMoney(){
        double s=money;
        for(int k=1;k<=time*periods;k++){
            s*=(1+rate/100/periods);
        }
        return s;
    }
    void showAll(){
        cout<<"Ім'я: "<<name<<endl;
        cout<<"Вклад: "<<money<<endl;
        cout<<"Ставка (%): "<<rate<<endl;
        cout<<"Період (років): "<<time<<endl;
        cout<<"Нарахувань на рік: "<<periods<<endl;
        cout<<"Кінцева сума: "<<getMoney()<<endl;
    }
    void setAll(string n,double m,double r,int t,int p){
        MyMoney::setAll(n,m,r,t);
        periods=p;
    }
    // Конструктор похідного класу (п'ять аргументів):
    BigMoney(string n,double m,double r,int t,int p=1):
MyMoney(n,m,r,t){
        periods=p;
    }
```

```
    // Конструктор похідного класу (без аргументів):
    BigMoney(): MyMoney(){
       periods=1;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єкта класу MyMoney:
    MyMoney objA("Пан Коцький",1200,8,5);
    // Створення об'єктів класу BigMoney:
    BigMoney objB("Дональд Дак",1000,7,6,2);
    BigMoney objC("Вінні Пух",1500,6,8);
    BigMoney objD;
    objD.setAll("Лис Микита",800,10,3,4);
    // Перевірка характеристик об'єктів:
    objA.showAll();
    cout<<endl;
    objB.showAll();
    cout<<endl;
    objC.showAll();
    cout<<endl;
    objD.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми буде такий:

Результат виконання програми (із лістингу 5.6)

```
Ім'я: Пан Коцький
Вклад: 1200
```

```
Ставка (%): 8
Період (років): 5
Кінцева сума: 1763.19

Ім'я: Дональд Дак
Вклад: 1000
Ставка (%): 7
Період (років): 6
Нарахувань на рік: 2
Кінцева сума: 1511.07

Ім'я: Вінні Пух
Вклад: 1500
Ставка (%): 6
Період (років): 8
Нарахувань на рік: 1
Кінцева сума: 2390.77

Ім'я: Лис Микита
Вклад: 800
Ставка (%): 10
Період (років): 3
Нарахувань на рік: 4
Кінцева сума: 1075.91
```

Базовий клас `MyMoney` описано досить просто: у нього є чотири поля (`name`, `money`, `rate` і `time`), три методи (`getMoney()`, `setAll()` і `showAll()`), а також дві версії конструктора - без аргументів і з чотирма аргументами. Нас він цікавить не сам по собі, а як основа для створення, шляхом успадкування, класу `BigMoney`. В описі класу `BigMoney` після імені класу (через двокрапку) вказано ключове слово `public` і ім'я класу `MyMoney`. Це означає, що клас `BigMoney`

створюється шляхом відкритого успадкування на основі класу `MyMoney`, і тому всі відкриті члени класу `MyMoney` автоматично стають членами класу `BigMoney`. Отже, безпосередньо в класі `BigMoney` є сенс описувати тільки ті члени, які "додаються" до успадкованих із базового класу. Щодо полів, у класі `BigMoney` додається тільки цілочислове поле `periods`. Також ми використовуємо *перевизначення методів*, успадкованих із класу `MyMoney`.

Теорія

Під час успадкування в похідному класі можна змінити, або ***перевизначити***, код методу, який успадковується з базового класу. Для цього в похідному класі метод (не дивлячись на те, що він успадковується), описують заново з потрібним кодом.

Не варто плутати ***перевантаження*** й ***перевизначення*** методу. Під час перевантаження створюється додаткова версія методу з такою ж назвою, але зміненим прототипом. Перевизначення методу може мати місце тільки під час успадкування і полягає в тому, що створюється нова версія методу замість тієї, що успадковується.

Незважаючи на те, що в класі `BigMoney` успадковуються методи `getMoney()`, `showAll()` і `setAll()`, нас вони не дуже влаштовують, оскільки у класу тепер є додаткове поле, що, крім іншого, призводить до зміни способу обчислення кінцевої суми за депозитом. Іншими словами, потрібно, щоб такі методи у класу `BigMoney` були, але необхідно внести зміни до їхнього програмного коду. Ми чинимо просто і невигадливо - у тілі класу `BigMoney` описуємо ці методи заново.

Програмний код методу `getMoney()` змінюється так, щоб враховувалася наявність декількох виплат на рік. Основних змін (порівняно з кодом методу з класу `MyMoney`) дві: кількість періодів нарахування відсотків множиться на значення поля `periods`, а відсоткова ставка ділиться на значення поля `periods`.

До уваги

Фактично, якщо раніше обчислення проводилися відповідно до формули $m\left(1+\frac{r}{100}\right)^t$, то тепер робоча формула має вигляд $m\left(1+\frac{r}{100p}\right)^{pt}$, де параметри m, r, tip

позначають відповідно початкову суму вкладу, річну відсоткову ставку, час, на який розміщується вклад, і кількість періодів у році для нарахування відсотків.

Програмний код методу `showAll()` у похідному класі відрізняється від коду аналогічного методу в базовому класі тільки командою `cout<<"Нарахувань у рік: "<<periods<<endl`, якою в консольне вікно виводиться значення поля `periods`.

Подробиці



У тілі методу `showAll()` у похідному класі є команда виклику методу `getMoney()`. Оскільки метод `getMoney()` перевизначено в похідному класі, то буде викликатися саме ця перевизначена версія методу.

Метод `setAll()` описано в похідному класі з п'ятьма, а не з чотирма аргументами, як було в базовому класі. Додатковий п'ятий аргумент потрібен для передачі значення полю `periods`. У тілі методу `setAll()` всього дві команди. Командою `MyMoney::setAll(n,m,r,t)` викликається стара, успадкована з базового класу `MyMoney` версія методу `setAll()` із чотирма аргументами. У результаті отримують значення поля `name`, `money`, `rate` і `time`. А значення полю `periods` присвоюється в явному вигляді командою `periods=p`.

Подробиці



Під час перевизначення методів у похідному класі "старі" версії, які успадковуються з базового класу, насправді нікуди не діваються. Технічно вони існують. Просто "перекриваються" новими версіями методів. За необхідності ці "сховані" версії методів із базових класів можна "дістати" і використати за призначенням. Однак для цього необхідно явно вказати, що мається на увазі та версія методу, яка успадковується з базового класу. Для цього перед іменем методу вказується ім'я базового класу, з якого метод успадковується. Між іменем класу та іменем методу ставиться оператор розширення області видимості : : (дві двокрапки).

Варта уваги і та обставина, що в класі `BigMoney` метод `setAll()` описується з п'ятьма аргументами, а в класі `MyMoney` у методу `setAll()`

чотири аргументи. Може скластися враження, що в похідному класі виконується перевантаження методу `setAll()`, але це не так. У похідному класі версія методу `setAll()` з п'ятьма аргументами "перекриває" успадковану з базового класу версію методу `setAll()` з чотирма аргументами. Так що прямий (без зазначення базового класу) доступ у похідному класі є лише до тієї версії методу, що описана в похідному класі.

У похідному класі описано дві версії конструктора: без аргументів і з п'ятьма аргументами (один із аргументів має значення за замовчуванням). Специфіка опису конструктора в похідному класі така, що необхідно явно вказувати виклик конструктора базового класу. Наприклад, під час опису конструктора класу `BigMoney` із п'ятьма аргументами після назви конструктора і списку аргументів через двокрапку вказано інструкцію `MyMoney(n,m,r,t)`, яка означає виклик конструктора базового класу з відповідними аргументами. Після того, як виконання конструктора базового класу закінчується, командою `periods=p` присвоюється значення полю `periods`. Простіше кажучи, в тілі конструктора похідного класу програмуються ті додаткові дії, які слід виконати після виклику конструктора базового класу.

До уваги

У тілі конструктора базового класу `MyMoney` викликається метод `setAll()` із чотирма аргументами. Оскільки виклик відбувається в конструкторі саме базового класу, то і викликається версія методу з базового класу.

Аналогічним чином описується версія конструктора похідного класу без аргументів. Тільки тепер конструктор базового класу також викликається без аргументів, а полю `periods` присвоюється одиничне значення.

У функції `main()` створюються декілька об'єктів і потім перевіряється їхні характеристики. Так, командою `MyMoney objA("Пан Коцький",1200,8,5)` створюється об'єкт базового класу `MyMoney`. Командами `BigMoney objB("Дональд Дак",1000,7,6,2)`,

`BigMoney objC("Вінні Пух",1500,6,8)` і `BigMoney objD` створюються об'єкти похідного класу `BigMoney`. Оскільки в похідному класі описано конструктор без аргументів і з п'ятьма аргументами (при тому, що один із аргументів має значення за замовчуванням), то під час створення об'єкта в конструктор можна передавати чотири чи п'ять аргументів, або не передавати їх узагалі. Командою `objD.setAll("Лис Микита",800,10,3,4)` задаються значення полів об'єкта похідного класу. Перевірка значень полів об'єктів виконується за допомогою методу `showAll()`.

Задачі для самостійного розв'язання

1. Напишіть програму, у якій створюється клас для опису такого фізичного об'єкта, як "паралелепіпед". У класу три поля: ширина, глибина і висота, а також метод для обчислення об'єму (добуток значень полів об'єкта). Передбачити наявність конструктора і методу для відображення характеристик об'єкта. Розгляньте механізм успадкування, створивши на основі класу похідний із додатковим полем, що визначає масу "паралелепіпеда". Додати в похідному класі метод для обчислення щільності матеріалу, з якого зроблено "паралелепіпед" (масу необхідно поділити на об'єм). Подумайте над тим, як додати в клас операторні методи чи описати операторні функції для виконання деяких простих операцій із об'єктами. Наприклад, ділення об'єкта на число може призводити до того, що маса "паралелепіпеда" зменшується в певну кількість разів. Або: в результаті додавання двох об'єктів отримуємо новий об'єкт. Його об'єм дорівнює сумі об'ємів вихідних об'єктів, його маса дорівнює сумі мас вихідних об'єктів, а всі три лінійних розміри однакові.

2. Напишіть програму з класом для реалізації комплексних чисел. У класу два поля (дійсна та уявна частини комплексного числа). Описати для класу операторні функції та методи, що дозволяють виконувати базові математичні

операції з об'єктами класу. Створити на основі базового класу похідний клас, у якому реалізувати представлення комплексного числа в тригонометричній формі. Нагадаємо, що комплексне число $z = x + iy$ має дійсну x і уявну y частини, а уявна одиниця i така, що $i^2 = -1$. Основні математичні операції з комплексними числами виконуються так само, як і зі звичайними числами, з урахуванням лише того, що $i^2 = -1$. У тригонометричній формі комплексне число може бути представлене у вигляді $z = |z| \exp(i\varphi)$, де модуль числа $|z| = \sqrt{x^2 + y^2}$, а аргумент φ такий, що $\sin(\varphi) = \frac{y}{\sqrt{x^2+y^2}}$ і $\cos(\varphi) = \frac{x}{\sqrt{x^2+y^2}}$. Урахувати, що має місце тотожність $\exp(i\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$.

3. Напишіть програму з класом для реалізації векторів у двовимірному просторі. У класу два поля (координати вектора). Передбачити виконання базових операцій з векторами (таких, наприклад, як додавання і віднімання векторів, множення і ділення векторів на число, обчислення скалярного добутку векторів). Шляхом успадкування створити клас для реалізації векторів у тривимірному просторі (у такого вектора три координати). Нагадаємо, що вектор у тривимірному просторі однозначно описується трьома координатами. Операції з векторами виконуються так: якщо $\vec{a} = (a_1, a_2, a_3)$ і $\vec{b} = (b_1, b_2, b_3)$ - вектори, а λ і ξ - числа, то $\lambda\vec{a} + \xi\vec{b} = (\lambda a_1 + \xi b_1, \lambda a_2 + \xi b_2, \lambda a_3 + \xi b_3)$, а скалярний добуток $\vec{a} \cdot \vec{b} = a_1 b_1 + a_2 b_2 + a_3 b_3$.

4. Напишіть програму з класом для опису "легкового автомобіля". Через поля класу реалізуються такі характеристики, як марка, колір, об'єм паливного бака (у літрах), норма споживання бензину (літрів на 100 кілометрів). Передбачити, крім іншого, наявність методу, що дозволяє обчислити відстань, яку проїде автомобіль, витративши повний бак бензину (обчислюється як помножене на 100 відношення об'єму паливного бака до норми споживання бензину). На основі класу шляхом успадкування створити похідний клас для опису "вантажного автомобіля". Додати поле, що описує вантажопідйомність

автомобіля (максимальна маса вантажу в тоннах, яку може перевозити автомобіль) і метод, який дозволяє обчислити собівартість транспортування однієї тонни вантажу на один кілометр. Результат методу розраховується так: норма споживання бензину ділиться на 100, множиться на ціну літра бензину і ділиться на вантажопідйомність. Ціна літра бензину передається як аргумент методу. Розглянути можливість визначення для класу операторних методів і операторних функцій.

5. Напишіть програму з класом для опису ситуації, коли грошовий вклад, що розміщується на депозиті, складається з двох частин. Час розміщення депозиту однаковий для обох частин вкладу, а відсоткова ставка для кожної частини вкладу своя. Фактично йдеться про розширений варіант задачі, розглянутої вище в цій главі. Пропонується реалізувати ті ж операції, що описувалися для об'єктів класу `MyMoney`, але тільки з урахуванням нової постановки задачі.

Рекомендації для самостійної роботи

- Запропонуйте об'єкти і явища з реального життя, які можна було б описати на програмному рівні в рамках концепції класів і об'єктів. Складіть відповідні програми.
- Які математичні об'єкти й операції можуть бути запрограмовані із застосуванням об'єктно-орієнтованого підходу? Запропонуйте свої програми.
- Запропонуйте варіанти розв'язання тих задач, які розглядалися в попередніх главах (чи пропонувалися там для самостійного розв'язання), із використанням класів і об'єктів.

Глава 6

Використання класів і об'єктів

Тобі горілка в голові, а мені, бач, - грамота.

з х/ф "Пропала грамота"

У цій главі ми розглянемо ряд специфічних питань, пов'язаних із використанням класів і об'єктів. Крім іншого, ми зупинимося на таких темах як:

- створення масиву об'єктів;
- створення об'єкта з полем-масивом;
- робота з вказівниками на об'єкт;
- створення функторів;
- індексування об'єктів;
- реалізація конструктора створення копії об'єкта;
- використання віртуальних методів;
- застосування множинного успадкування;
- особливості роботи із закритими членами базового класу під час успадкування;
- доступ до об'єктів похідного класу через об'єктні змінні базового класу.

Приклади, що розглядаються далі, і задачі покликані проілюструвати найважливіші механізми ООП, які реалізуються в C++.

Вказівник на об'єкт

Для об'єкта, як і для значення простого (базового типу) можна створити вказівник. Вказівник на об'єкт - змінна, значенням якої є адреса об'єкта.

Теорія

Отримати адресу об'єкта можна за допомогою інструкції `&`. Якщо перед вказівником поставити `*`, отримаємо значення - тобто сам об'єкт. Якщо є вказівник на об'єкт, то

для отримання доступу до полів і методів об'єкта через вказівник використовують оператор ->.

Невеликий приклад використання вказівників на об'єкти наведено в лістингу 6.1.

Лістинг 6.1. Доступ до об'єктів через вказівник

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyClass{
public:
    // Текстове поле:
    string name;
    // Цілочислове поле:
    int number;
    // Метод для відображення значення полів:
    void show(){
        cout<<"Поле name: "<<name<<endl;
        cout<<"Поле number: "<<number<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єктів:
    MyClass objA,objB;
    // Оголошення вказівника на об'єкт:
    MyClass* p;
    // Адреса об'єкта записується у вказівник:
```

```
    p=&objA;
    // Присвоювання значень полям об'єкта через вказівник:
    p->name="Об'єкт objA";
    p->number=111;
    // Виклик методу через вказівник:
    p->show();
    // Нове значення вказівника:
    p=&objB;
    // Присвоювання значень полям об'єкта через вказівник:
    p->name="Об'єкт objB";
    p->number=222;
    // Виклик методу через вказівник:
    p->show();
    cout<<"Перевіряємо об'єкти\n";
    // Перевірка значень полів об'єктів:
    objA.show();
    objB.show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 6.1)

```
Поле name: Об'єкт objA
Поле number: 111
Поле name: Об'єкт objB
Поле number: 222
Перевіряємо об'єкти
Поле name: Об'єкт objA
Поле number: 111
Поле name: Об'єкт objB
Поле number: 222
```

У програмі описується клас MyClass, у якому є текстове (тип string) поле name і цілочислове (тип int) поле number. Також у класі описано метод show(), призначений для відображення значення полів об'єкта, із якого викликається метод.

У функції main() командою MyClass objA,objB створюються два об'єкти класу MyClass. Потім командою MyClass* p оголошується вказівник на об'єкт класу MyClass. Командою p=&objA як значення вказівнику присвоюється адреса об'єкта objA. Тому командами p->name="Об'єкт objA" і p->number=111 присвоюються значення полям об'єкта objA, а командою p->show() із об'єкта objA викликається метод show().

Після присвоювання нового значення вказівникові p за допомогою команди p=&objB, інструкції p->name="Об'єкт objB", p->number=222 і p->show() означають звернення до полів і методу об'єкта objB. Перевірку значень полів об'єктів objA і objB виконуємо, викликавши з цих об'єктів метод show().

Ще один приклад використання вказівників на об'єкт наведено у наступній програмі. Там створюється *ланцюжок об'єктів*. Усі об'єкти динамічні - тобто пам'ять під них виділяється в динамічному режимі. У кожного об'єкта є поле-вказівник, що містить адресу наступного об'єкта в ланцюжку. Таким чином, маючи доступ до першого об'єкта в ланцюжку, можемо отримати доступ до будь-якого об'єкта. Програма, у якій реалізовано цю ідею, наведено в лістингу 6.2.

Лістинг 6.2. Використання вказівників на об'єкт

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Опис класу:
```

```
class MyClass{
public:
    // Символьне поле:
    char code;
    // Поле - вказівник на об'єкт:
    MyClass* next;
    // Деструктор:
    ~MyClass(){
        cout<<"Об'єкт із полем "<<code<<" видалено\n";
    }
    // Метод для відображення значення символьного поля:
    void show(){
        // Відображення значення поля:
        cout<<code<<" ";
        if(next){
            // Виклик методу для наступного об'єкта:
            next->show();
        }
    }
};
// Функція для видалення ланцюжка об'єктів:
void deleteAll(MyClass* q){
    // Перевірка адреси в полі next:
    if(q->next){
        // Виклик функції для наступного об'єкта:
        deleteAll(q->next);
    }
    // Видалення поточного об'єкта:
    delete q;
}
// Головна функція програми:
int main(){
```

```
// Зміна кодування консолі:
system("chcp 1251>nul");
// Цілочислова змінна:
int n=10;
// Динамічне створення об'єкта:
MyClass* pnt=new MyClass;
// Доступ до поля об'єкта через вказівник на нього:
pnt->code='A';
// Вказівник на об'єкт:
MyClass *p;
// Початкове значення вказівника:
p=pnt;
// Створення ланцюжка об'єктів:
for(int k=1;k<=n;k++){
   // Створення чергового об'єкта:
   p->next=new MyClass;
   // Значення поля code для нового об'єкта:
   p->next->code=p->code+1;
   // Нове значення вказівника - адреса
   // створеного об'єкта:
   p=p->next;
}
// Нульове посилання в полі next для останнього
// об'єкта в ланцюжку:
p->next=0;
// Виклик методу для першого об'єкта в ланцюжку:
pnt->show();
cout<<endl;
// Видалення ланцюжка об'єктів:
deleteAll(pnt);
// Затримка консольного вікна:
system("pause>nul");
```

```
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 6.2)

```
A B C D E F G H I J K
Об'єкт із полем K видалено
Об'єкт із полем J видалено
Об'єкт із полем I видалено
Об'єкт із полем H видалено
Об'єкт із полем G видалено
Об'єкт із полем F видалено
Об'єкт із полем E видалено
Об'єкт із полем D видалено
Об'єкт із полем C видалено
Об'єкт із полем B видалено
Об'єкт із полем A видалено
```

Проаналізуємо програмний код прикладу. У класі `MyClass` описано два поля: символьне поле `code` і поле `next`, яке є вказівником на об'єкт класу `MyClass` (поле оголошено за допомогою інструкції `MyClass* next`). Також у класі описано деструктор. У тілі деструктора командою `cout<<"Об'єкт із полем "<<code<<" видалено\n"` відображається повідомлення про видалення об'єкта і вказується значення поля `code` об'єкта, що видаляється. Деструктор нам потрібен виключно для того, щоб зробити наочним процес видалення об'єкта. Простіше кажучи, коли буде видалятися той чи інший об'єкт, ми це побачимо (практично в буквальному сенсі).

Ще в класі `MyClass` описано метод `show()`, код якого вартий окремої уваги. Метод не повертає результат, і в нього немає аргументів. Під час виклику методу спочатку командою `cout<<code<<" "` відображається значення поля `code` об'єкта, з якого викликається метод. Але це ще не все.

Після відображення значення поля `code` в умовному операторі перевіряється значення поля `next`. Якщо значення поля відмінне від нуля, то командою `next->show()` викликається метод `show()` для наступного об'єкта в ланцюжку.

Подробиці



У полі `next` об'єкта записано адресу наступного елемента. Для останнього об'єкта в ланцюжку наступного об'єкта немає, тому останній об'єкт не посилається на інший об'єкт. Передбачається, що поле `next` останнього в ланцюжку об'єкта містить нульове значення. Таким чином, рівність нулю поля `next` інтерпретується як ознака останнього об'єкта в ланцюжку.

Тут також доречно нагадати, що відмінні від нуля значення інтерпретуються як логічне значення `true`, а нульове значення інтерпретується як логічне значення `false`. Тому відмінна від нуля адреса в полі `next` (передана умовою в умовний оператор) інтерпретується як логічне значення `true`, а поле `next` із нульовим значенням в останньому об'єкті інтерпретується як значення `false`.

Отже, під час виклику методу `show()` із об'єкта відображається значення `code` об'єкта, після чого автоматично викликається метод `show()` для наступного об'єкта, і так далі. У результаті під час виклику методу `show()` послідовно відображаються значення полів `code` всіх об'єктів у ланцюжку, починаючи з того, з якого викликається метод.

Оскільки об'єкти в програмі створюються динамічні, то їх варто видаляти. Проблема пов'язана з тим, що йдеться про ланцюжок об'єктів, і прямий доступ є лише до першого об'єкта. Тому під час видалення об'єктів треба "пройти" по ланцюжку і видалити кожний об'єкт. Для розв'язання задачі описується функція `deleteAll()`. Функція не повертає результат, і у неї один аргумент (позначений як `q`), який є вказівником на об'єкт класу `MyClass`. Код тіла функції починається з умовного оператора (у спрощеній формі), у якому перевіряється умова `q->next`. Умова `q->next` полягає в тому, що адреса в

полі next об'єкта, переданого через вказівник q аргументом функції, не дорівнює нулю. Простіше кажучи, ця умова істинна, якщо об'єкт, адресу якого записано у вказівник q, не є останнім об'єктом у ланцюжку. Якщо так, то командою deleteAll(q->next) функція deleteAll() викликається для наступного об'єкта в ланцюжку. Після умовного оператора командою delete q видаляється об'єкт, адреса якого міститься у вказівнику q.

До уваги

Тобто загальна схема така: функція deleteAll() викликається із зазначенням адреси деякого об'єкта, який повинен бути видалений. Але перед його видаленням автоматично викликається функція deleteAll() для видалення наступного об'єкта. Перед видаленням наступного об'єкта знову викликається функція deleteAll() для видалення ще одного об'єкта в ланцюжку, і так далі - поки черга не дійде до останнього об'єкта. Він видаляється, після чого видаляється попередній об'єкт, потім об'єкт перед ним, і так далі. Таким чином, об'єкти будуть видалятися з кінця ланцюжка в його початок. Під час виклику функції deleteAll() як аргумент їй передається перший об'єкт у ланцюжку.

У головній функції програми оголошують цілочислову змінну n. Кількість об'єктів у ланцюжку на одиницю більша за значення цієї змінної. Командою MyClass* pnt=new MyClass створюється динамічний об'єкт, і його адреса записується у вказівник pnt. Командою pnt->code='A' полю code цього об'єкта присвоюється значення 'A'. Командою MyClass *p створюється вказівник p на об'єкт класу MyClass, і значення вказівнику присвоюється командою p=pnt. Тому в початковий момент вказівник p містить адресу того ж об'єкта, що і вказівник pnt. Далі запускається оператор циклу, в якому індексна змінна k приймає значення від 1 до n включно. За кожний цикл послідовно виконуються три команди. Спочатку командою p->next=new MyClass створюється черговий динамічний об'єкт, і його адреса записується в поле next об'єкта, що визначається вказівником p

(наступний об'єкт у ланцюжку). Командою `p->next->code=p->code+1` полю `code` створеного об'єкта присвоюється збільшене на одиницю значення поля `code` поточного (адресу якого записано у вказівнику `p`) об'єкта.

Подробиці



Звертання до поля `code` об'єкта, адресу якого записано в полі `next` (а поле `next`, у свою чергу, належить до об'єкта, що визначається вказівником `p`), виглядає як `p->next->code`. Така інструкція міститься в лівій частині команди присвоювання значення `p->next->code=p->code+1`. З іншого боку, в об'єкта, що визначається вказівником `p`, окрім поля `next` є ще й поле `code`. Звертання до цього поля здійснюється за допомогою інструкції `p->code`. Поле символьне, тому якщо до нього додати одиницю (вираз `p->code+1`), отримаємо наступну букву відносно тієї, що записана в полі `code` об'єкта, який визначається вказівником `p`. Саме таке значення (мається на увазі вираз `p->code+1`) присвоюється полю `p->next->code`.

Нарешті, командою `p=p->next` у вказівник `p` записується значення поля `next` об'єкта, адреса якого раніше була записана в цьому вказівникові. Таким чином вказівник "перекидається" на наступний об'єкт у ланцюжку.

Після виконання оператора циклу ланцюжок об'єктів створено. Залишилося тільки за допомогою команди `p->next=0` присвоїти нульове значення полю `next` останнього об'єкта в ланцюжку. Тут ми скористалися з того, що після закінчення оператора циклу, вказівник `p` міститиме адресу останнього об'єкта в ланцюжку об'єктів.

Командою `pnt->show()` із першого об'єкта в ланцюжку викликається метод `show()`. Завдяки автоматичному виклику цього методу для всіх інших об'єктів, у консольному вікні в рядок через пробіл відображуються значення символьних полів об'єктів для всього ланцюжка. Після цього командою `deleteAll(pnt)` видаляється ланцюжок об'єктів із пам'яті.

Створення масиву об'єктів

Нерідко доводиться зустрічатися із ситуацією, коли треба створити масив, елементами якого є об'єкти. Саме такий приклад розглянемо далі. У певному сенсі він нагадує попередній.

Теорія

Масив об'єктів створюється так само, як і масив елементів інших типів, з тією лише різницею, що як тип елемента вказують назву класу.

Важлива умова полягає в тому, що клас, на основі якого створюються елементи масиву, повинен мати конструктор без аргументів, оскільки саме такий конструктор викликається під час створення елементів масиву.

Основу наведеного в лістингу 6.3 програмного коду складає опис класу `MyWords`. У класу є текстове (тип `string`) поле `word`, логічне (тип `bool`) поле `state`, описано конструктор без аргументів і метод `read()`. У конструкторі командами `word=""` і `state=true` присвоюються значення полям. Метод `read()` (метод не повертає результат, і у нього немає аргументів) складається з команди `cout<<word<<" "` (командою відображається значення поля `word` об'єкта, з якого викликається метод) і умовного оператора у спрощеній формі. В умовному операторі умовою, що перевіряється, виступає значення поля `state`. Якщо значення поля дорівнює `true`, то командою `(this+1)->read()` викликається метод `read()` для наступного об'єкта в масиві. Тут ми скористалися двома обставинами:

- елементи масиву в пам'яті розміщені підряд;
- під час додавання до вказівника цілого числа отримуємо вказівник на область пам'яті, яка стоїть від вихідної на кількість комірок, що визначається числом, яке додають.

Ключове слово `this` у тілі методу є вказівником на об'єкт, із якого викликається метод. Значенням виразу `this+1` є вказівник на об'єкт, що знаходиться в пам'яті за цим об'єктом. Оскільки тут йдеться про об'єкти, які є елементами масиву, то значення виразу `this+1` - вказівник на наступний

елемент (він же об'єкт) масиву. Тому команда `(this+1)->read()` означає виклик методу `read()` для наступного (порівняно з поточним) об'єкта в масиві.



До уваги

Виходить так, що під час виклику методу `read()` із об'єкта в масиві цей метод почне викликатися з кожного наступного елемента в масиві. Так буде тривати до тих пір, поки в масиві не зустрінеться об'єкт зі значенням поля `state`, що дорівнює `false`. Як ми побачимо далі, поле з таким значенням буде в останнього об'єкта в масиві.

Увесь програмний код приклада наведено нижче:



Лістинг 6.3. Масив об'єктів

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис класу:
class MyWords{
public:
   // Текстове поле:
   string word;
   // Поле логічного типу:
   bool state;
   // Конструктор без аргументів:
   MyWords(){
      word="";
      state=true;
   }
   // Метод для зчитування значення текстового поля:
   void read(){
      // Відображення значення поля об'єкта:
```

```
        cout<<word<<" ";
        // Перевірка значення поля state:
        if(state){
            // Виклик методу для сусіднього об'єкта:
            (this+1)->read();
        }
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Розмір масиву:
    const int n=5;
    // Текстовий масив:
    string numbers[n]={"один","два","три","чотири","п'ять"};
    // Масив об'єктів:
    MyWords words[n];
    // Значення для поля state останнього об'єкта в масиві:
    words[n-1].state=false;
    // Присвоювання значення полю word для
    // об'єктів у масиві:
    for(int k=0;k<n;k++){
        words[k].word=numbers[k];
    }
    // Виклик методу:
    words[0].read();
    cout<<endl;
    words[2].read();
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
```

```
    return 0;
}
```

У головній функції програми оголошується цілочислова константа `n` зі значенням `5` (розмір масиву). Допоміжний масив із текстовими значеннями створюємо й ініціалізуємо за допомогою команди `string numbers[n]={"один","два","три","чотири","п'ять"}`. Цей масив містить значення, які будуть присвоюватися текстовим полям об'єктів із масиву, оголошуваного інструкцією `MyWords words[n]`. Конструктор без аргументів, який автоматично викликається для створення кожного з об'єктів у масиві, присвоює полю `state` кожного з об'єктів значення `true`. Командою `words[n-1].state=false` задається значення `false` для поля `state` останнього об'єкту в масиві `words`. Після цього запускається оператор циклу, в якому перебираються елементи масиву `words`, і за кожний цикл командою `words[k].word=numbers[k]` полю `word` чергового об'єкта в масиві `words` присвоюється значення відповідного елемента з текстового масиву `numbers`.

Після того, як поля об'єктів у масиві `words` заповнено, з першого об'єкта в масиві викликається метод `read()` (команда `words[0].read()`). У результаті для кожного з об'єктів у масиві `words` відображається значення поля `word`. Якщо ж скористатися командою `words[2].read()`, то значення поля `word` буде відображатися для всіх об'єктів у масиві, починаючи з третього (індекс 2) об'єкта, і до кінця масиву. Результат виконання програми такий:

Результат виконання програми (із лістингу 6.3)

```
один два три чотири п'ять
три чотири п'ять
```

Перший рядок у консольному вікні з'являється внаслідок виконання команди `words[0].read()`, другий - під час виконання команди `words[2].read()`.

Масив як поле класу

У деяких випадках є сенс чи просто необхідно, щоб у класі полем був масив. Розглянемо такий випадок. У поданому далі прикладі описано клас `Taylor`, призначений для реалізації ряду Тейлора для різних функцій.

Подробиці



Для функції $f(x)$ рядом Тейлора в околі точки x_0 називається представлення функції у вигляді $f(x) = \sum_{k=0}^{\infty} c_k (x - x_0)^k$, де коефіцієнти $c_k = \frac{f^{(k)}(x_0)}{k!}$. Ряд Тейлора можна використовувати для апроксимування функцій. У цьому випадку ряд обривається, і має місце оцінка $f(x) \approx \sum_{k=0}^{n} c_k (x - x_0)^k$. Якщо йдеться про розкладання в околі нуля (тобто при $x_0 = 0$), то для функції $f(x)$ отримуємо оцінку у вигляді полінома $f(x) \approx \sum_{k=0}^{n} c_k x^k$, яка дає наближене значення функції в точці x. У такому випадку представлення функції $f(x)$ у вигляді ряду зводиться до визначення коефіцієнтів c_k (індекс $k = 0,1, \ldots, n$).

У програмі, що розглядається далі, описується клас з полем-масивом, і в цей масив заносяться коефіцієнти, які визначають ряд (обмежений) Тейлора для деякої функції. Кожний об'єкт такого класу відповідає певній функціональній залежності.

У лістингу 6.4 наведено програму, в якій описується клас `Taylor`. У класу є поле-масив. Масив призначено для запам'ятовування коефіцієнтів, які ми ототожнюємо з коефіцієнтами розкладання функції в ряд Тейлора (в околі нуля). У класу є дві версії конструктора і метод, що дозволяє обчислювати значення ряду Тейлора в зазначеній точці. Розглянемо код програми:

Лістинг 6.4. Ряд Тейлора для функції

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Розмір масиву:
const int n=10;
// Клас для реалізації ряду Тейлора:
```

```
class Taylor{
public:
    // Поле - масив:
    double a[n];
    // Конструктор із одним числовим аргументом:
    Taylor(double p=0){
        for(int k=0;k<n;k++){
            a[k]=p;
        }
    }
    // Конструктор із аргументом - вказівником:
    Taylor(double* b){
        for(int k=0;k<n;k++){
            a[k]=b[k];
        }
    }
    // Метод для обчислення значення ряду:
    double value(double x){
        double s=0,q=1;
        for(int k=0;k<n;k++){
            s+=a[k]*q;
            q*=x;
        }
        return s;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Числовий масив:
    double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
```

```
    // Створення об'єктів:
    Taylor myexp,f(1),mytan(b);
    // Звертання до елемента масиву - поля об'єкта:
    myexp.a[0]=1;
    // Заповнення масиву - поля об'єкта:
    for(int k=1;k<n;k++){
       myexp.a[k]=myexp.a[k-1]/k;
    }
    // Аргумент для обчислення значення функції і ряду:
    double x=1.0;
    // Обчислення значень для ряду й функції:
    cout<<myexp.value(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan.value(x)<<" vs. "<<tan(x)<<endl;
    cout<<f.value(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Розмір масиву `a` (який описано як поле в класі `Taylor`) визначається глобальною цілочисловою константою `n`. Конструктор класу `Taylor` описано з одним `double`-аргументом, який має значення за замовчуванням. Аргумент конструктора визначає значення елементів масиву `a`, які їм присвоюються під час створення об'єкта. Якщо під час виклику конструктора аргумент не вказано, то у всіх елементів масиву будуть нульові значення. Також є версія конструктора з аргументом, що є масивом із коефіцієнтами, які присвоюються елементам масиву `a`.

До уваги

У загальному випадку масив у функцію або метод передається за допомогою двох параметрів: вказівника на перший елемент масиву і кількості елементів у масиві. У нашому випадку всі масиви, із якими нам доведеться мати справу, мають однаковий

розмір, який визначається глобальною константою n. Тому немає необхідності передавати в конструктор другий параметр, що визначає розмір масиву. В результаті у конструктора класу Taylor один аргумент, який є вказівником на значення типу double (вказівник на перший елемент масиву).

У класі Taylor описано метод value(), призначений для обчислення значення ряду Тейлора в точці (для зазначеного аргументу). Метод повертає значення типу double, і в нього один аргумент (позначений як x) типу double. У тілі методу локальній змінній s присвоюється початкове значення 0, а локальна змінна q отримує початкове значення 1. Далі запускається оператор циклу, в якому за допомогою індексної змінної k перебираються елементи масиву a. За кожний цикл командою s+=a[k]*q збільшується значення змінної s на черговий доданок, а командою q*=x обчислюється значення для змінної q на наступній ітерації. Після закінчення обчислень значення змінної s повертається як результат методу.

Подробиці



Нам фактично необхідно обчислити суму $a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1}$, де через a_0, a_1, ..., a_{n-1} позначено елементи масиву a, а через x позначено аргумент методу value(). Суму можна записати у вигляді $a_0q_0 + a_1q_1 + \cdots + a_{n-1}q_{n-1}$. Тут ми позначили $q_k = x^k$ (аналог локальної змінної q у тілі методу value()). Початкове значення $q_0 = 1$, а для інших ітерацій неважко помітити, що $q_{k+1} = q_kx$. Саме цим співвідношенням ми скористалися, коли обчислювали значення змінної q для наступної ітерації.

У головній функції програми командою

```
double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835}
```

створюється та ініціалізується числовий масив, який ми використовуємо для передачі аргументом конструктору класу Taylor.

Подробиці



Ряд Тейлора для тангенса має вигляд $tg(x) \approx x + \frac{1}{3}x^3 + \frac{2}{15}x^5 + \frac{17}{315}x^7 + \frac{62}{2835}x^9$. Відсутність доданків із парними показниками степені аргументу означає, що числові коефіцієнти для таких доданків дорівнюють нулю. Під

час визначення коефіцієнтів масиву b числа в чисельнику вказано з десятковою крапкою, щоб уникнути цілочислового ділення.

Також у програмі використано такі наближені представлення для функцій: $\frac{1}{1-x} \approx 1 + x + x^2 + x^3 + x^4 + x^5 + x^6 + x^7 + x^8 + +x^9$ (оцінка справедлива тільки при $|x| < 1$) і $\exp(x) \approx 1 + x + \frac{x^2}{2!} + \frac{x^3}{3!} + \frac{x^4}{4!} + \frac{x^5}{5!} + \frac{x^6}{6!} + \frac{x^7}{7!} + \frac{x^8}{8!} + \frac{x^9}{9!}$.

Командою `Taylor myexp,f(1),mytan(b)` створюється три об'єкти класу `Taylor`. Під час створення об'єкта `myexp()` викликається конструктор із нульовим (значення за замовчуванням) числовим аргументом. У результаті всі елементи масиву `a` в об'єкті `myexp` отримають нульові значення. Під час створення об'єкта `f` викликається конструктор із числовим аргументом, значення якого дорівнює `1`. Таке значення буде у всіх елементів масиву `a` в об'єкті `f` (отримуємо ряд для функції $f(x) = \frac{1}{1-x}$). Нарешті, під час створення об'єкта `mytan` як аргумент конструктору передається ім'я масиву `b`, у результаті чого в елементів масиву `a` в об'єкті `mytan` будуть такі ж значення, як і в елементів у масиві `b` (ряд для тангенса).

Для об'єкта `myexp` значення коефіцієнтів масиву обчислюються за допомогою оператора циклу. Перед викликом оператора циклу командою `myexp.a[0]=1` першому елементу присвоюється одиничне значення, після чого запускається оператор циклу, в якому індексна змінна `k` перебирає елементи (індекси елементів), починаючи з другого. За кожний цикл виконується команда `myexp.a[k]=myexp.a[k-1]/k`, якою значення елемента масиву обчислюється на основі значення попереднього елемента в тому ж масиві.

Подробиці



Якщо через a_k позначити елементи масиву `a`, то тоді $a_k = \frac{1}{k!}$. Легко помітити, що $\frac{a_k}{a_{k-1}} = \frac{1}{k}$, або $a_k = \frac{a_{k-1}}{k}$.

Командою `double x=1.0` оголошується змінна `x`, яка використовується для передачі аргументом під час обчислення значення ряду. Реалізується це за

допомогою інструкцій `myexp.value(x)` (експонента в точці `x`), `mytan.value(x)` (тангенс в точці `x`) і `f.value(x/2)` (значення ряду для функції $f(x) = \frac{1}{1-x}$ у точці `x/2`). Для порівняння наводяться точні значення, обчислені за допомогою вбудованих математичних функцій `exp()` (експонента), `tan()` (тангенс), і безпосередньо обчислюється значення виразу `1/(1-x/2)`.

До уваги

Для використання математичних функцій у програмі підключається заголовок `<cmath>`.

Результат виконання програми такий:

Результат виконання програми із лістингу 6.4

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

Бачимо, що збіг досить непоганий. Для покращення точності необхідно взяти більше доданків у ряді.

Функтори й індексація об'єктів

Розглянуту вище задачу можна реалізувати у вигляді програмного коду більш ефектно. Для цього слід у класі `Taylor` описати два операторних методи:

- метод для оператора "квадратні дужки" (метод `operator[]()`), що дозволяє індексувати об'єкти (звертатися до елементів масиву `a` в об'єкті, зазначаючи індекс безпосередньо після імені об'єкта);
- метод для оператора "круглі дужки" (метод `operator()()`), що дозволяє "викликати" об'єкт як функцію.

Нову версію програми наведено в лістингу 6.5.

Лістинг 6.5. Поле-масив і перевантаження операторів

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Розмір масиву:
const int n=10;
// Клас для реалізації ряду Тейлора:
class Taylor{
private:
   // Поле - масив:
   double a[n];
public:
   // Операторний метод для індексування об'єкта:
   double &operator[](int k){
      return a[k];
   }
   // Конструктор із одним числовим аргументом:
   Taylor(double p=0){
      for(int k=0;k<n;k++){
         (*this)[k]=p; // Індексування об'єкта
      }
   }
   // Конструктор із аргументом - вказівником:
   Taylor(double* b){
      for(int k=0;k<n;k++){
         (*this)[k]=b[k]; // Індексування об'єкта
      }
   }
   // Операторний метод для обчислення значення ряду:
   double operator()(double x){
```

```
        double s=0,q=1;
        for(int k=0;k<n;k++){
            s+=(*this)[k]*q; // Індексування об'єкта
            q*=x;
        }
        return s;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Числовий масив:
    double b[n]={0,1,0,1./3,0,2./15,0,17./315,0,62./2835};
    // Створення об'єктів:
    Taylor myexp,f(1),mytan(b);
    // Індексування об'єкта:
    myexp[0]=1;
    // Заповнення масиву - поля об'єкта:
    for(int k=1;k<n;k++){
        myexp[k]=myexp[k-1]/k; // Індексування об'єкта
    }
    // Аргумент для обчислення значення функції та ряду:
    double x=1.0;
    // Обчислення значень для ряду та функції:
    cout<<myexp(x)<<" vs. "<<exp(x)<<endl;
    cout<<mytan(x)<<" vs. "<<tan(x)<<endl;
    cout<<f(x/2)<<" vs. "<<1/(1-x/2)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Код схожий на розглянутий раніше, проте є деякі особливості. Насамперед, це стосується операторного методу для квадратних дужок. Метод описано дуже просто:

```
double &operator[](int k){
   return a[k];
}
```

Метод повертає *посилання* на елемент масиву `a` з індексом, який визначається аргументом методу. Як результат повертається саме посилання на елемент (про що свідчить інструкція `&` у прототипі методу), оскільки ми хочемо не просто зчитувати значення елемента масиву, а й мати можливість присвоювати значення елементу.

До уваги

Масив `a` в класі `Taylor` тепер оголошено як закритий. Тому поза тілом класу пряме звертання до масиву `a` неможливе.

Операторний метод для обчислення значення ряду `operator()()` описано практично так само, як метод `value()` у попередньому прикладі (див. лістинг 6.4). Єдина принципова відмінність полягає в тому, що замість інструкції `a[k]` для звернення до елемента масиву використано інструкцію `(*this)[k]`, у якій взято до уваги можливість індексувати об'єкти класу `Taylor`. Подібну "заміну" виконано не лише в методі, а й у конструкторі класу.

Подробиці

Необхідності відмовлятися від інструкцій виду `a[k]` не було - ми могли, як і раніше, використовувати цю інструкцію.

Вираз `(*this)[k]` є застосуванням операторного методу `operator[]()` до об'єкта, з якого викликають операторний метод `operator()()`. Оскільки `this` - це вказівник на об'єкт, то `*this` - сам об'єкт. Якщо дописати квадратні дужки з індексом, отримуємо індексований об'єкт. Обробка такого виразу відбувається за допомогою операторного

методу `operator[]()`. Круглі дужки використовуються з урахуванням пріоритету операторів `*` і `[]` (щоб спочатку "застосовувалася" зірочка `*`, а потім квадратні дужки `[]`).

У головній функції програми змінилися команди, які виконують звертання до елементів масиву в об'єкті, і спосіб обчислення значення ряду. Так, інструкція `myexp[k]` є еквівалентом звертання до елемента з індексом `k` масиву `a` в об'єкті `myexp`. Для обчислення значень рядів використовуються вирази `myexp(x)`, `mytan(x)` і `f(x/2)`. Тут об'єкти викликаються як функції, і все завдяки тому, що в класі `Taylor` описано операторний метод `operator()()`.

До уваги

Об'єкти, які можна викликати на зразок функції, називаються *функторами*.

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 6.5)

```
2.71828 vs. 2.71828
1.5425 vs. 1.55741
1.99805 vs. 2
```

Бачимо, що результат виконання програми точно такий, як і в попередньому випадку.

Конструктор створення копії

Зазвичай під час опису класу серед його конструкторів описують, як мінімум, конструктор без аргументів і конструктор створення копії. В останньому випадку йдеться не обов'язково про створення копії об'єкта, а швидше про процедуру створення нового об'єкта на основі вже існуючого об'єкта.

Теорія

Під час створення нового об'єкта на основі вже існуючого об'єкта (того ж класу), як аргумент відповідному конструктору передається вихідний об'єкт. Важливе правило полягає в тому, що аргумент конструктору створення копії передається ***за посиланням***. За значенням аргумент передавати неможна. Під час спроби передати

аргумент за значенням для об'єкта-аргументу створюється копія. Під час створення копії викликається конструктор створення копії. Йому передається копія для копії об'єкта-аргументу. Тому знову викликається конструктор створення копії, і так до нескінченності. Виходить такий своєрідний рекурсивний виклик конструктора створення копії, що неприпустимо.

Як ілюстрацію розглянемо задачу, яка розв'язувалася в лістингу 6.2, але тепер ідеологічний підхід буде базуватися на використанні конструктора створення копії. Розглянемо програмний код, представлений у лістингу 6.6.

Лістинг 6.6. Конструктор створення копії

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Опис класу:
class MyClass{
public:
   // Символьне поле:
   char code;
   // Поле - вказівник на об'єкт:
   MyClass* next;
   // Конструктор створення копії:
   MyClass(MyClass &obj){
      // Значення поля next вихідного об'єкта:
      obj.next=this;
      // Значення поля code створюваного об'єкта:
      code=obj.code+1;
   }
   // Конструктор із символьним аргументом:
   MyClass(char s){
      code=s;
   }
   // Деструктор:
```

```
    ~MyClass(){
        if(next){ // Якщо не останній об'єкт
            // Видалення наступного об'єкта:
            delete next;
        }
        cout<<"Об'єкт із полем "<<code<<" видалено\n";
    }
    // Метод для відображення значення символьного поля:
    void show(){
        // Відображення значення поля:
        cout<<code<<" ";
        if(next){
            // Виклик методу для наступного об'єкта:
            next->show();
        }
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Цілочислова змінна:
    int n=10;
    // Динамічне створення об'єкта:
    MyClass* pnt=new MyClass('A');
    // Вказівник на об'єкт:
    MyClass *p;
    // Початкове значення вказівника:
    p=pnt;
    // Створення ланцюжка об'єктів:
    for(int k=1;k<=n;k++){
```

```
        // Створення чергового об'єкта:
        p=new MyClass(*p);
    }
    // Нульове посилання в полі next для останнього
    // об'єкта в ланцюжку:
    p->next=0;
    // Виклик методу для першого об'єкта в ланцюжку:
    pnt->show();
    cout<<endl;
    // Видалення першого об'єкта (із автоматичним
    // видаленням всього ланцюжка об'єктів):
    delete pnt;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 6.6)

```
A B C D E F G H I J K
Об'єкт із полем K видалено
Об'єкт із полем J видалено
Об'єкт із полем I видалено
Об'єкт із полем H видалено
Об'єкт із полем G видалено
Об'єкт із полем F видалено
Об'єкт із полем E видалено
Об'єкт із полем D видалено
Об'єкт із полем C видалено
Об'єкт із полем B видалено
Об'єкт із полем A видалено
```

Бачимо, що результат виконання програми (порівняно з прикладом із лістингу 6.2) не змінився. Програмний код, тим не менше, зазнав значних змін. Проаналізуємо їх.

У класі `MyClass` тепер є конструктор із символьним аргументом, що визначає значення поля `code` створюваного об'єкта. Ще в класі `MyClass` описано конструктор створення копії. У цій версії конструктора один аргумент (позначено як `obj`), і це об'єкт класу `MyClass`. Аргумент передається за посиланням (інструкція `&` в описі аргументу конструктора). У тілі конструктора командою `obj.next=this` значенню поля `next` об'єкта, переданого конструктору, присвоюється адреса нового (створюваного) об'єкта. Ми виходимо з того, що в ланцюжку об'єктів новий об'єкт створюється на основі попереднього. Саме "попередній" об'єкт передається як аргумент конструктору під час створення нового об'єкта. Тому логічно, якщо в поле `next` об'єкта, на основі якого створюється новий, записується адреса створеного об'єкта.

Для поля `code` створюваного об'єкта значення присвоюється командою `code=obj.code+1`. Отже, в новому об'єкті значення поля `code` на одиницю "більше" за значення поля `code` в об'єкті, переданому конструктору. Оскільки йдеться про символьні поля, то насправді отримуємо наступний символ у кодовій таблиці символів.

У попередній версії програми для видалення ланцюжка об'єктів описувалася спеціальна функція. У даному випадку аналогічна задача розв'язується за допомогою деструктора. Ідея полягає в тому, що під час видалення об'єкта видаляється і весь ланцюжок об'єктів, із ним пов'язаних. Реалізується все в деструкторі за допомогою умовного оператора, умовою в якому перевіряється значення виразу `next`. Значення цього виразу дорівнює нулю й інтерпретується як `false`, якщо поле `next` поточного об'єкта дорівнює нулю, і, значить, об'єкт останній у ланцюжку. У цьому випадку команда в умовному операторі ігнорується і просто виводиться повідомлення за допомогою інструкції `cout<<"Об'єкт із полем "<<code<<" видалено\n"`. Таким чином, якщо деструктор викликається для останнього об'єкта в ланцюжку, то з'являється повідомлення, і об'єкт буде видалено (оскільки для нього викликано деструктор,

який "виходить на сцену" під час видалення об'єкта). Якщо ж значення поля `next` відмінне від нуля, то в умовному операторі командою `delete next` виконується спроба видалити об'єкт, адресу якого записано в полі `next`. Для цього об'єкта викликається деструктор, і якщо об'єкт у ланцюжку не останній, то викликаний деструктор викликає деструктор для наступного об'єкта, і так аж до останнього об'єкта в ланцюжку. Коли видаляється останній об'єкт, буде видалено передостанній об'єкт, потім об'єкт перед ним, і так аж першого об'єкта, для якого викликався найперший деструктор.

У функції `main()` перший динамічний об'єкт створюється командою `MyClass* pnt=new MyClass('A')`. Початкове значення вказівника `p` збігається зі значенням вказівника `pnt`. Далі в операторі циклу під час створення ланцюжка об'єктів виконується одна й та сама команда `p=new MyClass(*p)`, якою на основі поточного об'єкта (доступ до нього отримуємо за допомогою інструкції `*p`) створюється новий об'єкт у ланцюжку, і вказівник `p` "перекидається" на цей новий об'єкт. Полю `next` останнього об'єкта в ланцюжку командою `p->next=0` присвоюємо нульове значення. Командою `pnt->show()` відображається значення полів `code` всіх об'єктів у ланцюжку, а видалення першого об'єкта за допомогою команди `delete pnt` спричинює автоматичне видалення всіх об'єктів у ланцюжку.

Успадкування та закриті поля базового класу

Наступний невеликий приклад ілюструє ситуацію, коли в базовому класі є закриті члени, які не успадковуються в похідному класі, але при цьому використовуються в успадкованих відкритих методах.

Теорія

Стверджується, що закриті члени базового класу в похідному класі не успадковуються. Це так: у похідному класі неможливо напряму (тобто явно) звернутися до членів класу, які в базовому класі є закритими. При цьому "технічно" ці члени класу існують і в похідному класі. Тому, наприклад, якщо похідний клас

успадковує метод із базового класу, а цей метод у базовому класі, у свою чергу, звертається до закритих членів, то ситуація залишиться коректною і в похідному класі.

Розглянемо програмний код, представлений у лістингу 6.7.

Лістинг 6.7. Закриті члени базового класу та успадкування

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовий клас:
class Alpha{
private:
   // Закрите поле:
   char symb;
public:
   // Конструктор класу:
   Alpha(char s){
      symb=s;
   }
   // Метод для відображення значення поля:
   void show(){
      cout<<"Клас Alpha: "<<symb<<endl;
   }
   // Метод для присвоювання значення полю:
   void set(char s){
      symb=s;
   }
};
// Похідний клас:
class Bravo: public Alpha{
public:
   // Відкрите поле:
   int num;
```

```
    // Конструктор класу:
    Bravo(char s,int n):Alpha(s){
        num=n;
    }
    // У методі викликається успадкований метод:
    void showAll(){
        show();
        cout<<"Клас Bravo: "<<num<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єкта похідного класу:
    Bravo obj('A',100);
    // Відображення параметрів об'єкта:
    obj.showAll();
    // Виклик успадкованого методу:
    obj.set('Z');
    // Відображення параметрів об'єкта:
    obj.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 6.7)

```
Клас Alpha: A
Клас Bravo: 100
Клас Alpha: Z
```

```
Клас Bravo: 100
```

Програма достатньо проста, але деякі моменти все ж прокоментувати варто. Отже, у класі `Alpha` описано закрите символьне поле `symb`. У класі є конструктор із одним аргументом (значення для поля) і два відкритих методи. Метод `show()` відображує значення поля `symb`, а метод `set()` призначений для присвоювання значення полю. Клас `Bravo` створюється шляхом успадкування класу `Alpha`. У класі `Bravo` успадковуються методи `set()` і `show()`, але не успадковується поле `symb`. Також у класі `Bravo` описано відкрите цілочислове поле `num`. У конструктора класу `Bravo` два аргументи: другий аргумент визначає значення поля `num`, а перший символьний аргумент передається конструктору базового класу. Тут треба врахувати, що хоча поле `symb` у похідному класі `Bravo` не успадковується, пам'ять під поле виділяється, і ті методи (у тому числі й конструктор) із базового класу, які звертаються до цього поля й успадковуються в похідному класі, можуть звертатися до відповідної області пам'яті.

До уваги

Простіше кажучи, "не успадковування" поля (чи методу) означає, що в похідному класі до нього неможна звернутися за іменем, але можна звернутися не напряму.

У класі `Bravo` описано метод `showAll()`, у якому, крім іншого, викликається успадкований із класу `Alpha` метод `show()`, а він, у свою чергу, звертається до поля `symb` (яке не успадковується).

У функції `main()` командою `Bravo obj('A',100)` створюється об'єкт класу `Bravo`. Хоча поля `symb` у об'єкта немає, під час виконання команди `obj.showAll()` для цього поля відображується значення `'A'`. Більше того, після присвоювання значення командою `obj.set('Z')` "неіснуючому" полю перевірка (за допомогою команди `obj.showAll()`) показує, що значення "неіснуючого" поля змінилося. Але нічого дивного тут насправді немає. Як зазначалося вище, місце в пам'яті під поле `symb` під час створення об'єкта

похідного класу виділяється, просто ми до цієї області пам'яті не маємо прямого доступу.

Віртуальні методи та успадкування

Перш ніж почати обговорення суті наступної проблеми, розглянемо виключно простий програмний код, наведений у лістингу 6.8.

Лістинг 6.8. Невіртуальний метод

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовий клас:
class Alpha{
public:
    // Звичайний (невіртуальний) метод:
    void show(){
        cout<<"Клас Alpha"<<endl;
    }
    // У методі викликається метод:
    void showAll(){
        show();
    }
};
// Похідний клас:
class Bravo: public Alpha{
public:
    // Перевизначення методу:
    void show(){
        cout<<"Клас Bravo"<<endl;
    }
};
// Головна функція програми:
int main(){
```

```
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Об'єкт похідного класу:
    Bravo obj;
    // Виклик методів:
    obj.show();
    obj.showAll();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 6.8)

```
Клас Bravo
Клас Alpha
```

Ситуація виключно проста: у класі `Alpha` описано метод `show()`, який виводить повідомлення в консольне вікно. У цьому ж класі описано метод `showAll()`, у тілі якого викликається метод `show()`.

На основі класу `Alpha` шляхом успадкування створюється клас `Bravo`. Зрозуміло, що в класі `Bravo` успадковуються методи `show()` і `showAll()`. Тим не менше, метод `show()` у класі `Bravo` перевизначається - змінюється зміст повідомлення, що відображується в консольному вікні.

У головній функції програми створюється об'єкт `obj` похідного класу `Bravo`, і з нього послідовно викликаються методи `show()` і `showAll()`. Результат виконання команди `obj.show()` очикуваний - у консоль виводиться повідомлення `Клас Bravo`, відповідно до того, як перевизначено метод `show()` у класі `Bravo`. Та ось під час виконання команди `obj.showAll()` з'являється повідомлення `Клас Alpha`. Причина очевидна: метод `showAll()` викликає ту версію методу `show()`, яку описано в базовому

класі, а не перевизначену версію з похідного класу. Щоб змінити ситуацію, метод `show()` у базовому класі треба зробити *віртуальним*.

Опис віртуального методу починається з ключового слова `virtual`. Наприклад, щоб описати в базовому класі `Alpha` метод `show()` як віртуальний, використовуємо такий код:

```
virtual void show(){
   cout<<"Клас Alpha"<<endl;
}
```

З урахуванням змін, програма буде виглядати так (практично всі коментарі видалено, а важливі місця виділено жирним шрифтом):

Лістинг 6.9. Віртуальний метод

```
#include <iostream>
#include <cstdlib>
using namespace std;
class Alpha{
public:
   // Віртуальний метод:
   virtual void show(){
      cout<<"Клас Alpha"<<endl;
   }
   void showAll(){
      show();
   }
};
class Bravo: public Alpha{
public:
   void show(){
      cout<<"Клас Bravo"<<endl;
   }
};
int main(){
```

```
    system("chcp 1251>nul");
    Bravo obj;
    obj.show();
    obj.showAll();
    system("pause>nul");
    return 0;
}
```

Результат виконання програми тепер такий:

Результат виконання програми (із лістингу 6.9)

```
Клас Bravo
Клас Bravo
```

Таким чином, завдяки тому, що метод `show()` тепер віртуальний, під час виклику методу `showAll()` із об'єкта похідного класу викликається перевизначена в класі `Bravo` версія методу `show()`.

До уваги

Методи, які перевизначаються в похідних класах, в базових класах зазвичай оголошують віртуальними.

Множинне успадкування

У C++ похідний клас може створюватися на основі одразу декількох базових класів. Про це наступний приклад, поданий у лістингу 6.10.

Лістинг 6.10. Множинне успадкування

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Перший базовий клас:
class Alpha{
public:
    // Поле:
```

```
    int alpha;
    // Конструктор:
    Alpha(int a){
        alpha=a;
    }
    // Метод:
    void show(){
        cout<<"Клас Alpha: "<<alpha<<endl;
    }
};
// Другий базовий клас:
class Bravo{
public:
    // Поле:
    int bravo;
    // Конструктор:
    Bravo(int b){
        bravo=b;
    }
    // Метод:
    void show(){
        cout<<"Клас Bravo: "<<bravo<<endl;
    }
};
// Похідний клас:
class Charlie:public Alpha,public Bravo{
public:
    // Поле:
    int charlie;
    // Конструктор:
    Charlie(int a,int b,int c):Alpha(a),Bravo(b){
        charlie=c;
    }
```

```
    // Перевизначення методу:
    void show(){
        // Виклик версії методу з класу Alpha:
        Alpha::show();
        // Виклик версії методу з класу Bravo:
        Bravo::show();
        // Відображення значення поля:
        cout<<"Клас Charlie: "<<charlie<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єкта похідного класу:
    Charlie obj(10,20,30);
    // Виклик методу:
    obj.show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 6.10)

```
Клас Alpha: 10
Клас Bravo: 20
Клас Charlie: 30
```

Тепер проаналізуємо програмний код. У програмі описуються класи `Alpha` і `Bravo`. Вони стають базовими для класу `Charlie`, який створюється на їхній основі. Зокрема, в класі `Alpha` є цілочислове поле `alpha`, конструктор із

одним аргументом і метод show(), який під час виклику відображує значення поля alpha об'єкта, з якого викликається метод. Аналогічна структура й у класу Bravo, з поправкою на те, що його цілочислове поле називається bravo. У класу Bravo також є метод show(), що виконує аналогічну місію (як і метод show() у класі Alpha).

Клас Charlie створюється шляхом відкритого успадкування на основі класів Alpha і Bravo. Тому в описі класу Charlie після імені класу через двокрапку вказуються назви базових класів і тип спадкування (ключове слово public) для кожного класу. Безпосередньо в класі Charlie описано цілочислове поле charlie і звичайно, успадковуються поля alpha і bravo.

У конструктора класу Charlie три цілочислових аргументи: два передаються в конструктори базових класів, а ще один аргумент задає значення поля charlie. В описі похідного класу після імені конструктора через двокрапку вказують інструкції виклику конструкторів базових класів (із передачею їм аргументів).

Метод show() у класі Charlie перевизначається. У тілі методу show() у класі Charlie командами Alpha::show() і Bravo::show() послідовно викликаються версії методу із класу Alpha і класу Bravo. У цьому випадку для ідентифікації версії методу використовується назва класу, в якому визначено відповідну версію методу. Між іменем класу й іменем методу розміщується оператор розширення області видимості : : (подвійна двокрапка). Після виклику версій методу show() із класів Alpha і Bravo (у результаті чого в консолі відображуються значення полів alpha і bravo) командою cout<<"Клас Charlie: "<<charlie<<endl відображається значення поля charlie. У головній функції програми створюється об'єкт похідного класу і для перевірки значень його полів викликається метод show().

Доступ до об'єктів через змінну базового класу

Є важлива властивість, яка "пов'язує" об'єкти базових класів і об'єкти похідних класів: змінній для об'єкта базового класу як значення можна присвоїти об'єкт похідного класу. Цю важливу і фундаментальну можливість проілюстровано в програмному коді, наведеному в лістингу 6.11.

Лістинг 6.11. Об'єкти базових і похідних класів

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Перший базовий клас:
class Alpha{
public:
   // Поле:
   char codeA;
   // Конструктор:
   Alpha(char a){
      codeA=a;
   }
   // Віртуальний метод:
   virtual void show(){
      cout<<"Метод із класу Alpha: "<<codeA<<endl;
   }
};
// Другий базовий клас:
class Bravo{
public:
   // Поле:
   char codeB;
   // Конструктор:
   Bravo(char b){
```

```
        codeB=b;
    }
    // Віртуальний метод:
    virtual void show(){
        cout<<"Метод із класу Bravo: "<<codeB<<endl;
    }
};
// Похідний клас:
class Charlie:public Alpha,public Bravo{
public:
    // Конструктор із пустим тілом:
    Charlie(char a,char b):Alpha(a),Bravo(b){}
    // Перевизначення методу:
    void show(){
        cout<<"Метод із класу Charlie: ";
        cout<<codeA<<codeB<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    cout<<"Використовуємо змінні\n";
    // Створення об'єктів:
    Alpha objA('A');
    objA. show();
    Bravo objB('B');
    objB. show();
    Charlie objC('C','D');
    objC. show();
    // Присвоювання об'єктним змінним базових класів
    // значенням об'єкта похідного класу:
```

```
    objA=objC;
    objB=objC;
    objA.show();
    objB.show();
    cout<<"Використовуємо вказівники\n";
    // Оголошення вказівників:
    Alpha* pntA=&objC;
    Bravo* pntB=&objC;
    Charlie* pntC=&objC;
    // Метод викликається через вказівники:
    pntA->show();
    pntB->show();
    pntC->show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У результаті виконання програми отримуємо таке:

Результат виконання програми (із лістингу 6.11)

```
Використовуємо змінні
Метод із класу Alpha: A
Метод із класу Bravo: B
Метод із класу Charlie: CD
Метод із класу Alpha: C
Метод із класу Bravo: D
Використовуємо вказівники
Метод із класу Charlie: CD
Метод із класу Charlie: CD
Метод із класу Charlie: CD
```

Програмний код заслуговує на деякі пояснення. Класи Alpha і Bravo є базовими для класу Charlie. У класі Alpha є символьне поле codeA, конструктор з одним аргументом і метод show(), який описано як віртуальний (із ключовим словом virtual). Аналогічний метод є в класі Bravo. Ще там є символьне поле codeB і конструктор з одним аргументом. Клас Charlie успадковує класи Alpha і Bravo. У конструктора класу Charlie два аргументи, які передаються конструкторам базових класів. Оскільки інші дії в конструкторі не виконуються, то тіло конструктора пусте (у фігурних дужках немає команд). Метод show() у класі Charlie перевизначається так, що послідовно відображаються значення двох полів: codeA і codeB.

Усе найцікавіше відбувається у функції main(). Там спочатку створюється три об'єкти objA, objB і objC відповідно класів Alpha, Bravo і Charlie. Для кожного з цих об'єктів викликається метод show(). На цьому етапі все відбувається "штатно". Потім виконуються команди objA=objC і objB=objC, якими об'єктним змінним objA і objB базових класів Alpha і Bravo як значення присвоюється об'єкт objC похідного класу Charlie. Оскільки клас Charlie є похідним від класів Alpha і Bravo, такі присвоювання законні. Однак є важливе обмеження: через змінну базового класу можна отримати доступ до тих лише полів і методів, які оголошено в цьому базовому класі. Тому під час виклику через змінну objA отримуємо доступ тільки до поля codeA і версії методу show(), оголошеній у класі Alpha. Через змінну objB отримуємо доступ до поля codeB і версії методу show() із класу Bravo.

Інша справа, якщо ми використовуємо вказівники. У програмі оголошено три вказівники: вказівник pntA на об'єкт класу Alpha, вказівник pntB на об'єкт класу Bravo і вказівник pntC на об'єкт класу Charlie. Кожному з вказівників як значення присвоюється адреса об'єкта objC похідного класу Charlie (інструкція &objC). Дана операція також законна, і все завдяки тому, що класи Alpha і Bravo є базовими для класу Charlie. Пікантність ситуації

в тому, що якщо ми викликаємо метод `show()` через вказівники `pntA` і `pntB`, то викликається версія методу, описана в класі `Charlie`.

Подробиці



Як і у випадку з об'єктними змінними, через вказівники на об'єкти базового класу є доступ лише до тих полів і методів об'єкта похідного класу, які описано в базовому класі. Простіше кажучи, через вказівник `pntA` отримуємо доступ лише до поля `codeA`, а через вказівник `pntB` є доступ лише до поля `codeB`. Проте якщо через такий вказівник викликається перевизначений метод, а вихідну версію методу описано як віртуальну, то буде викликано саме перевизначену версію методу.

Задачі для самостійного розв'язання

1. Напишіть програму, в якій створюється масив об'єктів. У кожного об'єкта є закрите числове поле і метод, що повертає значення цього поля. Під час створення об'єкта полю присвоюється випадкове ціле число.

2. Напишіть програму з класом, у якого є поле, що являє собою числовий масив. Передбачити різні способи створення об'єктів класу (в тому числі на основі вже існуючого об'єкта). Описати операторні функції та методи для виконання різноманітних операцій з об'єктами (наприклад, додавання об'єктів, індексування об'єктів, множення і ділення на число). Наприклад, під час додавання об'єктів результатом може бути об'єкт із масивом, елементи якого визначаються як середнє арифметичне відповідних елементів у масивах об'єктів, які додають. Під час виклику об'єкта як результат може повертатися середнє арифметичне значення елементів масиву із даного об'єкта і так далі.

3. Напишіть програму, в якій створюється "кільце" з об'єктів: "ланцюжок" об'єктів, у якому один об'єкт посилається на інший, а останній об'єкт посилається на перший. Кожен об'єкт має, крім іншого, закрите числове поле з випадковим значенням, і закрите символьне поле теж з випадковим значенням. Запропонувати функції та методи для роботи з таким "кільцем": наприклад,

метод для відображення значень полів об'єктів, які входять у "кільце", або функцію для видалення об'єктів в "кільці". Рекомендується розв'язати задачу різними способами.

4. Напишіть програму, в якій створюється "бінарне дерево" об'єктів: кожний об'єкт посилається на два інших об'єкта. Запропонувати алгоритм створення такого "дерева", методи для ідентифікації об'єктів у ньому й способи видалення "дерева".

5. Напишіть програму з класом, у якого полем є масив вказівників на об'єкти того ж класу. Запропонуйте методи та функції для роботи з такими об'єктами.

Рекомендації для самостійної роботи

• Спробуйте розв'язати задачі, розглянуті в главі, використовуючи замість статичних масивів динамічні масиви і динамічний спосіб виділення пам'яті (зрозуміло, там, де це доречно).

• Спробуйте розв'язати задачі, розглянуті в попередніх главах (і запропоновані там для самостійного розв'язання), використовуючи замість масивів індексовані об'єкти.

• У прикладах із використанням функцій, розглянутих у попередніх главах, спробуйте замінити функції на функтори.

• Відомо, що функція не може як результат повертати статичний масив. Разом із тим, результатом функції може бути об'єкт, який допускає індексування. У такому випадку створюється ілюзія, що функція повертає масив. Також індексований об'єкт може бути результатом, який повертає функтор. Подумайте і запропонуйте способи розв'язання деяких із розглянутих раніше задач із використанням цієї технології.

Глава 7

Узагальнені функції та класи

Ну й брехун. Дід мій - брехун. А ти його перебрехав.

з х/ф "Пропала грамота"

Список тем, обговорюваних у цій главі, досить короткий. Разом з тим, самі по собі теми дуже місткі й цікаві. Отже, далі ми обговоримо і розглянемо:

- узагальнені функції;
- узагальнені класи,

ну і ще деякі супутні підходи й механізми.

Узагальнені функції

Ідея *узагальнених функцій* полягає в тому, що в них тип даних виступає параметром.

Теорія

Опис ***узагальненої функції*** починається з ключового слова `template`, після якого в кутових дужках зазначають ключове слово `class` і формальне позначення для типу даних (параметр типу або узагальнений параметр). У всьому іншому узагальнена функція описується як звичайна (не узагальнена) функція, з поправкою на те, що в описанні функції як ідентифікатор типу можна використовувати параметр.

У лістингу 7.1 наведено програму, що містить узагальнену функцію `show()`. Ця функція в консольному вікні відображує значення аргументу, який передається функції під час виклику.

Лістинг 7.1. Узагальнена функція з одним параметром типу

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Узагальнена функція:
```

```
template<class X> void show(X arg){
    cout<<arg<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Виклик узагальненої функції з символьним аргументом:
    show('A');
    // Виклик узагальненої функції з числовим аргументом:
    show(123);
    // Виклик узагальненої функції з текстовим аргументом:
    show("Текст");
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання функції такий:

Результат виконання програми (із лістингу 7.1)

```
A
123
Текст
```

Найцікавіше тут, безумовно - це опис узагальненої функції. Ось так виглядає код, що нас цікавить:

```
template<class X> void show(X arg){
    cout<<arg<<endl;
}
```

Опис узагальненої функції починається з ключового слова `template`. Інструкція `<class X>`, що йде далі, означає, що ідентифікатор `X` в описі функції позначає деякий тип даних. Який саме - стає зрозуміло під час виклику

функції. Функція, як бачимо, не повертає результат (ключове слово `void` перед іменем функції), і у неї один аргумент (позначений як `arg`). Тип аргументу визначається ідентифікатором `X`. У тілі функції всього одна команда `cout<<arg<<endl`, відповідно до якої значення аргументу відображується в консольному вікні (і виконується перехід до нового рядка).

У головній функції програми функція `show()` викликається тричі: йдеться про команди `show('A')`, `show(123)` і `show("Текст")`. Кожного разу під час виклику функції за типом переданого їй аргументу визначається значення для узагальненого параметра `X`, і код функції виконується при відповідному значенні даного параметра. Наприклад, у команді `show('A')` функція `show()` отримує як аргумент значення типу `char`. Тому код функції виконується так, як наче б параметр `X` був ідентифікатором `char`. Під час виконання команди `show(123)` параметр `X` "стає" ідентифікатором `int`, а під час виконання команди `show("Текст")` параметр `X` інтерпретується як вказівник на символьне значення (умовно кажучи, тип `char*`).

У наступній програмі використання узагальнених функцій носить більш витончений характер. Зокрема, у програмі описуються дві узагальнені функції: функція `show()` призначена для відображення вмісту масиву, а функція `sort()` використовується для сортування масиву методом бульбашки. Причому в тілі функції `sort()` викликається функція `show()`.

До уваги

Під час сортування масиву (в порядку збільшення значень елементів) методом бульбашки послідовно порівнюються значення сусідніх елементів масиву, і якщо значення елемента ліворуч більше за значення елемента праворуч, то вони обмінюються значеннями. Кількість повних "проходів" масиву з "перебиранням" елементів на одиницю менше за кількість елементів у масиві. За кожен наступний прохід кількість елементів, що перевіряються, зменшується на одиницю.

Розглянемо програмний код, наведений у лістингу 7.2.

Лістинг 7.2. Узагальнена функція для сортування масиву

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Узагальнена функція для відображення масиву:
template<class T> void show(T* m,int n){
   for(int i=0;i<n;i++){
      cout<<m[i]<<" ";
   }
   cout<<endl;
}
// Узагальнена функція для сортування масиву:
template<class X> void sort(X* m,int n){
   // Відображення несортованого масиву:
   show(m,n);
   // Локальна змінна узагальненого типу:
   X s;
   // Вкладені оператори циклу:
   for(int i=1;i<=n-1;i++){
      for(int j=0;j<n-i;j++){
         if(m[j]>m[j+1]){
            // Обмін значеннями сусідніх елементів:
            s=m[j+1];
            m[j+1]=m[j];
            m[j]=s;
         }
      }
   }
   // Відображення відсортованого масиву:
   show(m,n);
}
```

```
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Числовий масив:
    int A[5]={3,2,8,1,0};
    // Символьний масив:
    char B[7]={'Z','B','Y','A','D','C','X'};
    // Сортування масивів:
    sort(A,5);
    sort(B,7);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 7.2)

```
3 2 8 1 0
0 1 2 3 8
Z B Y A D C X
A B C D X Y Z
```

В описі узагальненої функції `show()` параметр типу позначено як `T`. Аргументів у функції два: перший аргумент `m` описано з ідентифікатором типу `T*`. Це означає, що аргумент `m` є вказівником на значення того типу, який "ховається" за ідентифікатором `T`. Неважко здогадатися, що під `m` мається на увазі ім'я масиву, елементи якого належать до типу `T`. Другий аргумент `n` функції `show()` цілочисловий. Він визначає розмір масиву, що передається у функцію. У тілі функції запускається оператор циклу, в якому аргумент `m` оброблюється як масив: у консольне вікно виводяться значення його елементів.

Узагальнену функцію `sort()` для сортування масиву описано з параметром типу `X`. Вона не повертає результат, і у неї, як і у функції `show()`, два аргументи. Перший аргумент (позначено як `m`) формально є вказівником на значення типу `X`, але ми припускаємо насправді, що це ім'я масиву, елементи якого належать до типу `X`. Другий цілочисловий аргумент `n` є розміром масиву. У тілі функції спочатку командою `show(m,n)` викликається узагальнена функція `show()`, у результаті чого в консольному вікні відображується вміст не відсортованого ще масиву. Потім оголошується локальна змінна `s`, яка належить до узагальненого типу `X`. Змінна нам знадобиться, коли ми у вкладених операторах циклу під час перебирання і порівняння елементів робитимемо "обмін" значеннями. Після того, як масив відсортовано, командою `show(m,n)` відображається новий порядок елементів у масиві (точніше, новий порядок значень елементів у масиві).

У функції `main()` командами `int A[5]={3,2,8,1,0}` і `char B[7]={'Z','B','Y','A','D','C','X'}` створюються цілочисловий і символьний масиви. Для їх сортування використано відповідно команди `sort(A,5)` і `sort(B,7)`.

До уваги

Під час порівняння значень елементів символьних масивів порівнюються коди відповідних символів.

Узагальнена функція з декількома параметрами

В узагальненій функції може бути декілька параметрів типу. Як ілюстрацію розглянемо програму, наведену в лістингу 7.3.

Лістинг 7.3. Узагальнена функція з декількома параметрами

```
#include <iostream>
```

```
#include <cstdlib>
using namespace std;
// Узагальнена функція з двома параметрами:
template<class X,class R> R apply(R (*fun)(X),X arg){
   return fun(arg);
}
// Звичайна функція (аргумент типу double
// і результат типу double):
double f(double x){
   return x*(1-x);
}
// Звичайна функція (аргумент типу int
// і результат типу int):
int factorial(int n){
   if(n==0){
      return 1;
   }
   else{
      return n*factorial(n-1);
   }
}
// Звичайна функція (аргумент типу int
// і результат типу char):
char symb(int n){
   return 'A'+n;
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Виклик узагальненої функції:
   cout<<apply(f,0.5)<<endl;
```

```
    cout<<apply(factorial,5)<<endl;
    cout<<apply(symb,3)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У програмі описано узагальнену функцію `apply()`, призначену для застосування певної (звичайної) функції, переданої першим аргументом функції `apply()`, до аргументу, який передано другим аргументом функції `apply()`. Опис узагальненої функції виглядає так:

```
template<class X,class R> R apply(R (*fun)(X),X arg){
    return fun(arg);
}
```

Функцію оголошено з двома параметрами типу `X` і `R`. Як результат функції повертається значення типу `R`. Аргументів у функції два: другий аргумент `arg` належить до типу `X`, а перший аргумент функції `apply()` є вказівником на функцію, в якої один аргумент типу `X` і яка повертає як результат значення типу `R`. Інструкція, що описує перший аргумент, виглядає як `R (*fun)(X)`.

До уваги

Нагадаємо, що вказівник на функцію описується таким чином:

- зазначається ідентифікатор типу результату функції;
- у круглих дужках зазначається назва змінної-вказівника, перед назвою - зірочка *;
- у круглих дужках через кому перераховуються ідентифікатори типу для аргументів функції.

Результат узагальненої функції `apply()` повертається командою `return fun(arg)`. Фактично, тут як результат узагальненої функції повертається значення, яке обчислюється під час застосування функції, ім'я якої передано першим аргументом узагальненої функції `apply()`, до значення, переданого другим аргументом функції `apply()`.

Для перевірки роботи функції `apply()` у програмі оголошено декілька звичайних функцій (головна умова - щоб вони повертали результат, і щоб у них був один аргумент). Зокрема:

- у функції `f()` аргумент типу `double` і результат типу `double`;
- функція для обчислення факторіала числа `factorial()` повертає значення типу `int`, і аргумент функції також належить до типу `int`;
- функція `symb()` повертає як результат символ (тип `char`), а аргумент функції має тип `int`.

У головній функції програми інструкціями `apply(f,0.5)`, `apply(factorial,5)` і `apply(symb,3)` викликається узагальнена функція `apply()`, як аргумент їй передається ім'я звичайної (неузагальненої функції) і значення, використовуване під час її виклику. Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 7.3)

```
0.25
120
D
```

Таким чином, як результат функція `apply()` повертає те значення, яке є результатом переданої аргументом функції.

Перевантаження узагальненої функції

Істина проста - можна створити декілька версій узагальненої функції. Приклад такої ситуації наведено у лістингу 7.4. Там описано дві версії узагальненої функції `show()`: із одним аргументом і з двома аргументами. Сама функція дуже проста: під час її виклику відображуються значення аргументів. Завдяки тому, що типи аргументів ідентифікуються через узагальнені параметри, у програмі функцію `show()` можна викликати з одним

або двома аргументами довільного типу (але такого, щоб мала сенс команда виведення значення відповідного аргументу в консольне вікно). Розглянемо наведений нижче програмний код:

Лістинг 7.4. Перевантаження узагальненої функції

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Версія узагальненої функції з одним аргументом:
template<class X> void show(X x){
   cout<<"Функція з одним аргументом\n";
   cout<<"Значення аргументу: "<<x<<endl;
}
// Версія узагальненої функції з двома аргументами:
template<class X,class Y> void show(X x,Y y){
   cout<<"Функція з двома аргументами\n";
   cout<<"Перший аргумент: "<<x<<endl;
   cout<<"Другий аргумент: "<<y<<endl;
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Виклик узагальненої функції з одним аргументом:
   show('A');
   show(123);
   show("Текст");
   // Виклик узагальненої функції з двома аргументами:
   show(321,"Текст");
   show('B',456);
   show('C','D');
   // Затримка консольного вікна:
```

```
    system("pause>nul");
    return 0;
}
```

Під час виконання програми отримуємо такий результат:

Результат виконання програми (із лістингу 7.4)

```
Функція з одним аргументом
Значення аргументу: A
Функція з одним аргументом
Значення аргументу: 123
Функція з одним аргументом
Значення аргументу: Текст
Функція з двома аргументами
Перший аргумент: 321
Другий аргумент: Текст
Функція з двома аргументами
Перший аргумент: B
Другий аргумент: 456
Функція з двома аргументами
Перший аргумент: C
Другий аргумент: D
```

Гадаємо, особливих пояснень з приводу цього програмного коду і результатів його виконання не потрібно.

Явна спеціалізація узагальненої функції

Під час роботи з узагальненими функціями можна створювати спеціальні версії функції для певних значень узагальнених параметрів. У такому випадку говорять про *явну спеціалізацію* узагальненої функції. Приклад такої ситуації наведено у лістингу 7.5.

Лістинг 7.5. Явна спеціалізація узагальненої функції

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Опис класу:
class MyClass{
public:
   int number;
   MyClass(int n){
      number=n;
   }
   void show(){
      cout<<"Об'єкт класу MyClass: "<<number<<endl;
   }
};
// Узагальнена функція:
template<class X> void show(X arg){
   cout<<"Значення аргументу функції: "<<arg<<endl;
}
// Явна спеціалізація узагальненої функції:
template<> void show<int>(int arg){
   cout<<"Цілочисловий аргумент: "<<arg<<endl;
}
template<> void show<MyClass>(MyClass obj){
   obj.show();
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   MyClass obj(300);
```

```
    // Виклик узагальненої функції:
    show('A');
    show(100.0);
    show(200);
    show(obj);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 7.5)

```
Значення аргументу функції: A
Значення аргументу функції: 100
Цілочисловий аргумент: 200
Об'єкт класу MyClass: 300
```

У програмі описано узагальнену функцію `show()` з одним аргументом, тип якого задано через узагальнений параметр. Функція під час виклику відображає значення аргументу. Разом з тим, у програмі також в явному вигляді описано дві спеціалізації функції для випадку, коли аргумент є цілим числом (значення типу `int`) і коли аргумент є об'єктом класу `MyClass`. Цей клас описано в програмі таким чином, що в об'єктів класу `MyClass` є поле `number` і метод `show()`, під час виклику якого відображується значення поля `number`. Явна спеціалізація функції `show()` для аргументу, що є об'єктом класу `MyClass`, передбачає, що з об'єкта, переданого аргументом функції `show()`, викликається однойменний метод.

Стосовно технічного питання щодо явної спеціалізації узагальненої функції, то починається опис явної спеціалізації з ключового слова `template`, після якого йдуть пусті кутові дужки <>. Далі йде звичайний опис функції, за тим

винятком, що після імені функції в кутових дужках зазначається ідентифікатор типу, для якого виконується явна спеціалізація функції.

Узагальнені класи

Аналогічно до узагальнених функцій можна створювати класи, у яких ідентифікатори типів даних є параметрами.

Теорія

Опис ***узагальненого класу*** починається так само, як і опис узагальненої функції: з ключового слова `template`, після якого в кутових дужках перераховуються параметри для узагальнених типів (кожен параметр зазначається з ключовим словом `class`). Потім йде опис класу, в якому можна використовувати параметри узагальнених типів.

Під час створення об'єкта на основі узагальненого класу після імені класу в кутових дужках зазначаються ідентифікатори типу, які слід використовувати замість узагальнених параметрів.

У лістингу 7.6 наведено приклад опису й використання узагальненого класу.

Лістинг 7.6. Узагальнений клас

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Узагальнений клас:
template<class A,class B> class MyClass{
public:
   // Поля класу:
   A first;
   B second;
   // Конструктор класу:
   MyClass(A f,B s){
      first=f;
      second=s;
```

```
    }
    // Метод для відображення значення полів:
    void show(){
        cout<<"Перше поле: "<<first<<endl;
        cout<<"Друге поле: "<<second<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єктів на основі узагальненого класу:
    MyClass<int,char> objA(100,'A');
    MyClass<string,double> objB("текст",10.5);
    MyClass<char*,string> objC("перший","другий");
    MyClass<int,int> objD(1,2);
    // Перевірка значень полів об'єктів:
    objA.show();
    objB.show();
    objC.show();
    objD.show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 7.6)

```
Перше поле: 100
Друге поле: A
Перше поле: текст
Друге поле: 10.5
```

```
Перше поле: перший
Друге поле: другий
Перше поле: 1
Друге поле: 2
```

Опис узагальненого класу, як і слід було очікувати, починається з ключового слова `template`. Інструкція `<class A,class B>` означає, що в класі використовуються два узагальнених параметри `A` і `B` для ідентифікації типів. Далі йде безпосередньо опис класу `MyClass`. Зокрема, в класі оголошено поле `first` типу `A`, і поле `second` типу `B`. Який саме тип приховується за параметрами `A` і `B`, стане зрозуміло після створення об'єкта класу `MyClass`.

У класі описано конструктор з двома аргументами (типу `A` перший і типу `B` другий), які задають значення полів `first` і `second`. Також у класі є метод `show()`, призначений для відображення значень полів.

У головній функції програми створюється ряд об'єктів. Зокрема, для цього використовуються команди `MyClass<int,char> objA(100,'A')`, `MyClass<string,double> objB("текст",10.5)`, `MyClass<char*,string> objC("перший","другий")` і `MyClass<int,int> objD(1,2)`. У кожній команді після імені класу `MyClass` в кутових дужках зазначено ідентифікатори типів, які слід використовувати замість узагальнених параметрів під час створення об'єкта класу. Скажімо, інструкція `<int,char>` у команді створення об'єкта `objA` означає, що як параметр `A` слід використовувати `int`, а як параметр `B` слід використовувати `char`.

У команді `MyClass<string,double> objB("текст",10.5)` для першого параметра вказано значення `string` і, відповідно, як перший аргумент конструктору передається текстовий літерал `"текст"`. Як неодноразово зазначалося, текстові літерали реалізуються у вигляді символьних масивів, тому насправді йдеться про формальний тип `char*`. Проте тут має

місце автоматичне зведення типів, і літерал без нашої участі зводиться до типу `string`. Разом із тим, нарівні з типом `string` допускається використовувати і формальний тип `char*` для позначення "текстового" типу. Прикладом є команда `MyClass<char*,string> objC("перший","другий")`. Нарешті, обидва узагальнених параметри можуть набувати однакових значень - як, наприклад, під час створення об'єкта командою `MyClass<int,int> objD(1,2)`. Тут обидва параметри `A` і `B` набувають значення `int`.

До уваги

Для використання в програмі класу `string` підключається заголовок `<string>`.

Для перевірки значення полів створених об'єктів із кожного об'єкта викликається метод `show()`.

Явна спеціалізація узагальненого класу

Як і у випадку з узагальненими функціями, під час опису узагальненого класу можна задавати його *явну спеціалізацію* для окремих випадків. Розглянемо приклад, поданий у лістингу 7.7.

Лістинг 7.7. Явна спеціалізація узагальненого класу

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Узагальнений клас:
template<class T> class MyClass{
private:
   // Закрите поле:
   T value;
```

```
public:
    // Конструктор:
    MyClass(T v){
        value=v;
    }
    // Метод для відображення значення поля:
    void show(){
        cout<<"Значення поля: "<<value<<endl;
    }
};
// Явна спеціалізація класу:
template<> class MyClass<string>{
private:
    // Закрите поле - масив:
    char value[100];
public:
    // Конструктор:
    MyClass(char* str){
        int k;
        // Перший символ в полі - масиві:
        value[0]='|';
        // Заповнення поля - масиву:
        for(k=0;str[k];k++){
            value[2*k+1]=str[k];
            value[2*k+2]='|';
        }
        // Додавання нуль-символу в кінець масиву:
        value[2*k+1]='\0';
    }
    void show(){
        cout<<value<<endl;
    }
```

```
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    MyClass<int> objA(100);
    // Створення об'єктів на основі узагальненого класу:
    MyClass<char> objB('A');
    MyClass<char*> objC("текст");
    MyClass<string> objD("текст");
    // Виклик методу з об'єкта, створеного на
    // основі узагальненого класу:
    objA.show();
    objB.show();
    objC.show();
    objD.show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми буде таким, як показано нижче:

Результат виконання програми (із лістингу 7.7)

```
Значення поля: 100
Значення поля: A
Значення поля: текст
|т|е|к|с|т|
```

У програмі описується узагальнений клас `MyClass`, у якому через `T` позначено узагальнений тип. У класу є закрите поле `value`, що належить до узагальненого типу `T`. Конструктор класу отримує один аргумент, який і задає

значення поля value. Для отримання інформації про значення поля value призначено метод show().

Явну спеціалізацію класу MyClass описано для випадку, коли як узагальнений параметр використовується тип string.

До уваги

Оскільки в програмі використовується клас string, підключаємо заголовок <string>. Щодо явної спеціалізації для узагальненого класу, то після ключового слова template ставляться пусті кутові дужки <>, а після імені класу в кутових дужках зазначається ідентифікатор типу (в цьому випадку string), для якого виконується спеціалізація класу.

Спеціальна версія класу MyClass містить, як і "загальна" версія класу, закрите поле з назвою value, яке тепер є символьним масивом (зі 100 елементів). Заповнення цього масиву виконується в конструкторі. Як аргумент конструктору (позначено як str) формально передається вказівник на значення типу char, що насправді передбачає передачу конструктору текстового літерала як аргументу. Принцип заповнення масиву value такий: в нього копіюються символи з текстового літерала, переданого аргументом конструктору, але тільки між символами додається вертикальна риска. Така ж риска додається на початок масиву value. Оскільки вертикальна риска - це окремий символ, то заповнення масиву value виглядає не зовсім тривіально. Зокрема, на самому початку командою value[0]='|' першим символом у масив value записується вертикальна риска. Потім запускається оператор циклу, в якому індексна змінна k пробігає значення від 0 і до кінця текстового літерала.

Подробиці

Умова в умовному операторі виглядає як str[k]. Ця умова буде інтерпретуватися як false, тільки якщо елемент з індексом k є нуль-символом закінчення рядка.

Індексну змінну k оголошено до запуску оператора циклу, оскільки значення цієї змінної, яке вона буде мати після закінчення оператора циклу, ми використовуємо (під час запису нуль-символу в масив value).

При фіксованому значенні індексної змінної командою value[2*k+1]=str[k] у масив value записується чергова буква з аргументу str конструктора, а після неї додається вертикальна риска (команда value[2*k+2]='|').

Подробиці



Індекс k визначає букву в текстовому літералі str (який насправді є символьним масивом). Масив value заповнюється по два елементи. На позиціях із парними індексами в масиві value розташовуються символи вертикальної риски. На непарних позиціях у масиві value містяться букви з масиву str. Формула для непарних індексів у масиві value задається виразом 2*k+1, а значення для парних індексів обчислюються як 2*k+2. Тут ми врахували, що при нульовому значенні індексу k (перша буква в масиві str) у масиві value слід заповнювати елемент з індексом 1, а символ вертикальної риски присвоюється елементу з індексом 2 в масиві value.

Після закінчення оператора циклу масив value заповнено буквами і вертикальними рисками, але треба ще додати нуль-символ. Оскільки в операторі циклу змінна k уже отримала чергове значення, то додавання нуль-символу виконуємо за допомогою команди value[2*k+1]='\0'.

Подробиці



Після закінчення оператора циклу індексна змінна k отримує значення елемента в масиві str, який відповідає нуль-символу. В масиві value цій позиції відповідає елемент з індексом 2*k+1.

Метод show() у цій версії класу MyClass визначено так, що відображується вміст масиву value (без додаткового пояснювального тексту).

До уваги

Таким чином, у явній спеціалізації класу `MyClass` для типу `string` цей самий тип `string` фактично не використовується.

У головній функції програми на основі узагальненого класу `MyClass` створюється декілька об'єктів. Особливої уваги варті об'єкти, які створюються з ідентифікаторами типів `char*` і `string` для узагальнених параметрів. Як аргумент конструктору в обох випадках передається один і той самий текст, але результат у цих випадках отримуємо різний, оскільки в одному випадку (формальний тип `char*`) використовується "загальна" версія класу `MyClass`, а у другому випадку (тип `string`) використовується явна специфікація цього класу.

До уваги

Коли використовують ідентифікатор типу `char*`, то в конструкторі полю `value` значенням присвоюється адреса того текстового літерала, який передано як аргумент конструктору. Якщо передати аргументом конструктору ім'я символьного масиву, в поле `value` запишеться адреса цього масиву. Якщо створити декілька об'єктів на основі класу `MyClass` з ідентифікатором типу `char*` і кожний раз передавати аргументом конструктору один і той самий символьний масив (його ім'я), то поля `value` всіх створених об'єктів будуть посилатися на один і той самий масив.

Значення параметрів за замовчуванням

Під час опису узагальненого класу для параметрів типу можна задати значення за замовчуванням. Значення за замовчуванням для узагальнених параметрів є ідентифікаторами типів і зазначаються через знак рівності після назви параметра. У лістингу 7.8 з цього приводу наведено програму.

Лістинг 7.8. Значення параметрів за замовчуванням

```
#include <iostream>
```

```
#include <cstdlib>
#include <string>
using namespace std;
// Узагальнений клас:
template<class A=int,class B=char> class MyClass{
public:
    // Поля:
    A first;
    B second;
    // Конструктор:
    MyClass(A f,B s){
        first=f;
        second=s;
    }
    // Метод для відображення значення полів:
    void show(){
        cout<<"Значення "<<first<<" і "<<second<<endl;
    }
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єктів:
    MyClass<double,int> objA(3.5,100);
    MyClass<string> objB("текст",'A');
    MyClass<> objC(200,'B');
    // Виклик методу:
    objA.show();
    objB.show();
    objC.show();
    // Затримка консольного вікна:
```

```
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 7.8)

```
Значення 3.5 і 100
Значення текст і A
Значення 200 і B
```

В основному код читачу повинен бути зрозумілим. Звернемо лише увагу на принципові й нові моменти. Насамперед, вони стосуються опису шаблонного класу `MyClass`: у кутових дужках після назв узагальнених параметрів подано значення за замовчуванням. Зокрема, інструкція `<class A=int,class B=char>` означає, що якщо під час створення об'єкта на основі узагальненого класу явно не зазначити ідентифікатори типів для узагальнених параметрів, то для першого параметра буде використано ідентифікатор `int`, а для другого параметра - ідентифікатор `char`. Головна функція програми містить приклади створення об'єктів на основі узагальненого класу. Так, командою `MyClass<double,int> objA(3.5,100)` створюється об'єкт із явним зазначенням використовуваних ідентифікаторів для узагальнених параметрів (`double` і `int`). У команді `MyClass<string> objB("текст",'A')` явно зазначений тільки один ідентифікатор типу `string`, тому для другого узагальненого параметра буде використано значення за замовчуванням `char`. Нарешті, в команді `MyClass<> objC(200,'B')` явно не зазначено жодного ідентифікатора, тому для узагальнених типів використовуються значення за замовчуванням (`int` і `char`).

До уваги

Навіть якщо ідентифікатори типів у команді створення об'єкта явно не вказують, кутові дужки (пусті) все одно присутні.

Успадкування узагальнених класів

У наступному прикладі показано, як на основі одного узагальненого класу шляхом успадкування можна створити інший узагальнений клас. Розглянемо програмний код у лістингу 7.9.

Лістинг 7.9. Успадкування узагальненого класу

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Перший базовий узагальнений клас:
template<class A> class Alpha{
public:
   // Поле:
   A alpha;
   // Конструктор:
   Alpha(A a){
      alpha=a;
   }
   // Метод:
   void show(){
      cout<<"Поле alpha: "<<alpha<<endl;
   }
};
// Другий базовий узагальнений клас:
template<class B> class Bravo{
public:
   // Поле:
   B bravo;
   // Конструктор:
   Bravo(B b){
```

```
        bravo=b;
    }
    // Метод:
    void show(){
        cout<<"Поле bravo: "<<bravo<<endl;
    }
};
// Похідний узагальнений клас:
template<class A,class B,class C> class Charlie: public
Alpha<A>,public Bravo<B>{
public:
    // Поле:
    C charlie;
    // Конструктор:
    Charlie(A a,B b,C c):Alpha<A>(a),Bravo<B>(b){
        charlie=c;
    }
    // Перевизначення методу:
    void show(){
        // Виклик версії методу з першого базового класу:
        Alpha<A>::show();
        // Виклик методу з другого базового класу:
        Bravo<B>::show();
        cout<<"Поле charlie: "<<charlie<<endl;
    }
};
// Головна функція програма:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення об'єктів:
    Charlie<string,char,int> objA("текст",'A',100);
```

```
    Charlie<int,double,char> objB(200,5.5,'B');
    // Виклик методу з об'єктів:
    objA.show();
    objB.show();
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 7.9)

```
Поле alpha: текст
Поле bravo: A
Поле charlie: 100
Поле alpha: 200
Поле bravo: 5.5
Поле charlie: B
```

Програма досить проста. В ній описано два однотипних узагальнених класи `Alpha` і `Bravo`, а потім на основі цих класів шляхом успадкування створюється узагальнений клас `Charlie`. Загалом, схема успадкування узагальнених класів нагадує успадкування звичайних класів. Поправку треба робити тільки на те, що у випадку з узагальненими класами важлива не лише назва класу, а й узагальнені параметри, які використовуються. Скажімо, в описі класу `Charlie` після ключового слова `template` зазначено вираз `<class A,class B,class C>`, що означає, що в класі `Charlie` використовуються три узагальнених параметри (`A`, `B` і `C`). Далі в описі як ті, що успадковуються, зазначено класи `Alpha<A>` і `Bravo<B>`. Тут ми не обмежуємося тільки назвами базових класів, але ще й вказуємо, які узагальнені параметри слід використовувати.

До уваги

Усе це означає, що коли створюється об'єкт класу `Charlie`, і задано значення для узагальнених параметрів, то перший параметр передається до узагальненого класу `Alpha`, а другий - до узагальненого класу `Bravo`.

Конструктор класу `Charlie` описано з трьома аргументами. Перший аргумент передається як аргумент під час виклику конструктора базового класу `Alpha<A>`, а другий аргумент передається конструктору базового класу `Bravo<B>`. Тут (маються на увазі інструкції `Alpha<A>` і `Bravo<B>`) ми явно підкреслюємо, які значення для узагальнених параметрів необхідно використовувати, коли викликається конструктор базового класу. Аналогічна ситуація має місце під час перевизначення в похідному класі `Charlie` методу `show()`. У тілі цього методу командою `Alpha<A>::show()` викликається версія методу з першого базового класу, а версія методу з другого базового класу викликається командою `Bravo<B>::show()`.

У головній функції програми показано як на основі узагальненого похідного класу `Charlie` створюються об'єкти.

У розглянутому прикладі на основі двох узагальнених класів створювався похідний узагальнений клас. Можна вчинити й інакше: узяти узагальнений базовий клас, і на його основі успадкуванням створити звичайний клас (або класи). Програму, в якій реалізується такий підхід, наведено у лістингу 7.10.

Лістинг 7.10. Створення звичайного класу на основі узагальненого

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Базовий узагальнений клас:
template<class X> class BaseClass{
private:
   // Закрите поле:
```

```
    X value;
public:
    // Конструктор:
    BaseClass(X val){
        set(val);
    }
    // Метод для присвоювання значення полю:
    void set(X val){
        value=val;
    }
    // Метод для отримання значення поля:
    X get(){
        return value;
    }
};
// Перший похідний клас:
class Alpha: public BaseClass<int>{
public:
    // Конструктор:
    Alpha():BaseClass<int>(0){}
};
// Другий похідний клас:
class Bravo: public BaseClass<char>{
public:
    // Конструктор:
    Bravo(char s):BaseClass<char>(s){}
};
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Об'єкт першого похідного класу:
```

```
    Alpha objA;
    // Виконання операцій із об'єктом:
    cout<<"Об'єкт objA: "<<objA.get()<<endl;
    objA.set(100);
    cout<<"Об'єкт objA: "<<objA.get()<<endl;
    // Об'єкт другого похідного класу:
    Bravo objB('A');
    // Виконання операцій із об'єктом:
    cout<<"Об'єкт objB: "<<objB.get()<<endl;
    objB.set('B');
    cout<<"Об'єкт objB: "<<objB.get()<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Як виглядає результат виконання програми, показано нижче:

Результат виконання програми (із лістингу 7.10)

```
Об'єкт objA: 0
Об'єкт objA: 100
Об'єкт objB: A
Об'єкт objB: B
```

У програмі описано узагальнений клас із назвою `BaseClass`. Для класу оголошено один узагальнений параметр, позначений як `X`. У самому класі описано закрите поле `value`, що належить до узагальненого типу `X`. Відкритий метод `set()` призначений для присвоювання значення полю `value`, а відкритий метод `get()` як результат повертає значення цього поля. Також в класі описано конструктор із одним аргументом. Значення, передане як аргумент конструктору, через виклик методу `set()` присвоюється полю `value`.

На основі класу `BaseClass` за допомогою спадкування створюються класи `Alpha` і `Bravo`. Під час створення класу `Alpha` у базовий узагальнений клас `BaseClass` як значення узагальненого параметра передається `int`. Під час створення класу `Bravo` клас `BaseClass` використовується з параметром `char`. У класах `Alpha` і `Bravo` описано конструктори. Для класу `Alpha` виконання конструктора зводиться до виклику конструктора класу `BaseClass<int>` з нульовим аргументом. Під час виконання конструктора класу `Bravo` викликається конструктор класу `BaseClass<char>` із одним символьним аргументом.

У функції `main()` показано, як створюються та використовуються об'єкти класів `Alpha` і `Bravo`.

Цілочислові узагальнені параметри

Параметри в узагальненому класі є позначенням для ідентифікаторів типу. Проте є в цьому правилі виняток - в узагальненому класі можуть також використовуватися цілочислові параметри. Описують такі параметри подібно до стандартних узагальнених параметрів і відіграють вони роль, подібну до аргументу, який передається в клас. Як приклад такої ситуації розглянемо програму у лістингу 7.11. У програмі описано узагальнений клас `Polynom`, призначений для реалізації поліномів і виконання деяких базових операцій, таких, наприклад, як множення поліномів, додавання і віднімання поліномів, а також обчислення похідної для полінома.

Подробиці



Для розуміння принципів організації програми є сенс нагадати особливості таких математичних об'єктів, як поліноми.

Поліномом степеня n називається функціональна залежність виду $P_n(x) = a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$. Поліном як функція однозначно визначається набором коефіцієнтів a_0, a_1, , a_n. Кількість коефіцієнтів полінома на одиницю більша за степінь полінома. Щоб дізнатися значення полінома в точці x, слід обчислити суму $a_0 + a_1x + a_2x^2 + \cdots + a_nx^n$.

Похідна $P_n'(x)$ для полінома $P_n(x)$ є поліномом степеня $n-1$ і визначається виразом $P_n'(x) = a_1 + 2a_2x + 3a_3x^2 + \cdots + na_nx^{n-1}$.

Якщо поліном помножити на деяке число k, то в результаті отримаємо також поліном. Порівняно з вихідним (який помножують) поліномом, коефіцієнти полінома-результату множаться на відповідне число: $kP_n(x) = ka_0x + ka_1x + ka_2x^2 + \cdots + ka_nx^n$.

Під час обчислення суми і різниці поліномів додають (віднімають) значення коефіцієнтів поліномів, що відповідають однаковим індексам (однаковим показникам степеня аргументу x).

Основу програми складає узагальнений клас `Polynom`. У ньому як узагальнений параметр виступає степінь полінома, що реалізується через об'єкт класу. У класі є звичайні й узагальнені операторні методи. Також у програмі описано узагальнені операторні функції. Тепер розглянемо наведений далі програмний код:

Лістинг 7.11. Узагальнений клас для реалізації поліномів

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Узагальнений клас для реалізації поліномів:
template<int power> class Polynom{
private:
   // Закрите поле - масив із коефіцієнтами.
   // Розмір масиву на одиницю більший за степінь полінома:
   double a[power+1];
public:
   // Конструктор без аргументів:
   Polynom(){
      // Заповнення масиву нулями:
      for(int k=0;k<=power;k++){
         a[k]=0;
      }
```

```
}
// Конструктор із аргументом - іменем числового масиву:
Polynom(double* nums){
   // Копіювання елементів із одного масиву в інший:
   for(int k=0;k<=power;k++){
      a[k]=nums[k];
   }
}
// Метод для відображення значень
// елементів поля - масиву:
void getAll(){
   cout<<"| ";
   for(int k=0;k<=power;k++){
      cout<<a[k]<<" | ";
   }
   cout<<endl;
}
// Операторний метод дозволяє викликати об'єкт.
// Як результат повертається значення полінома:
double operator()(double x){
   double s=0,q=1;
   for(int k=0;k<=power;k++){
      s+=a[k]*q;
      q*=x;
   }
   return s;
}
// Операторний метод для індексування об'єктів:
double &operator[](int k){
   return a[k];
}
// Узагальнений операторний метод для обчислення
```

```
    // добутку поліномів:
    template<int  n>  Polynom<power+n>  operator*(Polynom<n>
pol){
        // Локальний об'єкт узагальненого класу:
        Polynom<power+n> tmp;
        // Обчислення коефіцієнтів для
        // полінома - результату:
        for(int i=0;i<=power;i++){
           for(int j=0;j<=n;j++){
              tmp[i+j]+=a[i]*pol[j];
           }
        }
        // Результат методу:
        return tmp;
    }
    // Узагальнений операторний метод для обчислення
    // суми поліномів:
     template<int          n>          Polynom<(n>power?n:power)>
operator+(Polynom<n> pol){
        int i;
        // Локальний об'єкт узагальненого класу:
        Polynom<(n>power?n:power)> tmp;
        // Обчислення коефіцієнтів для
        // полінома - результату:
        for(i=0;i<=power;i++){
           tmp[i]+=a[i];
        }
        for(i=0;i<=n;i++){
           tmp[i]+=pol[i];
        }
        // Результат методу:
        return tmp;
```

```
    }
  };
  // Узагальнена операторна функція для обчислення
  // результату множення полінома на число:
  template<int power> Polynom<power> operator*(Polynom<power>
pol,double r){
    // Локальний об'єкт узагальненого класу:
    Polynom<power> tmp;
    // Обчислення коефіцієнтів для полінома - результату:
    for(int k=0;k<=power;k++){
      tmp[k]=pol[k]*r;
    }
    // Результат функції:
    return tmp;
  }
  // Узагальнена операторна функція для обчислення
  // результату множення числа на поліном:
  template<int power> Polynom<power> operator*(double
r,Polynom<power> pol){
    // Поліном множиться на число:
    return pol*r;
  }
  // Узагальнена операторна функція для обчислення
  // різниці двох поліномів:
  template<int m,int n> Polynom<(m>n?m:n)> operator-
(Polynom<m> x,Polynom<n> y){
    // До першого полінома додається другий, помножений
    // на число -1:
    return x+(-1)*y;
  }
  // Узагальнена функція для обчислення
  // похідної від полінома:
```

```
template<int power> Polynom<power-1> Diff(Polynom<power>
pol){
    // Локальний об'єкт узагальненого класу:
    Polynom<power-1> tmp;
    // Обчислення коефіцієнтів для полінома - результату:
    for(int k=0;k<=power-1;k++){
       tmp[k]=pol[k+1]*(k+1);
    }
    // Результат функції:
    return tmp;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Масиви з коефіцієнтами:
    double A[]={1,2,-1,1};
    double B[]={-1,3,0,2,-1,1};
    // Аргумент для поліномів:
    double x=2;
    // Перший поліном:
    Polynom<3> P(A);
    cout<<"Поліном P:\t";
    // Коефіцієнти першого полінома:
    P.getAll();
    cout<<"Значення P("<<x<<") = ";
    // Значення першого полінома в точці:
    cout<<P(x)<<endl;
    cout<<"Поліном P':\t";
    // Коефіцієнти для похідної від полінома:
    Diff(P).getAll();
    cout<<"Значення P'("<<x<<") = ";
```

```
    // Значення похідної в точці:
    cout<<Diff(P)(x)<<endl;
    // Другий поліном:
    Polynom<5> Q(B);
    cout<<"Поліном Q:\t";
    // Коефіцієнти другого полінома:
    Q.getAll();
    cout<<"Значення Q("<<x<<") = ";
    // Значення другого полінома в точці:
    cout<<Q(x)<<endl;
    cout<<"Поліном P*Q:\t";
    // Коефіцієнти для добутку поліномів:
    (P*Q).getAll();
    cout<<"Значення (P*Q)("<<x<<") = ";
    // Значення добутку поліномів в точці:
    cout<<(P*Q)(x)<<endl;
    cout<<"Поліном P+Q:\t";
    // Коефіцієнти для суми поліномів:
    (P+Q).getAll();
    cout<<"Значення (P+Q)("<<x<<") = ";
    // Значення суми поліномів в точці:
    cout<<(P+Q)(x)<<endl;
    cout<<"Поліном Q-P:\t";
    // Коефіцієнти для різниці поліномів:
    (Q-P).getAll();
    cout<<"Значення (Q-P)("<<x<<") = ";
    // Значення різниці поліномів в точці:
    cout<<(Q-P)(x)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У результаті виконання програми отримуємо таке:

Результат виконання програми (із лістингу 7.11)

```
Поліном P:       | 1 | 2 | -1 | 1 |
Значення P(2) = 9
Поліном P':      | 2 | -2 | 3 |
Значення P'(2) = 10
Поліном Q:       | -1 | 3 | 0 | 2 | -1 | 1 |
Значення Q(2) = 37
Поліном P*Q:     | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
Значення (P*Q)(2) = 333
Поліном P+Q:     | 0 | 5 | -1 | 3 | -1 | 1 |
Значення (P+Q)(2) = 46
Поліном Q-P:     | -2 | 1 | 1 | 1 | -1 | 1 |
Значення (Q-P)(2) = 28
```

В узагальненому класі `Polynom` через `power` позначено цілочисловий параметр. Цей параметр, як зазначалося вище, під час створення об'єкта визначає степінь полінома, який ототожнюється з об'єктом класу. Кількість коефіцієнтів у поліномі тоді дорівнює `power+1`. Для запам'ятовування коефіцієнтів полінома в класі `Polynom` оголошується закрите числове поле `a`, що є масивом розміру `power+1`. Індекси елементів масиву `a`, таким чином, змінюються в межах від `0` до `power` включно.

У класі описано два конструктори. Конструктор без аргументів всі елементи масиву заповнює нулями. Є ще одна версія конструктора, що дозволяє передавати як аргумент ім'я масиву з коефіцієнтами полінома. У такому випадку значення елементів із масиву, переданого аргументом конструктору, копіюються в масив `a`. Зрозуміло, що розміри обох масивів повинні збігатися.

З метою зручності та наочності в класі описано метод `getAll()`, який у консольному вікні відображає значення коефіцієнтів полінома.

Щоб об'єкт класу `Polynom` можна було викликати як функцію, описується операторний метод `operator()()`. У тілі методу за допомогою оператора циклу обчислюється сума, що дає значення для полінома в точці, що визначається як аргумент операторного методу.

Операторний метод `operator[]()` дозволяє індексувати об'єкти класу `Polynom`. Як аргумент методу передається цілочисловий індекс, а як результат метод повертає посилання (інструкція `&` в описі методу) на елемент в масиві `a` з індексом, що визначається аргументом операторного методу. На цьому простий код, загалом, закінчується.

Для обчислення добутку поліномів описано узагальнений операторний метод `operator*()`. Аргумент методу позначено як `pol` і його описано з типом `Polynom<n>` - тобто він є об'єктом узагальненого класу. Через `n` тут позначено цілочисловий параметр узагальненого методу. Тип результат методу визначається ідентифікатором `Polynom<power+n>`, тобто це теж об'єкт узагальненого класу. Вираз `power+n` у кутових дужках дорівнює сумі показників степенів поліномів, що перемножуються.

До уваги

В операторному методі аргумент методу ототожнюється з другим операндом. Першим операндом є об'єкт, із якого викликається метод. Об'єкту, з якого викликається метод, відповідає поліном степеня `power`. Степінь другого полінома дорівнює `n`. Тому степінь полінома, що дорівнює добутку поліномів зі степенями `power` і `n`, дорівнює `power+n`.

У тілі методу командою `Polynom<power+n> tmp` створюється локальний об'єкт `tmp` узагальненого класу. Цей об'єкт буде повертатися як результат методу. Спочатку всі коефіцієнти об'єкта `tmp` нульові (завдяки конструкторові класу без аргументів). Їх потрібно обчислити. З цією метою запускаються вкладені оператори циклу, в яких індексні змінні `i` і `j` "пробігають" значення індексів коефіцієнтів у перемножуваних об'єктах (поліномах). При фіксованих індексах `i` і `j` виконується команда `tmp[i+j]+=a[i]*pol[j]`. Тут, ми, по-

перше, скористалися можливістю індексувати об'єкт `pol` (відверто кажучи, іншого способу отримати доступ до закритого масиву цього об'єкта просто нема). По-друге, взяли до уваги, що під час перемножування доданків зі степенями аргументу `i` і `j` виходить доданок із аргументом у степені `i+j`.

Подробиці



Варто врахувати, що якщо $P_n(x) = \sum_{i=0}^{n} a_i x^i$ і $Q_m(x) = \sum_{j=0}^{m} b_j x^j$, то $P_n(x)Q_m(x) = \sum_{i=0}^{n}\sum_{j=0}^{m} a_i b_j x^{i+j}$. Таким чином, у поліномі-результаті коефіцієнт при доданку з x^k дорівнює сумі всіх можливих добутків виду $a_i b_j$, для яких $i + j = k$.

Після закінчення обчислень об'єкт `tmp` повертається як результат методу.

Схожим чином описується узагальнений операторний метод для обчислення суми поліномів. Тип результату визначений виразом `Polynom<(n>power?n:power)>`. Через `n` позначено цілочисловий параметр узагальненого методу. Як результат виразу `n>power?n:power` повертається більше зі значень параметрів `n` і `power`.

Подробиці



Тернарний оператор `?:` перевіряє умову, вказану першим операндом (для виразу `n>power?n:power` це умова `n>power`). Якщо умова істинна, повертається значення після знака питання (значення `n`). Якщо умова хибна, то повертається значення після двокрапки (значення `power`).

Результатом суми двох поліномів є поліном. Степені поліномів, які додаються, загалом різні. Якщо в одного з них степінь більший, то такий самий степінь буде й у полінома-результату. Тому в операторному методі необхідно визначати, у якого з поліномів степінь більший. Тому й порівнюються значення параметрів `power` і `n`. Більше з них визначає степінь полінома-результату.

У тілі методу локальний об'єкт із нульовими коефіцієнтами створюється командою `Polynom<(n>power?n:power)> tmp`. До нульових значень коефіцієнтів об'єкта `tmp` необхідно послідовно додати значення відповідних коефіцієнтів із об'єкта `pol` (аргумент методу) і масиву `a`. Для цього послідовно

запускається два оператори циклу. В підсумку "заповнений" об'єкт `tmp` повертається як результат методу.

Крім класу `Polynom`, у програмі описано декілька узагальнених функцій. Так, узагальнена операторна функція `operator*()` для обчислення добутку полінома на число повертає результатом об'єкт класу `Polynom<power>` (через `power` позначено параметр узагальненої функції). Аргументів у операторної функції два. Перший аргумент `pol` є об'єктом класу `Polynom<power>`. Другий аргумент `r` є числом типу `double`. У тілі функції командою `Polynom<power> tmp` створюється локальний об'єкт, і за допомогою оператора циклу кожному коефіцієнту (при фіксованому індексі `k`) командою `tmp[k]=pol[k]*r` (добуток коефіцієнта вихідного полінома на число) присвоюється значення. Потім об'єкт `tmp` повертається як результат функції. Варто зазначити, що ця функція оброблює ситуацію, коли об'єкт (поліном) множиться на число. Щоб можна було множити число на об'єкт, необхідно описати ще одну версію операторної функції. Таку функцію описано. Від першої версії вона відрізняється порядком розташування аргументів (спочатку зазначено числовий аргумент `r`, а потім об'єкт `pol`). Щоб не дублювати програмний код у тілі функції, ми звернулися за допомогою до інструкції `return pol*r`. Простіше кажучи, для обчислення результату функції, що оброблює операцію множення числа на об'єкт, ми викликаємо функцію, що оброблює множення об'єкта на число.

До схожої хитрості ми звернулися під час опису узагальненої операторної функції `operator-()`, яка обчислює різницю двох об'єктів (поліномів).

Подробиці



Ми виходимо з того, що віднімання від полінома $Q_m(x)$ полінома $P_n(x)$ еквівалентне додаванню до полінома $Q_m(x)$ полінома $P_n(x)$, помноженого на -1. Іншими словами, ми скористалися тотожністю $Q_m(x) - P_n(x) = Q_m(x) + (-1)P_n(x)$.

У функції два цілочислових узагальнених параметри `m` і `n` (степені поліномів, які віднімаються). Як результат повертається об'єкт класу `Polynom<(m>n?m:n)>` (степінь полінома-результату визначається більшим із двох значень параметрів `m` і `n`). Аргументи функції `x` і `y` є об'єктами відповідно класів `Polynom<m>` і `Polynom<n>`. Як результат функції повертається значення виразу `x+(-1)*y`. Воно має сенс, оскільки визначені операції множення об'єктів на число і додавання об'єктів.

Узагальнена функція `Diff()` з узагальненим цілочисловим параметром `power` (степінь диференційованого полінома) для обчислення похідної від полінома повертає результатом об'єкт класу `Polynom<power-1>` (степінь полінома-похідної на одиницю менший за степінь диференційованого полінома). Аргумент функції `pol` (диференційований поліном) є об'єктом класу `Polynom<power>`. У тілі функції створюється локальний об'єкт `tmp` класу `Polynom<power-1>`, і в операторі циклу заповнюються значення коефіцієнтів для цього об'єкта. Об'єкт `tmp` повертається як результат функції.

У головній функції програми оголошуються й ініціалізуються два числових масиви `A` і `B`, на основі яких командами `Polynom<3> P(A)` і `Polynom<5> Q(B)` створюються об'єкти, які ми ототожнюємо з поліномами. Далі з об'єктами виконуються певні операції, такі як обчислення похідної, множення, додавання і віднімання.

До уваги

Інструкція `\t` є символом табуляції.

Хто бажає, може самостійно зайнятись перевіркою результатів цих обчислень.

До уваги

У розглянутому прикладі об'єкти для поліномів створюються на основі узагальненого класу. Однак, кожного разу узагальненому класу передається цілочисловий параметр. Формально об'єкти, створені на основі одного й того ж узагальненого

класу, але з різними значеннями цілочислового параметра, мають різні тип. Це накладає певні обмеження. Наприклад, під час обчислення добутку, суми, різниці поліномів або похідної від полінома характеристики класу (мається на увазі значення числового параметра) обчислюються автоматично. Все це виконується в операторних методах або функціях. Проте, щоб присвоїти результат виклику такої функції або методу об'єктній змінній, ця об'єктна змінна повинна бути попередньо оголошена на основі узагальненого класу з таким самий цілочисловим параметром, як для об'єкта - результату функції або методу. Тому, наприклад, ми послуговувалися командами `(P*Q).getAll()`, `(P+Q).getAll()`, `(Q-P).getAll()`, `(P*Q)(x)`, `(P+Q)(x)` або `(Q-P)(x)`, а не записували результат добутку об'єктів в окрему змінну.

Рекомендації для самостійної роботи

• Подумайте, які універсальні алгоритми можуть бути реалізовані за допомогою узагальнених функцій? Запропонуйте програмні коди таких функцій.

• Спробуйте створити узагальнений контейнерний клас, призначений для "зберігання" упорядкованого списку елементів, до якого, крім іншого, можна було б уже після створення об'єкта додавати й видаляти елементи. Іншими словами, пропонується створити аналог масиву, до якого можна додавати нові елементи (змінюючи тим самим розмір "масиву") і видаляти елементи з "масиву".

• Спробуйте створити узагальнений клас, призначений для створення "ланцюжка" об'єктів або "бінарного дерева". Такий клас повинен містити поле (або поля), що є вказівником на об'єкт того ж узагальненого класу.

Глава 8

Різні задачі

- Андрію! Може покаємось?

- Та йди ти до всіх чортів! Он чоловік у карти програє!

з х/ф "Пропала грамота"

У цій главі розглядаються різноманітні задачі. Різноманітні в плані тематики. Зокрема, ми дізнаємося:

- що таке структури, і як вони використовуються;
- як у програмі реалізуються комплексні числа;
- які узагальнені контейнерні класи можуть використовуватися для створення масивів, множин й асоціативних контейнерів;
- як виконується обробка виняткових ситуацій;
- що таке багатопоточне програмування,

ну і ще дещо.

Знайомство зі структурами

У наведеній далі програмі задача про обчислення кінцевої суми депозиту розв'язується з використанням *структур*.

Теорія

Структура є набором змінних (різного типу в загальному випадку), об'єднаних спільною назвою. Опис структури починається з ключового слова `struct`, після якого зазначають назву структури й у фігурних дужках - типи й назви змінних, які входять до структури (після фігурних дужок ставиться крапка з комою). Змінні, що входять у структуру, будемо називати полями структури.

У певному сенсі структура нагадує клас, але тільки без методів. Структура, подібно до класу, визначає своєрідний тип даних. Аналогом об'єкта класу виступає екземпляр структури.

Для створення екземпляра структури оголошується змінна, типом якої вказується

ім'я структури. Якщо необхідно ініціалізувати екземпляр структури, то відповідній змінній у команді оголошення екземпляра структури присвоюється список із значеннями полів.

Звертання до полів екземпляра структури виконується в "крапковій" нотації: після імені екземпляра структури через крапку вказується назва поля.

Розглянемо програмний код, наведений у лістингу 8.1.

Лістинг 8.1. Знайомство зі структурами

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Опис структури:
struct MyMoney{
    // Поля структури:
    string name;
    double money;
    double rate;
    int time;
};
// Функція для обчислення (на основі екземпляра
// структури) кінцевої суми:
double getMoney(MyMoney str){
    double s=str.money;
    for(int k=1;k<=str.time;k++){
       s*=(1+str.rate/100);
    }
    return s;
}
// Функція для відображення повної інформації
// для екземпляра структури:
void show(MyMoney str){
```

```
    cout<<"Ім'я вкладника: "<<str.name<<endl;
    cout<<"Початкова сума: "<<str.money<<endl;
    cout<<"Відсоткова ставка: "<<str.rate<<endl;
    cout<<"Час (років): "<<str.time<<endl;
    cout<<"Кінцева сума: "<<getMoney(str)<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Перший екземпляр структури:
    MyMoney cat={"Пан Коцький",1000,8,5};
    // Другий екземпляр структури:
    MyMoney fox={"Лис Микита",1200,7,4};
    // Відображення інформації:
    show(cat);
    show(fox);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 8.1)

```
Ім'я вкладника: Пан Коцький
Початкова сума: 1000
Відсоткова ставка: 8
Час (років): 5
Кінцева сума: 1469.33
Ім'я вкладника: Лис Микита
Початкова сума: 1200
Відсоткова ставка: 7
```

```
Час (років): 4
Кінцева сума: 1572.96
```

У програмі описується структура з назвою MyMoney. У структури декілька полів:

- поле name типу string (для використання цього класу підключається заголовок <string>) - призначене для запису імені вкладника;
- поля money і rate типу double - призначені відповідно для запису початкової суми вкладу й річної процентної ставки;
- цілочислове поле time - призначене для запису часу розміщення вкладу (в роках).

У програмі для обчислення за значеннями полів екземпляра структури MyMoney описано функцію getMoney(). Як аргумент функції передається екземпляр структури MyMoney (позначено як str). Як результат функція повертає значення типу double (кінцева сума по депозиту).

У тілі функції оголошується локальна змінна s із початковим значенням str.money (значення поля money екземпляра str). Далі запускається оператор циклу. Кількість циклів визначається значенням поля time екземпляра str. За кожний цикл виконується команда s*=(1+str.rate/100), якою поточне значення змінної s множиться на величину (1+str.rate/100). Після закінчення оператора циклу значення змінної s повертається як результат функції.

Подробиці



Нагадаємо, що якщо початкова сума дорівнює m, річна відсоткова ставка дорівнює r, а вклад розміщується на час t, то кінцева сума по депозиту обчислюється за формулою $m\left(1+\frac{r}{100}\right)^t$.

Функція show() не повертає результат. Як аргумент функції передається екземпляр структури MyMoney (позначено як str). Під час виклику функції в консольному вікні відображуються значення полів name, money, rate і time

екземпляра str. Для обчислення кінцевої суми по депозиту викликається функція getMoney(), а як аргумент їй передається екземпляр str.

У головній функції програми командами MyMoney cat={"Пан Коцький",1000,8,5} і MyMoney fox={"Лис Микита",1200,7,4} створюються й ініціалізуються (полям присвоюються значення) два екземпляри (cat і fox) структури MyMoney. Командами show(cat) і show(fox) у консольне вікно виводиться інформація по кожному з екземплярів.

Узагальнені структури

У структуру дозволяється передавати узагальнені параметри так, як це робиться в узагальнених класах. У лістингу 8.2 наведено програму, в якій створюється узагальнена структура з двома полями. Типи полів визначаються узагальненими параметрами. Крім структури, у програмі описано узагальнену функцію, призначену для роботи з екземплярами узагальненої структури.

Лістинг 8.2. Узагальнені структури

```
#include <iostream>
#include <cstdlib>
#include <string>
using namespace std;
// Узагальнена структура:
template<class A,class B> struct MyStruct{
   A first;
   B second;
};
// Узагальнена функція для роботи з екземплярами
// узагальненої структури:
template<class A,class B> void show(MyStruct<A,B> str){
   cout<<"Перше поле: "<<str.first<<endl;
   cout<<"Друге поле: "<<str.second<<endl;
}
```

```
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Створення екземплярів узагальненої структури:
    MyStruct<int,char> strA={100,'A'};
    MyStruct<double,string> strB={2.5,"текст"};
    // Виклик узагальненої функції:
    show(strA);
    show(strB);
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 8.2)

```
Перше поле: 100
Друге поле: A
Перше поле: 2.5
Друге поле: текст
```

Опис узагальненої структури починається з ключового слова `template`. Інструкція `<class A,class B>` визначає узагальнені параметри `A` і `B`. У тілі структури поля `first` і `second` описано з ідентифікаторами типа `A` і `B` відповідно. У головній функції програми командами `MyStruct<int,char> strA={100,'A'}` і `MyStruct<double,string> strB={2.5,"текст"}` створюються екземпляри `strA` і `strB` узагальненої структури `MyStruct`. Тут простежується явна аналогія з класами і об'єктами. Для відображення значень полів екземплярів узагальненої структури використовуємо функцію `show()`,

аргументом якій передається екземпляр структури (команди `show(strA)` і `show(strB)`).

Функцію `show()` описано як узагальнену. У неї в описі два параметри типу (`A` і `B`), і вона не повертає результат. Аргумент функції описаний як екземпляр `str` структури `MyStruct<A,B>`. Ідентифікація типів (значення параметрів `A` і `B`) під час виклику функції `show()` виконується на основі типу аргументу, переданого функції.

Робота з комплексними числами

У бібліотеці стандартних класів є клас `complex`, призначений для роботи з комплексними числами.

Теорія

Для використання класу `complex` у програмі підключається заголовок `<complex>`. Оскільки клас `complex` є узагальненим, то в кутових дужках зазначається базовий тип (зазвичай `double`) для дійсної та уявної частин комплексного числа. Значення дійсної та уявної частин комплексного числа передаються як аргументи конструктору.

У наведеній далі програмі створюють комплексні числа і з ними виконують базові обчислювальні операції.

Подробиці

Будь-яке комплексне число z може бути представлене у вигляді $z = x + iy$ (алгебраїчна форма представлення комплексного числа), де дійсні числа x і y є відповідно дійсною й уявною частинами комплексного числа, а уявна одиниця i така, що $i^2 = -1$.

Операції додавання, віднімання, множення й ділення виконують, як і зі звичайними числами, але з поправкою на те, що $i^2 = -1$. Наприклад, якщо $z_1 = x_1 + iy_1$ і $z_2 = x_2 + iy_2$, то $z_1 + z_2 = (x_1 + x_2) + i(y_1 + y_2)$, $z_1 - z_2 = (x_1 - x_2) + i(y_1 - y_2)$, $z_1 \cdot z_2 = (x_1 + iy_1) \cdot (x_2 + iy_2) = (x_1x_2 - y_1y_2) + i(x_1y_2 + x_2y_1)$, а $\frac{z_1}{z_2} = \frac{x_1+iy_1}{x_2+iy_2} = \frac{(x_1+iy_1)(x_2-iy_2)}{(x_2+iy_2)(x_2-iy_2)} = \frac{(x_1x_2+y_1y_2)+i(x_2y_1-x_1y_2)}{x_2^2+y_2^2}$.

Комплексно-спряженим до числа $z = x + iy$ називається число $z^* = x - iy$

(отримують із вихідного заміною i на $-i$).

Модуль комплексного числа $z = x + iy$ обчислюють як $|z| = \sqrt{x^2 + y^2}$.

Окрім алгебраїчної форми представлення комплексного числа $z = x + iy$, існує тригонометрична форма представлення числа у вигляді $z = |z| \exp(i\varphi)$, де через φ позначено аргумент комплексного числа такий, що $\cos(\varphi) = \frac{x}{\sqrt{x^2+y^2}}$ і $\sin(\varphi) = \frac{y}{\sqrt{x^2+y^2}}$ (тут треба врахувати, що за означенням $\exp(i\varphi) = \cos(\varphi) + i \cdot \sin(\varphi)$).

Розглянемо програмний код, наведений у лістингу 8.3.

Лістинг 8.3. Робота з комплексними числами

```
#include <iostream>
#include <cstdlib>
#include <complex>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Дійсні числа:
   double x=2,y=3;
   // Комплексні числа:
   complex<double> A(3,4),B(2,-1);
   // Сума комплексних чисел:
   cout<<"Сума: ";
   cout<<A<<" + "<<B<<" = "<<A+B<<endl;
   // Різниця комплексних чисел:
   cout<<"Різниця: ";
   cout<<A<<" - "<<B<<" = "<<A-B<<endl;
   // Добуток комплексних чисел:
   cout<<"Добуток: ";
   cout<<A<<" * "<<B<<" = "<<A*B<<endl;
   // Частка комплексних чисел:
   cout<<"Частка: ";
```

```
    cout<<A<<" / "<<B<<" = "<<A/B<<endl;
    // Сума комплексного і дійсного чисел:
    cout<<"Сума: ";
    cout<<A<<" + "<<x<<" = "<<A+x<<endl;
    // Різниця комплексного і дійсного чисел:
    cout<<"Різниця: ";
    cout<<A<<" - "<<x<<" = "<<A-x<<endl;
    // Добуток комплексного і дійсного чисел:
    cout<<"Добуток: ";
    cout<<A<<" * "<<x<<" = "<<A*x<<endl;
    // Частка комплексного і дійсного чисел:
    cout<<"Частка: ";
    cout<<A<<" / "<<x<<" = "<<A/x<<endl;
    // Сума дійсного і комплексного чисел:
    cout<<"Сума: ";
    cout<<y<<" + "<<B<<" = "<<y+B<<endl;
    // Різниця дійсного і комплексного чисел:
    cout<<"Різниця: ";
    cout<<y<<" - "<<B<<" = "<<y-B<<endl;
    // Добуток дійсного і комплексного чисел:
    cout<<"Добуток: ";
    cout<<y<<" * "<<B<<" = "<<y*B<<endl;
    // Частка дійсного і комплексного чисел:
    cout<<"Частка: ";
    cout<<y<<" / "<<B<<" = "<<y/B<<endl;
    // Дійсна частина комплексного числа:
    cout<<"Дійсна частина: ";
    cout<<"Re"<<A<<" = "<<A.real()<<endl;
    // Уявна частина комплексного числа:
    cout<<"Уявна частина: ";
    cout<<"Im"<<A<<" = "<<A.imag()<<endl;
    // Модуль комплексного числа:
```

```
    cout<<"Модуль: ";
    cout<<"abs"<<A<<" = "<<abs(A)<<endl;
    // Аргумент комплексного числа:
    cout<<"Аргумент: ";
    cout<<"arg"<<A<<" = "<<arg(A)<<endl;
    // Комплексно-спряжене число:
    cout<<"Комплексно-спряжене: ";
    cout<<A<<"* = "<<conj(A)<<endl;
    // Визначення числа на основі модуля й аргументу:
    cout<<"Визначення числа: ";
    cout<<polar(abs(A),arg(A))<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 8.3)

```
Сума: (3,4) + (2,-1) = (5,3)
Різниця: (3,4) - (2,-1) = (1,5)
Добуток: (3,4) * (2,-1) = (10,5)
Частка: (3,4) / (2,-1) = (0.4,2.2)
Сума: (3,4) + 2 = (5,4)
Різниця: (3,4) - 2 = (1,4)
Добуток: (3,4) * 2 = (6,8)
Частка: (3,4) / 2 = (1.5,2)
Сума: 3 + (2,-1) = (5,-1)
Різниця: 3 - (2,-1) = (1,1)
Добуток: 3 * (2,-1) = (6,-3)
Частка: 3 / (2,-1) = (1.2,0.6)
Дійсна частина: Re(3,4) = 3
Уявна частина: Im(3,4) = 4
```

```
Модуль: abs(3,4) = 5
Аргумент: arg(3,4) = 0.927295
Комплексно-спряжене: (3,4)* = (3,-4)
Визначення числа: (3,4)
```

Коротко резюмуючи, зазначимо, що об'єкт класу `complex` ототожнюється з комплексним числом. Із об'єктами можна виконувати ті самі операції, що й зі звичайними числами: додавати, віднімати, множити і ділити. Причому одним із операндів може бути число. Під час відображення об'єкта класу `complex` в консольному вікні за допомогою оператора виведення з'являються круглі дужки з парою числових значень у них - дійсна й уявна частини комплексного числа.

Деякі операції виконуються за допомогою спеціальних функцій: модуль числа можна обчислити за допомогою функції `abs()`, аргумент числа - за допомогою функції `arg()`, комплексно-спряжене число - за допомогою функції `conj()`. Функція `polar()` дозволяє створити комплексне число на основі значень його модуля й аргументу. Тому, наприклад, якщо `A` - об'єкт, який відповідає комплексному числу, то результатом виразу `polar(abs(A),arg(A))` буде таке саме число (простіше кажучи, якщо ми раніше створили об'єкт `A` для комплексного числа, то під час виконання команди `polar(abs(A),arg(A))` отримаємо таке саме число, проте реалізоване через інший об'єкт).

Методи `real()` і `imag()` (викликаються з об'єкта комплексного числа) повертають як результат відповідно дійсну й уявну частини комплексного числа.

Клас для реалізації числових масивів

Узагальнений клас `valarray` бібліотеки стандартних класів призначений для реалізації числових масивів. Під час роботи з класом підключається заголовок `<valarray>`. Далі наведено програму, в якій клас `valarray` використовується для створення і заповнення числового масиву числами

Фібоначчі. Специфіка задачі в тому, що розмір масиву визначається в процесі виконання програми.



До уваги

Тут і далі ми говоримо про масив, хоча насправді маємо справу з об'єктом класу `valarray`, який за властивостями схожий на масив. Використання терміна "масив" є зручне та інтуїтивно зрозуміле, проте все ж варто розуміти, що насправді йдеться про об'єкт.

Розглянемо програмний код, наведений у лістингу 8.4.

Лістинг 8.4. Розмір масиву визначається в процесі виконання програми

```
#include <iostream>
#include <cstdlib>
#include <valarray>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Змінна для запису розміру масиву:
   int n;
   cout<<"Вкажіть розмір масиву: ";
   // Зчитування розміру масиву:
   cin>>n;
   // Створення числового масиву:
   valarray<int> fibs(n);
   // Значення перших двох елементів:
   fibs[0]=1;
   fibs[1]=1;
   cout<<fibs[0]<<" "<<fibs[1];
   // Заповнення масиву та відображення значень елементів:
   for(int k=2;k<n;k++){
      fibs[k]=fibs[k-1]+fibs[k-2];
```

```
        cout<<" "<<fibs[k];
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Можливий результат виконання програми наведено нижче (жирним шрифтом виділено значення, яке вводить користувач):

Результат виконання програми (із лістингу 8.4)

```
Вкажіть розмір масиву: 15
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

У програмі оголошують цілочислову змінну `n`, значення якої користувач уводить з клавіатури. Змінна визначає кількість елементів у масиві. Масив реалізують через об'єкт узагальненого класу `valarray`. Об'єкт, ототожнений нами з масивом, створюють за допомогою команди `valarray<int> fibs(n)`. Інструкція `<int>` означає, що елементи масиву `fibs` (будемо для зручності називати цей об'єкт масивом) належать до типу `int`. Аргумент, зазначений у круглих дужках, визначає кількість елементів у масиві.

Хоча `fibs` є об'єктом, базові операції з ним виконуються так, як наче б це був звичайний масив. Зокрема, це зауваження стосується звертання до елементів масиву. Скажімо, першим двом елементам одиничні значення присвоюються командами `fibs[0]=1` і `fibs[1]=1` - тобто так, як наче б ми мали справу зі справжнім масивом. Після присвоювання значень елементам вони відображуються в консолі (команда `cout<<fibs[0]<<" "<<fibs[1]`). Потім запускається оператор циклу, в якому індексна змінна `k` пробігає значення від `2` до `n-1` включно. За кожний

цикл командою `fibs[k]=fibs[k-1]+fibs[k-2]` обчислюється значення для поточного елемента, а командою `cout<<" "<<fibs[k]` це значення відображається в консольному вікні.

До уваги

Нагадаємо, що числа Фібоначчі отримують так: перші два числа дорівнюють одиниці, а кожне наступне дорівнює сумі двох попередніх.

Вище ми використали об'єкт класу `valarray` фактично як звичайний масив. Однак використання об'єкта, порівняно з масивом, має беззаперечні переваги. Досить відзначити, що розмір масиву, "захованого" в об'єкті класу `valarray`, можна не лише дізнатись за допомогою методу `size()`, а й змінити за допомогою методу `resize()`. Таких можливостей під час роботи зі звичайними масивами немає.

Ще один невеликий приклад використання узагальненого стандартного класу `valarray` наведено у лістингу 8.5. Там подано програму, у якій розв'язується вже знайома нам задача про представлення полінома у вигляді об'єкта певного класу (таку задачу розглянуто в попередній главі - там використано узагальнені класи з передачею цілочислового значення як параметра). Ідея й алгоритм реалізовані практично ті самі, що й у програмі з лістингу 7.11. Однак за рахунок використання класу `valarray` код у багатьох моментах спростився.

До уваги

Нагадаємо, що йдеться про клас для реалізації полінома. Поліноміальна функція однозначно визначається набором коефіцієнтів, які записують у масив, що є полем класу. Для поліномів передбачено такі операції, як обчислення похідної, множення поліномів, сума і різниця поліномів, а також ряд інших. У всіх перерахованих випадках результатом є поліном. Більш докладно операції з поліномами описано в поясненнях до прикладу з лістингу 7.11 у попередній главі.

Розглянемо наведену нижче програму:

Лістинг 8.5. Клас для реалізації полінома

```
#include <iostream>
```

```
#include <cstdlib>
#include <valarray>
using namespace std;
// Клас для реалізації поліномів:
class Polynom{
private:
   // Закрите поле - масив, реалізований через
   // об'єкт класу valarray:
   valarray<double> a;
public:
   // Конструктор із цілочисловим аргументом:
   Polynom(int n=0){
      // Визначення розміру масиву і заповнення нулями:
      a.resize(n+1,0);
   }
   // Конструктор із аргументом - іменем числового масиву
   // і степенем полінома:
   Polynom(double* nums,int n){
      // Створення нового об'єкта для масиву:
      valarray<double> b(nums,n+1);
      // Присвоювання полю нового значення:
      a=b;
   }
   // Як результат методом повертається степінь полінома:
   int power(){
      return a.size()-1;
   }
   // Метод для відображення коефіцієнтів полінома:
   void getAll(){
      cout<<"| ";
      for(int k=0;k<=power();k++){
         cout<<a[k]<<" | ";
```

```
    }
    cout<<endl;
}
// Операторний метод дозволяє викликати об'єкт.
// Як результат повертається значення полінома:
double operator()(double x){
    // Копія масиву з коефіцієнтами полінома:
    valarray<double> b(a);
    double q=1;
    // Обчислення елементів масиву:
    for(int k=0;k<b.size();k++){
        b[k]*=q;
        q*=x;
    }
    // Результат методу - сума елементів масиву:
    return b.sum();
}
// Операторний метод для індексування об'єктів:
double &operator[](int k){
    return a[k];
}
// Операторний метод для обчислення
// добутку поліномів:
Polynom operator*(Polynom pol){
    // Локальний об'єкт:
    Polynom tmp(pol.power()+power());
    // Обчислення коефіцієнтів для полінома - результату:
    for(int i=0;i<=power();i++){
        for(int j=0;j<=pol.power();j++){
            tmp[i+j]+=a[i]*pol[j];
        }
    }
```

```
        // Результат методу:
        return tmp;
    }
    // Операторний метод для обчислення суми поліномів:
    Polynom operator+(Polynom pol){
        int i;
        int length=pol.power()>power()?pol.power():power();
        // Локальний об'єкт:
        Polynom tmp(length);
        // Обчислення коефіцієнтів для полінома - результату:
        for(i=0;i<=power();i++){
            tmp[i]+=a[i];
        }
        for(i=0;i<=pol.power();i++){
            tmp[i]+=pol[i];
        }
        // Результат методу:
        return tmp;
    }
};
// Операторна функція для обчислення
// результату множення полінома на число:
Polynom operator*(Polynom pol,double r){
    // Локальний об'єкт:
    Polynom tmp(pol.power());
    // Обчислення коефіцієнтів для полінома - результату:
    for(int k=0;k<=pol.power();k++){
        tmp[k]=pol[k]*r;
    }
    // Результат функції:
    return tmp;
}
```

```
// Операторна функція для обчислення
// результату множення числа на поліном:
Polynom operator*(double r,Polynom pol){
   // Поліном множиться на число:
   return pol*r;
}
// Операторна функція для обчислення різниці
// двох поліномів:
Polynom operator-(Polynom x,Polynom y){
   // До першого полінома додається другий, помножений
   // на число -1:
   return x+(-1)*y;
}
// Узагальнена функція для обчислення
// похідної від полінома:
Polynom Diff(Polynom pol){
   // Локальний об'єкт:
   Polynom tmp(pol.power()-1);
   // Обчислення коефіцієнтів для полінома - результату:
   for(int k=0;k<=tmp.power();k++){
      tmp[k]=pol[k+1]*(k+1);
   }
   // Результат функції:
   return tmp;
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Масиви з коефіцієнтами:
   double A[]={1,2,-1,1};
   double B[]={-1,3,0,2,-1,1};
```

```
    // Аргумент для поліномів:
    double x=2;
    // Об'єкт для запису результатів обчислень:
    Polynom res;
    // Перший поліном:
    Polynom P(A,3);
    cout<<"Поліном P:\t";
    // Коефіцієнти першого полінома:
    P.getAll();
    cout<<"Значення P("<<x<<") = ";
    // Значення першого полінома в точці:
    cout<<P(x)<<endl;
    // Похідна:
    res=Diff(P);
    cout<<"Поліном P':\t";
    // Коефіцієнти для похідної від полінома:
    res.getAll();
    cout<<"Значення P'("<<x<<") = ";
    // Значення похідної в точці:
    cout<<res(x)<<endl;
    // Другий поліном:
    Polynom Q(B,5);
    cout<<"Поліном Q:\t";
    // Коефіцієнти другого полінома:
    Q.getAll();
    cout<<"Значення Q("<<x<<") = ";
    // Значення другого полінома в точці:
    cout<<Q(x)<<endl;
    // Добуток поліномів:
    res=P*Q;
    cout<<"Поліном P*Q:\t";
    // Коефіцієнти для добутку поліномів:
```

```
    res.getAll();
    cout<<"Значення (P*Q)("<<x<<") = ";
    // Значення добутку поліномів у точці:
    cout<<res(x)<<endl;
    // Сума поліномів:
    res=P+Q;
    cout<<"Поліном P+Q:\t";
    // Коефіцієнти для суми поліномів:
    res.getAll();
    cout<<"Значення (P+Q)("<<x<<") = ";
    // Значення суми поліномів у точці:
    cout<<res(x)<<endl;
    // Різниця поліномів:
    res=Q-P;
    cout<<"Поліном Q-P:\t";
    // Коефіцієнти для різниці поліномів:
    res.getAll();
    cout<<"Значення (Q-P)("<<x<<") = ";
    // Значення різниці поліномів у точці:
    cout<<res(x)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 8.5)

```
Поліном P:      | 1 | 2 | -1 | 1 |
Значення P(2) = 9
Поліном P':     | 2 | -2 | 3 |
Значення P'(2) = 10
Поліном Q:      | -1 | 3 | 0 | 2 | -1 | 1 |
```

```
Значення Q(2) = 37
Поліном P*Q:     | -1 | 1 | 7 | -2 | 6 | -3 | 5 | -2 | 1 |
Значення (P*Q)(2) = 333
Поліном P+Q:     | 0 | 5 | -1 | 3 | -1 | 1 |
Значення (P+Q)(2) = 46
Поліном Q-P:     | -2 | 1 | 1 | 1 | -1 | 1 |
Значення (Q-P)(2) = 28
```

У багатьох місцях код повинен бути знайомий і зрозумілий читачеві. Ми виділимо лише те нове, що з'явилося в програмі (порівняно з програмою з лістингу 7.11).

Отже, використання об'єкта класу `valarray` як контейнера для масиву дозволяє відмовитися від опису класу `Polynom` як узагальненого. Тепер це звичайний клас. Закрите поле `a`, через яке реалізується масив з коефіцієнтами полінома, у класі оголошується інструкцією `valarray<double> a`. У цьому випадку оголошується пустий об'єкт (не містить масиву) класу `valarray<double>`. Параметри об'єкта змінюються під час виклику конструктора (нагадаємо, що розмір масиву, що міститься в об'єкті класу `valarray`, можна змінювати). Передбачено дві версії конструктора: з одним аргументом (зі значенням аргументу за замовчуванням) і з двома аргументами. Якщо конструктор викликається з одним аргументом, то аргумент визначає розмір масиву, а сам масив заповнюється нулями. Зокрема, ці дві операції виконуються однією командою `a.resize(n+1,0)` (через `n` позначено цілочисловий аргумент конструктора). Перший аргумент методу `resize()` визначає розмір масиву, а другий аргумент методу визначає значення, яке присвоюється елементам масиву.

До уваги

Кількість елементів у масиві на одиницю більша за значення аргументу, переданого конструктору. Аргумент конструктора інтерпретується як степінь полінома, який реалізується через об'єкт класу `Polynom`.

Якщо аргументів у конструктора два, то перший (позначено як nums) є іменем числового масиву, а другий (позначено як n) визначає степінь полінома, що реалізується через цей масив (на одиницю менший за розмір масиву). У тілі такого конструктора командою valarray<double> b(nums,n+1) створюється об'єкт b класу valarray<double>, у якому міститься масив з такими самими елементами, як і масив nums. Командою a=b об'єкт b присвоюється як значення об'єкту a.

У класі Polynom описано метод power(), який як значення повертає степінь полінома, що реалізується через об'єкт. Це значення на одиницю менше за кількість елементів у масиві a і обчислюється за допомогою виразу a.size()-1.

До уваги

Функція power() використовується у програмі там, де в прикладі з лістингу 7.11 використовувався параметр power.

В операторному методі operator()() зазнав змін алгоритм обчислення значення полінома в точці. У тілі методу командою valarray<double> b(a) створюється копія (об'єкт b) об'єкта a. Потім у тілі оператора циклу, в якому перебираються елементи в масиві b, командою b[k]*=q черговий елемент множиться на значення змінної q, а потім командою q*=x змінна q множиться на значення аргументу x методу. Оскільки початкове значення змінної q дорівнює одиниці, то в підсумку виходить, що в масиві об'єкта b значення елементів збігаються з доданками у виразі для полінома (коефіцієнт, помножений на аргумент x у відповідному степені). Тому все, що залишається - обчислити суму елементів у масиві об'єкта b. Зробити це можна за допомогою методу sum(), який викликають з об'єкта b (команда b.sum()).

У всьому іншому код повинен бути зрозумілим читачу. Єдине, на що ще хочеться звернути увагу - оголошення об'єкта res класу Polynom. Як

значення змінній res послідовно присвоюється результат таких виразів, як добуток поліномів (об'єктів класу Polynom), їхня сума, різниця і результат виклику функції Diff(). Відображення коефіцієнтів і обчислення значення поліномів в точці виконуються через звертання до об'єктної змінної res.

Контейнер для динамічного масиву

У стандартній бібліотеці є клас vector, який властивостями нагадує динамічний масив, але порівняно з масивом функціональні можливості класу набагато ширші.

До уваги

Для використання класу vector підключають заголовок <vector>.

У лістингу 8.6 запропоновано програму, в якій створюється масив і заповнюється випадковими символами. Масив реалізується через об'єкт класу vector (ситуація схожа на випадок, коли масив реалізувався через об'єкт класу valarray). Розмір масиву визначається в процесі виконання програми. Розглянемо наведений далі програмний код:

Лістинг 8.6. Випадкові символи і клас vector

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Ініціалізація генератора випадкових чисел:
    srand(2);
    // Кількість символів:
    int n;
    cout<<"Кількість символів: ";
    // Зчитування значення змінної:
```

```
    cin>>n;
    // Об'єкт із символьним масивом:
    vector<char> symbs(n);
    cout<<"|";
    // Обчислення значень елементів і їх
    // відображення в консольному вікні:
    for(int k=0;k<symbs.size();k++){
        // Генерування випадкового символу:
        symbs[k]='A'+rand()%(n+5);
        // Відображення значення елемента:
        cout<<" "<<symbs[k]<<" |";
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Можливий результат (можливий - тому що використовується генератор випадкових чисел) виконання програми наведено нижче (жирним шрифтом виділено введене користувачем значення):

Результат виконання програми (із лістингу 8.6)

```
Кількість символів: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Хоча код простий, проаналізуємо його. А саме, варто звернути увагу, що об'єкт із масивом створюється командою `vector<char> symbs(n)`. Тут ідеться про об'єкт `symbs`, який створюється на основі узагальненого класу `vector` і містить масив із `n` елементів типу `char`.

Поводитися з об'єктом `symbs` можна як зі звичайним масивом. Зокрема, об'єкт індексується. Метод `size()`, що викликається з об'єкта, дозволяє визначити розмір масиву. Тому в операторі циклу значення елементам масиву в

об'єкті `symbs` присвоюється командою `symbs[k]='A'+rand()%(n+5)`, при цьому індексна змінна `k` пробігає значення від `0` до `symbs.size()-1` включно. Відображається елемент у консольному вікні командою `cout<<" "<<symbs[k]<<" |"`.

Цю саму задачу, але реалізовану за допомогою *ітераторів*, наведено в програмному коді в лістингу 8.7.

Теорія

Ітератор - об'єкт, що має властивості вказівника. Іншими словами, ітератор є об'єктом, із яким можна виконувати операції так, як наче б об'єкт був вказівником. Ітератори створюють для роботи з контейнерними класами - такими, наприклад, як узагальнений клас `vector`.

Принципи використання ітераторів обговоримо на прикладі, наведеному нижче:

Лістинг 8.7. Використання ітераторів

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Кількість символів:
   int n;
   cout<<"Кількість символів: ";
   // Зчитування значення:
   cin>>n;
   // Об'єкт із символьним масивом:
   vector<char> symbs(n,'A');
```

```
    // Ітератор:
    vector<char>::iterator p;
    cout<<"|";
    // Обчислення значень елементів масиву
    // і відображення цих значень у консолі:
    for(p=symbs.begin();p!=symbs.end();p++){
       // Обчислення значення елемента
       // (доступ через ітератор):
       *p+=rand()%(n+5);
       // Відображення значення елемента:
       cout<<" "<<*p<<" |";
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий самий, як і в попередньому випадку (жирний шрифт - те, що вводить користувач):

Результат виконання програми (із лістингу 8.7)

```
Кількість символів: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Контейнерний об'єкт для реалізації масиву створюється командою `vector<char> symbs(n,'A')`. Другий аргумент `'A'`, який передається конструктору, є значенням для елементів масиву. Простіше кажучи, у створеному масиві в усіх елементів значення буде `'A'`. Для роботи з цим контейнерним об'єктом створюємо ітератор `p`. Оголошуємо ітератор командою `vector<char>::iterator p`. Відповідно до неї ітератор `p` є об'єктом класу `iterator`, який є внутрішнім класом класу `vector<char>`.

До уваги

Внутрішній клас - клас, описаний в іншому (зовнішньому) класі.

Метод `begin()` об'єкта `symbs` як результат повертає ітератор, що вказує на початковий елемент у масиві об'єкта `symbs`. В операторі циклу в першому блоці командою `p=symbs.begin()` змінній `p` присвоюється як значення об'єкт-ітератор, що вказує на початковий елемент масиву. У третьому блоці розташовано команду `p++`, якою значення ітератора "збільшується на одиницю": в результаті нове значення змінної `p` буде ітератором, що вказує на наступний елемент масиву.

До уваги

Загалом, щоб зрозуміти принцип роботи з ітератором, розумно інтерпретувати ітератор як вказівник.

Умовою, що перевіряють, в операторі циклу зазначено вираз `p!=symbs.end()`. У ньому використано результат виклику методу `end()` із об'єкта `symbs`. Як результат метод `end()` повертає ітератор, що вказує на область пам'яті, розташовану відразу за останнім елементом. Умова `p!=symbs.end()` істинна, поки ітератор `p` указує на елемент масиву, і стає хибною, коли ітератор "вийде" за межі масиву. В такому випадку виконання оператора циклу закінчується.

За аналогією з вказівником, щоб отримати доступ до елемента, на який указує ітератор, перед ітератором слід поставити зірочку `*`. Тому в результаті виконання команди `*p+=rand()%(n+5)` до початкового значення елемента (а це значення `'A'`), на який указує ітератор `p`, додається значення `rand()%(n+5)` (і через механізм автоматичного зведення типів отриманий у підсумку код перетворюється на символ). Командою `cout<<" "<<*p<<" |"` значення елемента, на який указує ітератор `p`, відображається в консольному вікні.

У розглянутих прикладах контейнерний об'єкт потрібного розміру створювався одразу. Підхід може бути іншим, коли розмір масиву, "схованого" в контейнерному об'єкті, збільшується у міру додавання елементів у масив. Подібний спосіб організації коду проілюстровано в програмі з лістингу 8.8.

Лістинг 8.8. Зміна розмірів контейнерного об'єкта

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Ініціалізація генератора випадкових чисел:
   srand(2);
   // Кількість елементів:
   int n;
   cout<<"Кількість символів: ";
   // Зчитування значення для змінної:
   cin>>n;
   // Створення пустого контейнерного об'єкта:
   vector<char> symbs;
   cout<<"|";
   // Додавання елементів у контейнерний об'єкт:
   while(symbs.size()<n){
      // У кінець масиву додається новий елемент:
      symbs.push_back('A'+rand()%(n+5));
      // Відображення значення
      // останнього елемента в масиві:
      cout<<" "<<symbs[symbs.size()-1]<<" |";
   }
   cout<<endl;
```

```
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми (жирним шрифтом виділено уведене користувачем значення):

Результат виконання програми (із лістингу 8.8)

```
Кількість символів: 12
| L | K | H | N | G | P | E | N | M | H | P | D |
```

Бачимо, що результат такий самий, як у попередніх випадках. Тепер спробуємо зрозуміти, чому все відбувається саме так, а не інакше.

Командою `vector<char> symbs` у програмі створюється пустий (без масиву або з масивом нульової довжини) контейнерний об'єкт `symbs` (розмір внутрішнього масиву в цьому об'єкті змінюється у процесі виконання програми). Для додавання елементів у контейнерний об'єкт запускається оператор циклу `while`. Умова в операторі циклу `symbs.size()<n` полягає в тому, що розмір масиву в об'єкті `symbs` менший за значення `n`, яке користувач увів із клавіатури.

В операторі циклу командою `symbs.push_back('A'+rand()%(n+5))` у кінець масиву в об'єкті `symbs` додається новий елемент, а значення цього елемента визначається аргументом методу `push_back()`. Далі, оскільки метод `size()` повертає кількість елементів у масиві об'єкта, з якого викликано метод, то індекс останнього елемента на одиницю менший за результат методу `size()`. Таким чином, командою `cout<<" "<<symbs[symbs.size()-1]<<" |"` відображається значення останнього (на цей момент) елемента в масиві. Нескладно здогадатися, що йдеться про той елемент, який було додано в масив попередньою командою.

Контейнерний клас для реалізації множини

Модифікуємо розглянуту вище задачу про заповнення масиву випадковими символами, накладаючи додаткове обмеження: символи повинні бути різними. Також нас буде цікавити тільки набір символів, безвідносно до порядку, в якому їх було згенеровано.

Для розв'язання задачі скористаємося контейнерним класом `set`, що дозволяє реалізувати *множини*.

Подробиці



Множина (з математичної точки зору) є невпорядкований набір різних елементів. Таким чином, у множини не може бути двох ідентичних елементів, а порядок елементів у множині не фіксований - має значення лише входить елемент у множину чи ні. Додавання у множину елемента, який там уже є, насправді не спричинює появу у множині нового елемента. Простіше кажучи, елемент додається у множину, якщо там такого елемента ще немає.

У лістингу 8.9 наведено програму, в якій на основі класу `set` створюється і заповнюється множина, що складається з випадкових символів.

Лістинг 8.9. Реалізація множини на основі класу set

```
#include <iostream>
#include <cstdlib>
#include <set>
using namespace std;
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Ініціалізація генератора випадкових чисел:
    srand(2);
    // Кількість різних символів:
```

```
int n;
cout<<"Кількість різних символів: ";
// Зчитування значення:
cin>>n;
// Створення пустого контейнерного об'єкта:
set<char> symbs;
// Символьна змінна:
char s;
// Лічильник символів:
int counter=0;
// Заповнення множини:
while(symbs.size()<n){
   // Випадковий символ:
   s='A'+rand()%(n+5);
   // Зміна значення лічильника символів:
   counter++;
   // Відображення згенерованого символу:
   cout<<s<<" ";
   // Додавання елемента у множину:
   symbs.insert(s);
}
cout<<"\nВсього згенеровано символів: "<<counter<<"\n";
cout<<"Різні символи:\n";
// Ітератор для роботи з елементами множини:
set<char>::iterator p;
// Ітератор установлено на початковий елемент множини:
p=symbs.begin();
cout<<"|";
// Відображається вміст множини:
while(p!=symbs.end()){
   // Поточний елемент:
   cout<<" "<<*p<<" |";
```

```
        // Ітератор на наступний елемент:
        p++;
    }
    cout<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче показано, як може виглядати результат виконання програми (жирним шрифтом виділено введене користувачем значення):

Результат виконання програми (із лістингу 8.9)

```
Кількість різних символів: 12
L K H N G P E N M H P D E L G N M F L K G M N K D O M K M P Q
Всього згенеровано символів: 31
Різні символи:
| D | E | F | G | H | K | L | M | N | O | P | Q |
```

Розглянемо більш детально програмний код. Командою `set<char> symbs` створюється контейнерний об'єкт для реалізації множини, в якій на момент створення немає елементів. Символьна змінна `s` призначена для запам'ятовування згенерованих символів, а в цілочислову змінну `counter` (з нульовим початковим значенням) записується кількість *усіх* згенерованих символів. При цьому значення змінної `n`, яке зчитується з клавіатури, визначає кількість *різних* символів. Заповнення множини виконується в операторі циклу `while`, у якому перевіряється умова `symbs.size()<n` (кількість елементів у множині менша за значення змінної `n`).

У тілі оператора циклу командою `s='A'+rand()%(n+5)` генерується випадковий символ і записується в змінну `s`. Після цього значення змінної `counter` збільшується на одиницю, а згенерований символ відображується в консольному вікні. Далі командою `symbs.insert(s)` у множину додається

елемент. Але на відміну від випадку з масивом, під час додавання елемента у множину елемент насправді додається, тільки якщо у множині такого елемента немає. Оператор циклу закінчується, коли розмір множини дорівнюватиме n - причому за визначенням всі елементи у множині мають різні значення.

Для відображення елементів множини в програмі командою `set<char>::iterator p` оголошується ітератор для роботи з елементами множини. Ітератор установлюється на початковий елемент множини за допомогою команди `p=symbs.begin()`. В операторі циклу `while` командою `cout<<" "<<*p<<" |"` відображається значення поточного (на який указує ітератор `p`) елемента множини, після чого командою `p++` ітератор перекидається на наступний елемент. Процес триває, поки виконується умова `p!=symbs.end()` - тобто поки ітератор не стане указувати на область пам'яті за останнім елементом у множині.

До уваги

Оскільки елементи у множині не впорядковані, то сказати однозначно, який елемент перший, а який останній досить проблематично. Важливо те, що якийсь із них перший, а якийсь останній. Якщо подивитися на результат виконання програми, то легко помітити, що елементи множини відображаються в алфавітному порядку, а не в тому, як вони додавалися у множину.

Асоціативний контейнер

Контейнерний клас `map` дозволяє створювати *асоціативні контейнери* - аналог масиву, в якому роль індексів можуть відігравати значення нечислових типів. Аналог індексу називається *ключем* елемента. Таким чином, у кожного елемента в асоціативному масиві є значення, і є ключ.

Теорія

Для використання контейнерного класу `map` у програмі підключається заголовок `<map>`. В узагальненого класу `map` два узагальнених параметри: тип ключа і тип значення (вказують у кутових дужках після імені класу).

Доступ до елементу контейнера здійснюється, крім іншого, через ключ: його вказують

(як індекс) у квадратних дужках після імені об'єкта чи передають як аргумент методу `at()`, який викликають із об'єкта контейнера.

Вставка елемента в контейнерний об'єкт може виконуватися за допомогою методу `insert()`. Як аргумент методу передається об'єкт типу `pair` (назва узагальненої структури, для використання якої підключається заголовок `<utility>`). У екземпляра структури `pair` є два поля: поле `first` відповідає ключу елемента, а поле `second` - значенню елемента.

Приклад використання асоціативного масиву, створеного на основі контейнерного класу `map`, наведено у лістингу 8.10.

Лістинг 8.10. Асоціативний контейнер

```
#include <iostream>
#include <cstdlib>
#include <string>
#include <map>
#include <utility>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Об'єкт для асоціативного контейнера:
   map<string,int> numbers;
   // Розмір масиву:
   const int n=5;
   // Текстовий масив із ключами:
   string names[n]={"один","два","три","чотири","п'ять"};
   // Числовий масив зі значеннями елементів:
   int nms[n]={1,2,3,4,5};
   // Додавання елементів у контейнер:
   for(int k=0;k<n;k++){
      numbers.insert(pair<string,int>(names[k],nms[k]));
   }
```

```
    // Додається ще один елемент:
    numbers.insert(pair<string,int>("шість",6));
    // Видалення елемента з масиву:
    numbers.erase("три");
    // Ітератор для роботи з асоціативним
    // контейнерним об'єктом:
    map<string,int>::iterator p;
    // Ітератор установлено на початковий елемент:
    p=numbers.begin();
    // Відображення вмісту контейнера:
    while(p!=numbers.end()){
       cout<<(*p).first<<"\t- "<<(*p).second<<endl;
       p++;
    }
    // Явне звернення до елементів за ключем:
    cout<<"Одиниця\t - це "<<numbers["один"]<<endl;
    cout<<"Двійка\t - це "<<numbers.at("два")<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 8.10)

```
два      - 2
один     - 1
п'ять    - 5
чотири   - 4
шість    - 6
Одиниця  - це 1
Двійка   - це 2
```

Прокоментуємо програмний код. У програмі командою `map<string,int> numbers` створюється об'єкт `numbers`, який є асоціативним контейнером (поки що без елементів). Ключами елементів у контейнері можуть бути значення типа `string`, а значення елементів належать до типу `int`.

Для автоматичного заповнення контейнера оголошуються два масиви: текстовий `names` із текстовими назвами цифр (значення для ключів елементів контейнера) і числовий масив `nms` із числовими значеннями (значення для елементів контейнера). Заповнюється контейнер `numbers` за допомогою оператора циклу, в якому індексна змінна `k` перебирає індекси в масивах `names` і `nms` (масиви однакового розміру). Додавання елементів у контейнер виконується за допомогою команди `numbers.insert(pair<string,int>(names[k],nms[k]))`. Для вставки елемента в контейнер використано метод `insert()`, аргументом якому передається елемент, що додається. Елемент у цьому випадку - це два параметри (ключ елемента і значення елемента). Елемент формується за допомогою інструкції `pair<string,int>(names[k],nms[k])`. Тут йдеться про створення анонімного екземпляра структури узагальненого типу `pair` (з використанням типів `string` і `int`). Два значення (`names[k]` і `nms[k]`), зазначені в круглих дужках - відповідно ключ і значення елемента.

Команда `numbers.insert(pair<string,int>("шість",6))` після оператора циклу дає ще один приклад вставки елемента в контейнер, але тепер ключ і значення елемента вказані у вигляді літералів (текстового і цілочислового).

Після додавання елементів у контейнер, один із них видаляється з контейнера. Використовуємо для цього команду `numbers.erase("три")`, у якій із об'єкта контейнера викликається метод `erase()`, а як аргумент йому передається ключ елемента, що видаляється.

Відображення вмісту контейнера реалізуємо через ітератор. Ітератор для роботи з асоціативним контейнерним об'єктом оголошується командою `map<string,int>::iterator p`. Командою `p=numbers.begin()` ітератор установлюється на початковий елемент контейнера.

> ***До уваги***
>
> Оскільки в контейнерному об'єкті елементи не впорядковано, то поняття "перший" і "останній" відносно елементів досить умовне. Зрозуміло, що якось елементи в пам'яті розподілені, і під час перебирання якийсь із них переглядається першим, а якийсь останнім. Але все одно як такого порядку розташування елементів у контейнері немає.

Відображається вміст контейнерного об'єкта за допомогою оператора циклу `while`. Цикл виконується, поки істинна умова `p!=numbers.end()` (тобто поки ітератор установлено на елемент контейнера). В тілі оператора циклу відображаються значення `(*p).first` для ключа елемента і `(*p).second` для значення елемента, на який установлено ітератор `p`. Після цього командою `p++` ітератор "перекидається" на наступний елемент. Інший приклад звернення до елементів контейнера за ключем дають інструкції `numbers["один"]` і `numbers.at("два")`.

Обробка помилок

Раніше ми вже зустрічалися з обробкою виключень - правда, ми їх генерували вручну для створення точок розгалужування в програмі. Тепер розглянемо приклад, у якому механізм обробки виключень використовується, так би мовити, "за прямим призначенням".

У задачі про створення масиву з числами Фібоначчі (з лістингу 8.4) розмір масиву вводять із клавіатури. Теоретично можлива ситуація, коли користувач уводить некоректне значення для розміру масиву (наприклад, від'ємне число). У такому випадку під час спроби створити масив виникає помилка. Щоб обробити подібну помилку, доповнимо програмний код блоком `try-catch`,

завдяки чому під час уведення від'ємного значення для розміру масиву аварійного закінчення програми не буде. Розглянемо програмний код у лістингу 8.11.

Лістинг 8.11. Обробка помилок

```
#include <iostream>
#include <cstdlib>
#include <vector>
using namespace std;
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Змінна для запису розміру масиву:
   int n;
   cout<<"Числа Фібоначчі\n";
   cout<<"Уведіть розмір масиву: ";
   // Зчитування розміру масиву:
   cin>>n;
   // Контрольований код:
   try{
      // Створення числового масиву:
      vector<int> fibs(n,1);
      // Відображення значень перших двох елементів:
      cout<<fibs[0]<<" "<<fibs[1];
      // Заповнення масиву й відображення
      // значень елементів масиву:
      for(int k=2;k<n;k++){
         fibs[k]=fibs[k-1]+fibs[k-2];
         cout<<" "<<fibs[k];
      }
   }
   // Блок перехоплення помилки:
   catch(...){
```

```
        cout<<"Виникла помилка!";
    }
    cout<<"\nПрограма закінчила виконання...\n";
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми може бути таким (якщо користувач увів коректне значення для розміру масиву):

Результат виконання програми (із лістингу 8.11)

```
Числа Фібоначчі
Уведіть розмір масиву: 10
1 1 2 3 5 8 13 21 34 55
Програма закінчила виконання...
```

Або таким (якщо введено некоректне значення):

Результат виконання програми (із лістингу 8.11)

```
Числа Фібоначчі
Уведіть розмір масиву: -3
Виникла помилка!
Програма закінчила виконання...
```

У цій програмі ми для створення масиву з числами Фібоначчі використовуємо контейнерний клас `vector`. Програмний код із командою `vector<int> fibs(n,1)` для створення об'єкта `fibs` з внутрішнім масивом розташовано в `try`-блоці.

До уваги

Масив складається з `n` елементів із одиничними значеннями, тому відпадає необхідність у явному вигляді присвоювати одиничні значення першим двом елементам.

Якщо під час виконання команди зі створення масиву і наступних команд помилок не виникає, то `catch`-блок ігнорується. Якщо ж виникає помилка, то код у `try`-блоці припиняє виконуватися, і буде виконано код у `catch`-блоці.

До уваги

Три крапки в круглих дужках після інструкції `catch` означають, що цей блок перехоплює помилки всіх типів, які можуть виникнути в `try`-блоці.

Знайомство з багатопоточним програмуванням

У наступному прикладі в програмі виконується декілька *потоків*.

Теорія

Потоки - різні частини програми, які виконуються одночасно. Загальна схема така: у головному потоці, який ототожнюється з виконанням функції `main()`, можуть запускатися ***дочірні потоки***. Код, який виконується в рамках дочірніх потоків, зазвичай оформлюється у вигляді функції чи функцій, які викликаються в головному потоці в режимі паралельного виконання коду.

Для реалізації в програмі багатопоточного підходу підключається заголовок `<thread>`. Для запуску дочірнього потоку в головному потоці створюється об'єкт класу `thread`, а як аргумент конструктору під час створення об'єкта передається ім'я функції, яка виконується в рамках дочірнього потоку.

У лістингу 8.12 увазі читача пропонується достатньо простий приклад програми, у якій окрім головного потоку виконується ще й два дочірніх потоки. У головному потоці і двох дочірніх потоках у консольне вікно виводяться повідомлення. Специфіка ситуації в тому, що виведення повідомлень виконується одночасно всіма потоками. Для початку є сенс поглянути на наведений далі програмний код:

Лістинг 8.12. Багатопоточне програмування

```
#include <iostream>
#include <cstdlib>
```

```
#include <string>
#include <thread>
#include <chrono>
#include <mutex>
using namespace std;
mutex m;
// Функція для створення потоків:
void mythread(string name,int time,int steps){
    for(int k=1;k<=steps;k++){
        // Тимчасова затримка у виконанні команд:
        this_thread::sleep_for(chrono::seconds(time));
        // Блокування доступу до ресурсу (консолі):
        m.lock();
        // Відображення повідомлення в консолі:
        cout<<"Потік "<<name<<":\tповідомлення "<<k<<endl;
        // Розблокування ресурсу (консолі):
        m.unlock();
    }
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Кількість циклів у потоках:
    int n=5;
    cout<<"Запускаються потоки...\n";
    // Перший дочірній потік:
    thread A(mythread,"Alpha",4,n);
    // Другий дочірній потік:
    thread B(mythread,"Bravo",3,n);
    // Виклик функції в головному потоці:
    mythread("Main",2,n);
```

```
    // Очікування закінчення першого дочірнього потоку:
    if(A.joinable()){
       A.join();
    }
    // Очікування закінчення другого дочірнього потоку:
    if(B.joinable()){
       B.join();
    }
    cout<<"Виконання програми закінчено...\n";
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Можливий результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 8.12)

```
Запускаються потоки...
Потік Main:     повідомлення 1
Потік Bravo:    повідомлення 1
Потік Alpha:    повідомлення 1
Потік Main:     повідомлення 2
Потік Main:     повідомлення 3
Потік Bravo:    повідомлення 2
Потік Alpha:    повідомлення 2
Потік Main:     повідомлення 4
Потік Bravo:    повідомлення 3
Потік Main:     повідомлення 5
Потік Alpha:    повідомлення 3
Потік Bravo:    повідомлення 4
Потік Bravo:    повідомлення 5
Потік Alpha:    повідомлення 4
Потік Alpha:    повідомлення 5
```

```
Виконання програми закінчено...
```

Не зайвим буде проаналізувати й пояснити ключові елементи програми. У першу чергу слід охарактеризувати її загальну структуру.

У програмі крім функції `main()` описано ще й функцію `mythread()` із трьома аргументами. У головній функції `main()` функція `mythread()` використовується тричі: функція викликається в двох дочірніх потоках, а також є команда виклику функції безпосередньо в функції `main()`.

Загальний алгоритм виконання функції `mythread()` такий. Під час виклику функції запускається оператор циклу, в якому певну кількість раз у консольне вікно виводиться повідомлення. У повідомленні (для ідентифікації потоку) міститься значення першого текстового аргументу функції `mythread()` і номер повідомлення. Кількість виведених повідомлень визначається третім аргументом функції `mythread()`. Між виведенням повідомлень робиться пауза. Тривалість паузи (у секундах) визначається другим аргументом функції `mythread()`. Як все це організовано в тілі функції ми розглянемо далі, а зараз проаналізуємо код функції `main()`.

Отже, у тілі головної функції оголошують змінну `n` зі значенням `5`. Це значення визначає кількість повідомлень, яке виводиться в консоль кожним із потоків. Перед запуском дочірніх потоків командою `cout<<"Запускаються потоки...\n"` у консоль виводиться повідомлення, яке є індикатором того, що починається запуск дочірніх потоків. Перший дочірній потік запускається завдяки команді `thread A(mythread,"Alpha",4,n)`. У результаті виконання команди створюється об'єкт `A` класу `thread` (для використання класу `thread` підключають заголовок `<thread>`). Об'єкт `A` ототожнюється з першим дочірнім потоком. Доступ до цього потоку здійснюється через об'єкт `A`. Аргументи, передані конструктору під час створення об'єкта `A`, означають, що в рамках цього потоку буде виконуватися код функції `mythread()` із

аргументами "Alpha", 4 і n (усього буде n повідомлень із текстом "Alpha" й інтервалом між повідомленнями в 4 секунди). Щойно потік створено, не чекаючи його закінчення і паралельно з ним, виконується команда thread B(mythread,"Bravo",3,n), що означає створення іншого дочірнього потоку. Доступ до потоку здійснюється через об'єкт B, а в рамках потоку виконується функція mythread() із аргументами "Bravo", 3 і n (усього буде n повідомлень із текстом "Bravo" й інтервалом між повідомленнями в 3 секунди). У результаті, нарівні з головним потоком (код функції main()) буде виконуватися вже два дочірні потоки. Далі, командою mythread("Main",2,n) в головному потоці запускається функція mythread() (з'явиться n повідомлень із текстом "Main" й інтервалом між повідомленнями в 2 секунди).

Отримуємо три паралельних процеси (головний і два дочірні потоки), і в них різний час виконання. Нам необхідно забезпечити ситуацію, коли до закінчення головного потоку закінчаться всі дочірні потоки. Для цього використовуємо команди виду A.join() і B.join(). Команди означають, що перед виконанням наступної інструкції потрібно дочекатися закінчення відповідно першого потоку і другого потоку. Однак, перед тим як викликати метод join() із об'єкта відповідного потоку, необхідно перевірити, що потік ще виконується. Перевірити цю обставину можна зо допомогою методу joinable(), який викликається з об'єкта потоку, а сама інструкція передається як умова в умовний оператор. Після закінчення двох дочірніх потоків командою cout<<"Виконання програми закінчено...\n" у головному потоці виводиться повідомлення про закінчення програми.

Тепер розберемося з програмним кодом функції mythread(). Код функції простий і невеликий, але містить декілька нових для нас команд і незнайомих (поки ще) синтаксичних конструкцій. Зокрема, командою this_thread::sleep_for(chrono::seconds(time)) виконується блокування виконання команд на час (у секундах), який визначає аргумент

time функції. Використання цієї команди можливе завдяки підключенню заголовку <chrono> у програмі. Команда означає таке:

- Викликається функція sleep_for(), описана в просторі імен this_thread (цей простір імен доступний після підключення заголовка <thread>). Простір імен, у якому визначено функцію, вказується явно перед іменем функції і відокремлюється від імені функції оператором розширення області видимості ::.
- Як аргумент функції sleep_for() передається результат виклику функції seconds() (результатом функції є об'єкт, що визначає характеристики інтервалу для затримки у виконанні коду) із простору імен chrono. Простір імен chrono доступний після підключення заголовка <chrono>.

Перед командою cout<<"Потік "<<name<<":\tповідомлення " <<k<<endl, якою в консольному вікні відображається повідомлення, виконується команда m.lock(), а після відображення повідомлення виконується команда m.unlock(). Якщо коротко, то виконання команди m.lock() спричинює блокування консольного вікна, а виконання команди m.unlock() таке блокування знімає. Якщо детальніше, то ситуація така: оскільки декілька потоків виконуються одночасно, і всі вони звертаються до консольного вікна, то теоретично можливо, що в процес виведення повідомлення одним потоком може "вклинитися" інший потік, так що всередині одного повідомлення з'являється інше. Ми хочемо перестрахуватися і досягти того, що виведення повідомлення в консоль одним потоком спричинює автоматичне блокування консолі для інших потоків. Щойно повідомлення виведено, консоль розблокується для використання іншими потоками. Блокування і розблокування ресурсів (синхронізація потоків) виконується за допомогою м'ютексного об'єкта. М'ютексний об'єкт - об'єкт класу mutex. Клас доступний після підключення заголовка <mutex>. Об'єкт повинен бути доступний у всіх потоках, тому створюється як глобальний командою mutex m. Коли якийсь із потоків за допомогою команди m.lock() викликає із

м'ютексного об'єкта m метод блокування lock(), то це є індикатором для інших потоків про блокування ресурсу. Ресурс знову доступний після того, як у потоці, що заблокував ресурс, із м'ютексного об'єкта викликається метод unlock().

Рекомендації для самостійної роботи

- У задачах із попередніх глав, у яких використовувалися класи (включаючи узагальнені) й об'єкти, розгляньте можливість використання структур і зовнішніх функцій.
- У задачах із попередніх глав, у яких використовувалися статичні та динамічні масиви, розгляньте можливість використання контейнерних класів (таких як vector, valarray, set і map).
- Для розглянутих раніше задач запропонуйте механізм перехоплення й обробки можливих помилок.

Глава 9

Математичні задачі

- Послухай, а як же ми битися з ними будемо?

- Та вже якось будемо...

з х/ф "Пропала грамота"

У цій главі ми розглянемо деякі математичні задачі, які традиційно входять до університетського курсу з обчислювальних методів і передбачають написання спеціальних програм. Зокрема, ми зосередимо увагу на:

- розв'язанні алгебраїчних рівнянь;
- побудові інтерполяційних поліномів;
- обчисленні інтегралів;
- розв'язанні диференціальних рівнянь,

причому практично для всіх із перерахованих типів задач пропонується декілька способів розв'язання. Варто також зауважити, що ця глава є в певному сенсі особливою.

До уваги

По-перше, розглядаються виключно математичні задачі. А, по-друге, розбір програмних кодів вимагатиме від читача певної самостійності, оскільки коментарі в книзі будуть стосуватися насамперед математичного боку питання й загального алгоритму розв'язання задачі. Стосовно програмних кодів наводяться лише пояснення загального характеру. Такий підхід повинен сприяти кращому закріпленню розглянутого раніше матеріалу й дозволить перевірити отримані знання й навички.

Метод послідовних наближень

Під час розв'язання рівняння виду $x = \varphi(x)$ за деяких додаткових умов може використовуватися метод послідовних наближень. Суть методу зводиться до того, що задається початкове наближення x_0 для кореня рівняння, після чого,

використовуючи ітераційну процедуру, кожне наступне наближення обчислюється на основі попереднього відповідно до формули $x_{n+1} = \varphi(x_n)$. У лістингу 9.1 наведено програму, за допомогою якої описаним способом розв'язують алгебраїчні рівняння $x = 0.5 \cdot \cos(x)$, $x = \exp(-x)$ і $x = \frac{x^2+6}{5}$.

Лістинг 9.1. Метод послідовних наближень

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>
using namespace std;
// Функція для розв'язання рівняння методом
// послідовних наближень:
double findRoot(double (*f)(double),double x0,int n){
   // Початкове наближення для кореня рівняння:
   double x=x0;
   // Послідовні ітерації:
   for(int k=1;k<=n;k++){
      x=f(x);
   }
   // Результат функції:
   return x;
}
// Функції для визначення правої
// частини розв'язуваних рівнянь:
double f(double x){
   return 0.5*cos(x);
}
double g(double x){
   return exp(-x);
}
```

```
double h(double x){
    return (x*x+6)/5;
}
// Допоміжна функція для розв'язання рівнянь
// і тестування знайдених розв'язків:
void test(double (*f)(double),double x0,string eq){
    // Кількість ітерацій:
    int n=100;
    // Змінна для запису значення кореня рівняння:
    double z;
    cout<<"Розв'язок рівняння "<<eq<<":\t";
    // Розв'язання рівняння:
    z=findRoot(f,x0,n);
    // Відображення результату:
    cout<<z<<endl;
    cout<<"Перевірка знайденого розв'язку:\t";
    // Перевірка розв'язку:
    cout<<z<<" = "<<f(z)<<endl;
    for(int k=1;k<=50;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Приклади розв'язання рівнянь:
    test(f,0,"x=0.5cos(x)");
    test(g,0,"x=exp(-x)");
    test(h,1,"x=(x*x+6)/5");
    // Затримка консольного вікна:
```

```
    system("pause>nul");
    return 0;
}
```

Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 9.1)

```
Розв'язок рівняння x=0.5cos(x): 0.450184
Перевірка знайденого розв'язку: 0.450184 = 0.450184
-------------------------------------------------

Розв'язок рівняння x=exp(-x):   0.567143
Перевірка знайденого розв'язку: 0.567143 = 0.567143
-------------------------------------------------

Розв'язок рівняння x=(x*x+6)/5: 2
Перевірка знайденого розв'язку: 2 = 2
-------------------------------------------------
```

У програмі для розв'язання рівнянь описано функцію `findRoot()`. Перший аргумент функції - вказівник на функцію, що визначає праву частину розв'язуваного рівняння. Це повинна бути функція з одним `double`-аргументом, що повертає `double`-результат. Другий аргумент функції `findRoot()` визначає початкове наближення для кореня рівняння. Третій цілочисловий аргумент задає кількість виконуваних під час пошуку кореня ітерацій.

У програмі розв'язуються три рівняння. Праві частини рівнянь визначаються за допомогою функцій `f()`, `g()` і `h()`. Назви цих функцій передаються як перший аргумент під час виклику функції `findRoot()`. Ця функція, в свою чергу, викликається в тілі функції `test()`.

Після обчислення кореня рівняння виконується перевірка. З цією метою в консольне вікно виводиться обчислене значення для кореня рівняння (значення x) і значення у відповідній точці функції для правої частини рівняння (значення $\varphi(x)$). В ідеалі, якщо знайдено точний розв'язок, обидва значення повинні збігатися.

До уваги

Метод послідовних ітерацій дозволяє знайти розв'язок далеко не для кожного рівняння. Тут важливо все: і яка функція в правій частині рівняння, і яке початкове наближення обрано для пошуку кореня.

Метод половинного ділення

Метод половинного ділення для пошуку розв'язку алгебраїчного рівняння виду $f(x) = 0$ передбачає, що шуканий корінь локалізований на інтервалі $a \leq x \leq b$, причому на межах інтервалу пошуку кореня функція $f(x)$ повинна набувати значення різних знаків (що формально може бути записано у вигляді умови $f(a)f(b) < 0$).

Суть методу полягає в таких діях. Обчислюється значення функції $f(x)$ у центральній точці інтервалу пошуку розв'язку, яка визначається як $x = \frac{a+b}{2}$. Цікавість викликає знак функції в цій точці: у цю точку зсувається одна з меж інтервалу пошуку кореня. А саме, якщо в центральній точці інтервалу значення функції додатне, то в центр зсувається та межа, де функція додатна. Якщо в центральній точці функція від'ємна, то зсувається та межа, де функція від'ємна. У результаті після кожної такої операції інтервал пошуку кореня рівняння зменшується рівно вдвічі, і при цьому функція $f(x)$ на межах нового інтервалу пошуку кореня набуває значення різних знаків. Задача, таким чином, зводиться до попередньої.

Приклад програми, у якій пошук кореня рівняння виконується методом половинного ділення, наведено в лістингу 9.2.

Лістинг 9.2. Розв'язання рівняння методом половинного ділення

```
#include <iostream>
#include <cstdlib>
#include <cmath>
#include <string>
```

```
using namespace std;
// Функція для розв'язання рівняння методом
// половинного ділення:
double findRoot(double (*f)(double),double a,
double b,double dx){
    // Змінна для запису значення кореня:
    double x=(a+b)/2;
    // Обчислення кореня:
    while((b-a)/2>dx){
        // Якщо корінь потрапив на ліву межу:
        if(f(a)==0){
            return a;
        }
        // Якщо корінь потрапив на праву межу:
        if(f(b)==0){
            return b;
        }
        // Якщо корінь посередині інтервалу:
        if(f(x)==0){
            return x;
        }
        // Зміщення межі в центр інтервалу:
        if(f(a)*f(x)>0){
            a=x;
        }
        else{
            b=x;
        }
        // Нове значення для центральної точки інтервалу:
        x=(a+b)/2;
    }
    // Результат функції:
```

```
    return x;
}
// Функції для визначення розв'язуваних рівнянь:
double f(double x){
    return 0.5*cos(x)-x;
}
double g(double x){
    return exp(-x)-x;
}
double h(double x){
    return x*x-5*x+6;
}
// Допоміжна функція для розв'язання рівнянь
// і тестування знайдених розв'язків:
void test(double (*f)(double),double a,double b,string eq){
    // Контрольований код:
    try{
        // Якщо на межах інтервалу функція рівняння
        // набуває значення однакових знаків:
        if(f(a)*f(b)>0){
            // Генерування помилки з текстовим об'єктом:
            throw "Зазначено неправильний діапазон!";
        }
        // Точність обчислення кореня:
        double dx=0.001;
        // Змінна для запису значення кореня рівняння:
        double z;
        cout<<"Розв'язок рівняння "<<eq<<":\t";
        // Розв'язання рівняння:
        z=findRoot(f,a,b,dx);
        // Відображення результату:
```

```
        cout<<z<<endl;
        cout<<"Перевірка знайденого розв'язку:\t";
        // Перевірка розв'язку:
        cout<<f(z)<<" = 0"<<endl;
    }
    // Обробка помилки:
    catch(char* e){
        // Повідомлення про некоректний інтервал
        // для пошуку кореня рівняння:
        cout<<e<<endl;
    }
    for(int k=1;k<=50;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Приклади розв'язання рівнянь:
    test(f,0,1,"0.5cos(x)-x=0");
    test(g,0,2,"exp(-x)-x");
    test(h,0,5,"x*x-5*x+6=0");
    test(h,0,2,"x*x-5*x+6=0");
    test(h,1,3,"x*x-5*x+6=0");
    test(h,2.5,4.5,"x*x-5*x+6=0");
    test(h,2.5,10,"x*x-5*x+6=0");
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 9.2)

```
Розв'язок рівняння 0.5cos(x)-x=0:        0.450195
Перевірка знайденого розв'язку: -1.4247e-005 = 0
-------------------------------------------------
Розв'язок рівняння exp(-x)-x:    0.567383
Перевірка знайденого розв'язку: -0.000375349 = 0
-------------------------------------------------
Зазначено неправильний діапазон!
-------------------------------------------------
Розв'язок рівняння x*x-5*x+6=0: 2
Перевірка знайденого розв'язку: 0 = 0
-------------------------------------------------
Розв'язок рівняння x*x-5*x+6=0: 3
Перевірка знайденого розв'язку: 0 = 0
-------------------------------------------------
Розв'язок рівняння x*x-5*x+6=0: 3
Перевірка знайденого розв'язку: 0 = 0
-------------------------------------------------
Розв'язок рівняння x*x-5*x+6=0: 3.00079
Перевірка знайденого розв'язку: 0.000794087 = 0
-------------------------------------------------
```

Метод половинного ділення реалізовано в програмі через функцію `findRoot()`. У функції такі аргументи:

- вказівник на функцію, що визначає розв'язуване рівняння (для рівняння виду $f(x) = 0$ мається на увазі функція $f(x)$);
- ліва і права межі інтервалу, на якому виконується пошук кореня рівняння;
- параметр, що визначає точність, з якою обчислюється корінь.

До уваги

Якщо корінь рівняння локалізовано на інтервалі від a до b, то він знаходиться (зміщений) від центральної точки інтервалу на величину, не більшу ніж половина

довжини інтервалу. Цей критерій використано у функції `findRoot()` під час обчислення кореня рівняння. Обчислення триває доти, поки половина довжини інтервалу локалізації кореня не стане меншою за параметр, який передається як останній аргумент функції `findRoot()` і визначає точність обчислень.

Алгоритм виконання функції `findRoot()` такий:

- Послідовно перевіряють на предмет рівності нулю значення функції, переданої першим аргументом (функція рівняння), на межах інтервалу пошуку розв'язку і в його центрі. Якщо корінь рівняння потрапляє в одну із зазначених точок, відповідне значення повертається як результат функції `findRoot()`. Така перевірка виконується для того, щоб уникнути непотрібної ітераційної процедури у випадку, якщо чергова пробна точка для розв'язку рівняння випадково потрапила в точне значення для кореня.
- Якщо значення функції рівняння на межах інтервалу і в його центрі відмінні від нуля, то одна з меж зсувається в центр, і процес повторюється.

Функція для розв'язання рівнянь `findRoot()` викликається у функції `test()`. В останній перед початком процесу обчислень перевіряється, чи коректно вказано інтервал для пошуку кореня. Якщо виявляється, що функція рівняння на межах інтервалу має значення однакових знаків, то генерується помилка (об'єктом помилки є текстовий літерал). Під час перехоплення помилки виводиться повідомлення, що вказано некоректний інтервал.

До уваги

Функції `f()`, `g()` і `h()` у попередньому прикладі (лістинг 9.1) визначали праву частину для рівняння виду $x = \varphi(x)$. Тут же розв'язується рівняння виду $f(x) = 0$. Нескладно зрозуміти, що якщо рівняння $x = \varphi(x)$ записати у вигляді $\varphi(x) - x = 0$, то маємо справу з функцією $f(x) = \varphi(x) - x$. Простіше кажучи, тепер функції для тих самих розв'язуваних рівнянь визначено по-іншому.

У головній функції програми функція `test()` викликається з різними аргументами. Зокрема, там реалізуються ситуації, коли інтервал зазначено некоректно, коли корінь потрапляє в одну з граничних точок інтервалу або коли

через декілька ітерацій центральна точка інтервалу потрапляє в точний розв'язок для рівняння.

Метод дотичних

Ще один ітераційний метод розв'язання рівняння, який називається методом дотичних, передбачає реалізацію такої процедури:

- Для пошуку кореня рівняння $f(x) = 0$ задається початкове наближення x_0.
- У точці x_0 будується дотична до кривої, що визначається залежністю $f(x)$.
- Точка перетину дотичної з координатною віссю абсцис є новим наближенням для кореня рівняння x_1.
- У точці x_1 будується дотична, і її перетин із горизонтальною координатною віссю визначає нове наближення для кореня x_2 і так далі.

Описана схема зводиться до ітераційної формули $x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$ для обчислення нового наближення для кореня рівняння x_{n+1} за відомим попереднім наближенням x_n. Тут через $f'(x)$ позначено похідну від функції рівняння $f(x)$. У лістингу 9.3 до уваги читача пропонується програма, в якій реалізовано процес пошуку кореня рівняння методом дотичних.

Лістинг 9.3. Метод дотичних

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функція, що визначає розв'язуване рівняння:
double f(double x){
   return 2*exp(-x)-1;
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
```

```
    system("chcp 1251>nul");
    // Кількість ітерацій:
    int n=10;
    // Приріст аргументу для обчислення похідної:
    double dx=0.00001;
    // Початкове наближення для кореня рівняння:
    double x=0;
    // Обчислення кореня рівняння:
    for(int k=1;k<=n;k++){
       x=x-f(x)/((f(x+dx)-f(x))/dx);
    }
    // Відображення результату:
    cout<<"Знайдено такий розв'язок:\t"<<x<<endl;
    cout<<"Контрольне значення:\t"<<log(2)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче наведено результат виконання програми:

Результат виконання програми (із лістингу 9.3)

```
Знайдено такий розв'язок:       0.693147
Контрольне значення:    0.693147
```

Програма зовсім маленька і проста. У ній розв'язується рівняння $2\exp(-x) - 1 = 0$, розв'язком якого насправді є значення $x = \ln(2) \approx 0.693147$. Рівняння задається через функцію `f()`. Ітераційний процес реалізується безпосередньо в головній функції програми. Там запускається оператор циклу, в якому виконується всього `10` ітерацій (значення змінної `n`). Похідна, значення якої треба використовувати в рекурентній формулі для обчислення кореня рівняння, розраховується в наближеному вигляді $f'(x) \approx$

$\frac{f(x+dx)-f(x)}{dx}$ як відношення приросту функції до приросту аргументу. Величина приросту для аргументу в програмі задається змінною `dx`. Як початкове наближення для кореня рівняння використано нульове значення. У програмі окрім обчисленого кореня для порівняння наводиться й "точне" значення для розв'язку рівняння.

До уваги

Як і у випадку з методом послідовних ітерацій, збіжність методу дотичних вимагає окремого дослідження. Іншими словами, метод дотичних застосовний далеко не до кожного рівняння. Читачі, що цікавляться питанням застосовності цього й інших методів, можуть звернутися до спеціальної літератури.

Інтерполяційний поліном Лагранжа

Задача з побудови інтерполяційного полінома коротко може бути сформульована таким чином. Є набір вузлових точок x_0, x_1, x_2, ..., x_n (усього $n+1$ точка), і в цих точках задано значення y_0, y_1, y_2, ..., y_n деякої функції. Необхідно побудувати поліном степеня n (степінь полінома на одиницю менший за кількість точок) такий, щоб у вузлових точках $\{x_k\}$ поліном набував значення $\{y_k\}$ (індекс $k = 0,1,2,\dots,n$).

Розв'язувати задачу можна різними способами. Тут ми розглянемо метод побудови інтерполяційного полінома за схемою Лагранжа. У такому випадку шуканий поліном представляється у вигляді $L_n(x) = y_0\varphi_0(x) + y_1\varphi_1(x) + y_2\varphi_2(x) + \cdots + y_n\varphi_n(x)$. Ми ввели позначення для функцій $\varphi_k(x) = \frac{(x-x_0)(x-x_1)\dots(x-x_{k-1})(x-x_{k+1})\dots(x-x_n)}{(x_k-x_0)(x_k-x_1)\dots(x_k-x_{k-1})(x_k-x_{k+1})\dots(x_k-x_n)}$ (індекс $k = 0,1,2,\dots,n$). Легко перевірити, що $\varphi_k(x_m) = 0$, якщо $k \neq m$ і $\varphi_k(x_k) = 1$, тому умови $L_n(x_k) = y_k$, що накладаються на інтерполяційний поліном, задовольняються автоматично.

У програмі з лістингу 9.4 реалізовано описаний вище спосіб побудови інтерполяційного полінома на основі масиву зі значеннями вузлових точок і масиву зі значеннями функції у вузлових точках.

Лістинг 9.4. Інтерполяційний поліном Лагранжа

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Допоміжна функція:
double phi(int k,double z,double* x,int n){
   // Індексна змінна:
   int i;
   // Змінна для запису результату:
   double res=1;
   // Обчислення добутку:
   for(i=0;i<k;i++){
      res*=(z-x[i])/(x[k]-x[i]);
   }
   for(i=k+1;i<n;i++){
      res*=(z-x[i])/(x[k]-x[i]);
   }
   // Результат функції:
   return res;
}
// Поліном Лагранжа:
double L(double z,double* x,double* y,int n){
   // Змінна для запису результату:
   double s=0;
   // Обчислення поліноміальної суми:
   for(int k=0;k<n;k++){
      s+=y[k]*phi(k,z,x,n);
   }
   // Результат функції:
   return s;
}
```

```
// Функція для відображення "лінії":
void line(int m){
   for(int k=1;k<=m;k++){
      cout<<"-";
   }
   cout<<endl;
}
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Індексна змінна і довжина "лінії":
   int k,m=20;
   // Розмір масивів:
   const int n=5;
   // Вузлові точки:
   double x[n]={1,3,5,7,9};
   // Значення функції у вузлових точках:
   double y[n]={0,2,-1,1,3};
   line(m);
   cout<<"x\t| L(x)\n";
   line(m);
   // Відображення вузлових точок і значень
   // інтерполяційного полінома в цих точках:
   for(k=0;k<n;k++){
      cout<<x[k]<<"\t| "<<L(x[k],x,y,n)<<endl;
   }
   // Зсув для аргументу:
   double dx=1;
   line(m);
   cout<<"x\t| L(x)\n";
   line(m);
```

```
    // Значення аргументу і значення полінома:
    for(k=0;k<n;k++){
        cout<<x[k]+dx<<"\t| "<<L(x[k]+dx,x,y,n)<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Нижче показано, як виглядає результат виконання програми:

Результат виконання програми (із лістингу 9.4)

```
--------------------
x       | L(x)
--------------------
1       | 0
3       | 2
5       | -1
7       | 1
9       | 3
--------------------
x       | L(x)
--------------------
2       | 2.83594
4       | 0.148438
6       | -0.664063
8       | 2.89844
10      | -1.66406
```

Для реалізації функції $\varphi_k(x)$ у програмі описано функцію `phi()`. Перший аргумент функції відповідає індексу k. Другий аргумент функції `phi()` відповідає точці x, у якій обчислюється значення $\varphi_k(x)$. Третій аргумент функції `phi()` передає ім'я масиву, в якому записано значення вузлових точок.

Нарешті, останній аргумент функції `phi()` визначає розмір масиву вузлових точок. Як результат функції повертається добуток множників виду $\frac{x-x_i}{x_k-x_i}$, серед яких відсутній множник з індексом $i = k$.

> ***До уваги***
>
> Константа `n` у програмі, що визначає розмір масивів з вузловими точками і значеннями функції у вузлових точках, за своєю "природою" на одиницю менша за параметр n, який використовувався у формулах, що наводилися для інтерполяційного полінома.

Значення інтерполяційного полінома обчислює функція `L()`. Як аргументи функції передаються:

- точка, у якій слід обчислити значення полінома;
- масив із вузловими точками;
- масив зі значеннями функції у вузлових точках;
- цілочисловий аргумент, що визначає розмір масивів.

У тілі функції `L()` під час обчислення поліноміальної суми викликається функція `phi()`.

У головній функції програми як ілюстрація обчислюються значення інтерполяційного полінома у вузлових точках і в точках, зсунутих на одиничну відстань відносно вузлових точок.

Інтерполяційний поліном Ньютона

Під час створення інтерполяційного полінома методом Ньютона для полінома використовується такий вираз: $P_n(x) = a_0 + a_1(x - x_0) + a_2(x - x_0)(x - x_1) + a_3(x - x_0)(x - x_1)(x - x_2) + \cdots + a_n(x - x_0)(x - x_1) \ldots (x - x_{n-1})$. Задача, фактично, зводиться до обчислення коефіцієнтів a_k (індекс $k = 0,1,2, \ldots, n$). Коефіцієнти визначають, виходячи з накладених на поліном умов $P_n(x_k) = y_k$. Зокрема, зі співвідношення $P_n(x_0) = y_0$ виходить, що $a_0 = y_0$. Із умови $P_n(x_1) = y_1$ отримуємо $a_1 = \frac{y_1-a_0}{(x_1-x_0)}$. Умова $P_n(x_2) = y_2$ дає $a_2 = \frac{y_2-a_0-a_1(x_2-x_0)}{(x_2-x_0)(x_2-x_1)}$ і так далі. Рекурентне співвідношення для обчислення коефіцієнтів має вигляд

$a_k = \frac{y_k - a_0 - a_1(x_k - x_0) - a_2(x_k - x_0)(x_k - x_1) - \cdots - a_{k-1}(x_k - x_0)\ldots(x_k - x_{k-2})}{(x_k - x_0)(x_k - x_1)\ldots(x_k - x_{k-1})}$. Програму, у якій на основі масивів із вузловими точками і значеннями функції у вузлових точках обчислюється інтерполяційний поліном Ньютона, представлено в лістингу 9.5.

Лістинг 9.5. Інтерполяційний поліном Ньютона

```
#include <iostream>
#include <cstdlib>
using namespace std;
// Функція обчислення коефіцієнтів для полінома:
void findA(double* a,double* x,double* y,int n){
   // Змінна для запису добутку
   // різниць вузлових точок:
   double q;
   // Значення першого коефіцієнта для полінома:
   a[0]=y[0];
   // Обчислення інших коефіцієнтів для полінома:
   for(int k=1;k<n;k++){
      // Початкове значення для коефіцієнта:
      a[k]=y[k];
      // Початкове значення для добутку
      // різниць значень вузлових точок:
      q=1;
      // Уточнення значення коефіцієнта й обчислення
      // добутку різниць значень вузлових точок:
      for(int m=0;m<k;m++){
         a[k]-=a[m]*q;
         q*=(x[k]-x[m]);
      }
      // Кінцеве значення коефіцієнта:
      a[k]/=q;
   }
```

```
}
// Функція для обчислення значення полінома Ньютона:
double P(double* a,double z,double* x,int n){
    // Змінна для запису результату функції:
    double s=0;
    // Локальна змінна для запису добутку
    // різниць значень аргументу і вузлових точок:
    double q;
    // Обчислення поліноміальної суми:
    for(int k=0;k<n;k++){
        // Початкове значення для добутку:
        q=1;
        // Обчислення добутку:
        for(int m=0;m<k;m++){
            q*=(z-x[m]);
        }
        s+=a[k]*q;
    }
    // Результат функції:
    return s;
}
// Функція для відображення "лінії":
void line(int m){
    for(int k=1;k<=m;k++){
        cout<<"-";
    }
    cout<<endl;
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Індексна змінна і довжина "лінії":
```

```
    int k,m=20;
    // Розмір масивів:
    const int n=5;
    // Вузлові точки:
    double x[n]={1,3,5,7,9};
    // Значення функції у вузлових точках:
    double y[n]={0,2,-1,1,3};
    // Коефіцієнти для полінома Ньютона:
    double a[n];
    // Обчислення значень для коефіцієнтів:
    findA(a,x,y,n);
    line(m);
    cout<<"x\t| P(x)\n";
    line(m);
    // Відображення вузлових точок і значень
    // інтерполяційного полінома в цих точках:
    for(k=0;k<n;k++){
       cout<<x[k]<<"\t| "<<P(a,x[k],x,n)<<endl;
    }
    // Зсув для аргументу:
    double dx=1;
    line(m);
    cout<<"x\t| P(x)\n";
    line(m);
    // Значення аргументу й значення полінома:
    for(k=0;k<n;k++){
       cout<<x[k]+dx<<"\t| "<<P(a,x[k]+dx,x,n)<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 9.5)

```
--------------------
x        | P(x)
--------------------
1        | 0
3        | 2
5        | -1
7        | 1
9        | 3
--------------------
x        | P(x)
--------------------
2        | 2.83594
4        | 0.148438
6        | -0.664063
8        | 2.89844
10       | -1.66406
```

Нескладно помітити, що під час виконання цієї програми отримуємо такий самий результат (такі самі значення полінома), як і в попередньому прикладі. Нічого дивного в цьому немає, оскільки насправді в обох випадках створюється один і той самий поліном, просто ми використовували різні алгоритми його обчислення.

Щодо безпосередньо програми, то ми застосували загальний підхід, який полягає в тому, що на основі значень вузлових точок і значень інтерпольованої функції в цих вузлових точках обчислюються коефіцієнти для інтерполяційного полінома. Коефіцієнти записуються в масив, і цей масив потім використовується під час обчислення значення інтерполяційного полінома.

Коефіцієнти для полінома обчислюються під час виклику функції `findA()`. Функція не повертає результат, а як аргументи їй передаються:

- ім'я масиву, в який записуються значення коефіцієнтів інтерполяційного полінома;
- масив зі значеннями вузлових точок;
- масив зі значеннями інтерпольованої функції у вузлових точках;
- цілочислове значення, що визначає розмір масивів.

У тілі функції `findA()` із використанням ітераційної процедури обчислюються значення коефіцієнтів для полінома, і ці обчислені значення записуються в масив, переданий першим аргументом функції.

Функція `P()` призначена для обчислення значення інтерполяційного полінома в точці. Точка (аргумент полінома), у якій обчислюється значення, передається як другий аргумент функції `P()`. Перший її аргумент - масив із коефіцієнтами. Третій аргумент - масив зі значеннями вузлових точок, а четвертий аргумент функції `P()` задає розміри використовуваних масивів. Передавати у функцію `P()` масив зі значеннями інтерпольованої функції у вузлових точках у цьому випадку не потрібно, оскільки відповідний масив уже було "враховано" при обчисленні коефіцієнтів для інтерполяційного полінома.

Обчислення інтеграла методом Сімпсона

Далі розглянемо задачу про обчислення визначеного інтеграла виду $\int_a^b f(x)dx$. Існують різні способи її розв'язання (у числовому вигляді). Метод Сімпсона - один із найпростіших методів обчислення інтеграла. Коротко суть методу полягає в тому, що інтервал, на якому обчислюється інтеграл, розбивається на велику і, що важливо, парну, кількість інтервалів рівної довжини. Позначимо кількість інтервалів, на які розбивається інтервал інтегрування, як $2m$ (тоді довжина одного інтервалу дорівнює $h = \frac{b-a}{2m}$). Вузлові точки визначаються як $x_k = a + kh$, де індекс $k = 0,1,2,\dots,2m$. Зрозуміло, що $x_0 = a$ і $x_{2m} = b$ за означенням. Позначимо також як $f_k = f(x_k)$ значення

підінтегральної функції $f(x)$ у вузлових точках. Далі на кожних двох суміжних інтервалах (тобто по трьох сусідніх вузлових точках) будуються інтерполяційні поліноми (другого степеня). Перевага підходу в тому, що замість функції $f(x)$ під інтегралом опиняються відносно прості поліноміальні вирази, які можна проінтегрувати в явному вигляді. У підсумку для інтеграла отримуємо такий наближений вираз: $\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + f_{2m} + 4\sum_{k=1}^{m} f_{2k-1} + 2\sum_{k=1}^{m-1} f_{2k})$. Цю формулу, записану у вигляді $\int_a^b f(x)dx \approx \frac{h}{3}(f_0 + f_{2m} + 4f_{2m-1}) + \frac{h}{3}\sum_{k=1}^{m-1}(4f_{2k-1} + 2f_{2k})$, ми й використовуємо в програмі з лістингу 9.6 для обчислення визначених інтегралів.

Лістинг 9.6. Обчислення інтегралів методом Сімпсона

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функція для обчислення інтегралів методом Сімпсона:
double integrate(double (*f)(double),double a,double b,
int m=1000){
    // Довжина внутрішнього інтервалу:
    double h=(b-a)/2/m;
    // Змінна для запису інтегральної суми:
    double s=0;
    // Обчислення інтегральної суми:
    for(int k=1;k<=m-1;k++){
        s+=4*f(a+(2*k-1)*h)+2*f(a+2*k*h);
    }
    s+=f(a)+f(b)+4*f(a+(2*m-1)*h);
    s*=h/3;
    // Результат функції:
    return s;
```

```
}
// Підінтегральні функції:
double F1(double x){
    return x*(1-x);
}
double F2(double x){
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
double F3(double x){
    return exp(-x)*cos(x);
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    cout<<"Обчислення інтегралів\n";
    cout<<integrate(F1,0,1)<<" vs. "<<(double)1/6<<endl;
    cout<<integrate(F2,0,1)<<" vs. "<<log(2)<<endl;
    cout<<integrate(F3,0,100,1e5)<<" vs. "<<0.5<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми такий:

Результат виконання програми (із лістингу 9.1)

```
Обчислення інтегралів
0.166667 vs. 0.166667
0.693147 vs. 0.693147
0.5 vs. 0.5
```

У програмі для обчислення інтегралів описано функцію `integrate()`. Як перший аргумент їй передається вказівник на функцію, що визначає підінтегральний вираз. Два наступних аргументи визначають межі області інтегрування. Четвертий необов'язковий аргумент (у нього є значення за замовчуванням) визначає кількість інтервалів, на які розбивається область інтегрування (кількість таких інтервалів у два рази більша за значення четвертого аргументу).

У головній функції програми обчислюється три інтеграли: інтеграл $\int_0^1 x(1-x)dx = \frac{1}{6}$, інтеграл $\frac{\pi}{2}\int_0^1 tg\left(\frac{\pi x}{4}\right)dx = \ln(2)$, а також інтеграл $\int_0^\infty \exp(-x)\cos(x)dx = \frac{1}{2}$. В останньому випадку маємо справу з невласним інтегралом. На практиці ми верхню нескінченну межу замінюємо на велике, але скінчене значення 100 (при цьому функції `integrate()` у явному вигляді передається значення 10^5 для останнього аргументу). Підінтегральні функції, від яких обчислюються інтеграли, задаються за допомогою програмних функцій `F1()`, `F2()` і `F3()`. Для порівняння в програмі поряд з обчисленим значенням для інтеграла відображається і контрольне значення. Неважко помітити, що результати обчислень більш ніж прийнятні навіть для невласного інтеграла (хоча в загальному випадку для обчислення невласних інтегралів використовуються спеціально розроблені для цього методи).

Обчислення інтегралів методом Монте-Карло

Одразу варто зазначити, що обчислення інтегралів методом Монте-Карло не належить ні до точних, ні до швидких, хоча алгоритм не дуже складний. Зазвичай до методу Монте-Карло звертаються, коли інші методи непридатні.

Суть методу пояснимо на прикладі обчислення інтеграла $\int_a^b f(x)dx$ від функції $f(x)$, яка на інтервалі інтегрування невід'ємна й обмежена, так що має

місце співвідношення $0 \le f(x) \le f_{max}$, де через f_{max} позначена деяка оцінка для значення функції на інтервалі інтегрування. Якщо розглянути в координатній площині квадрат $a \le x \le b$ і $0 \le y \le f_{max}$, то площа такого квадрата дорівнює, очевидно, величині $(b-a)f_{max}$. Далі обираємо випадковим чином точку всередині цього квадрата. Теорія ймовірностей стверджує, що ймовірність P для випадково вибраної точки опинитися під кривою, що визначається функцією $f(x)$, дорівнює відношенню площі S області, обмеженої цією кривою, до площі прямокутника, тобто $P = \frac{S}{(b-a)f_{max}}$. Тоді $S = (b-a)f_{max}P$. З іншого боку, площа області під кривою і є інтегралом. Тому для обчислення інтеграла достатньо обчислити ймовірність P події, що полягає в тому, що випадково вибрана у прямокутнику точка опиниться під кривою $f(x)$. Якщо помножити таку ймовірність на площу квадрата, отримаємо шуканий інтеграл.

Для обчислення ймовірності P можна послідовно генерувати випадкові точки і перевіряти, чи потрапляють вони під криву. Відношення числа "внутрішніх" (що потрапили під криву) точок до загального числа точок дає оцінку для ймовірності P. При цьому важливо, щоб точок було багато, а розподілялися вони по області квадрата рівномірно. Тому суттєву роль відіграє генератор випадкових чисел.

На практиці замість генерування випадкових точок допускається скористатися сіткою з рівномірно розподілених точок. В іншому алгоритм залишається незмінним: підраховуються точки під кривою, обчислюється відношення кількості таких точок до загальної кількості точок в сітці, результат множиться на площу прямокутника, і це буде відповідь. Програму з реалізацією такого підходу представлено в лістингу 9.7.

Лістинг 9.7. Обчислення інтегралів методом Монте-Карло

```
#include <iostream>
#include <cstdlib>
```

```
#include <cmath>
using namespace std;
// Функція для обчислення інтегралів методом Монте-Карло:
double integrate(double (*f)(double),double a,
double b,double Fmax){
    // Змінні для запису координат точки:
    double x,y;
    // Кількість інтервалів по кожній із осей:
    int m=10000;
    // Приріст по першій координаті:
    double dx=(b-a)/m;
    // Приріст по другій координаті:
    double dy=Fmax/m;
    // Змінна для підрахунку внутрішніх точок:
    int count=0;
    // Перебирання точок на площині:
    for(int i=0;i<=m;i++){
       for(int j=0;j<=m;j++){
          // Перша координата точки:
          x=a+i*dx;
          // Друга координата точки:
          y=j*dy;
          // Перевірка точки:
          if(y<=f(x)){
             // Якщо точка під графіком функції:
             count++;
          }
       }
    }
    // Частка внутрішніх точок:
    double z=(double)count/(m+1)/(m+1);
    // Результат функції:
```

```
    return Fmax*(b-a)*z;
}
// Підінтегральні функції:
double F1(double x){
    return x*(1-x);
}
double F2(double x){
    double pi=3.141592;
    return pi/2*tan(pi*x/4);
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    cout<<"Обчислення інтегралів\n";
    cout<<integrate(F1,0,1,0.25)<<" vs. "
<<(double)1/6<<endl;
    cout<<integrate(F2,0,1,1.6)<<" vs. "<<log(2)<<endl;
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

У функції `integrate()` чотири аргументи: вказівник на підінтегральну функцію, нижня та верхня межі для інтервалу інтегрування, а також числова оцінка для максимального значення підінтегральної функції на інтервалі інтегрування. Результат виконання програми наведено нижче:

Результат виконання програми (із лістингу 9.7)

```
Обчислення інтегралів
0.166646 vs. 0.166667
0.693167 vs. 0.693147
```

Хоча результат обчислень непоганий, але його отримання забирає чимало часу. Отже, цей метод краще приберегти для особливих випадків.

Розв'язання диференціального рівняння методом Ейлера

Розв'яжемо диференційне рівняння виду $y'(x) = f(x, y(x))$ із початковою умовою $y(x_0) = y_0$ (у такій постановці йдеться про задачу Коші). Тут функцію двох аргументів $f(x, y)$ задано, через x позначено незалежний аргумент, $y(x)$ є шуканою невідомою функцією, а через $y'(x)$ позначено похідну від функції $y(x)$. Завдання полягає в тому, щоб знайти таку функцію $y(x)$, яка б задовольняла початкову умову й, будучи підставленою у рівняння $y'(x) = f(x, y(x))$, перетворювала б його на тотожність.

Найпростіший (але не найнадійніший у плані точності) метод числового розв'язання диференціального рівняння полягає в тому, що значення функції $y(x)$ обчислюється у вузлових точках $x_k = x_0 + kh$, де індекс $k = 0,1,2, \ldots$, крок приросту по аргументу h за можливістю повинен бути мінімальним, а через x_0 позначено точку, в якій задано початкову умову.

Розв'язання задачі базується на тому, що для похідної від функції $y(x)$ у вузловій точці x_k використовується наближений вираз $y'(x_k) \approx \frac{y(x_{k+1}) - y(x_k)}{h}$. У межах такого підходу легко отримуємо рекурентне співвідношення $y(x_{k+1}) = y(x_k) + hf(x_k, y(x_k))$, що дозволяє отримати значення функції $y(x)$ у точці x_{k+1}, якщо відомо значення функції у вузловій точці x_k. Відправною є точка x_0, у якій задано значення функції y_0.

У лістингу 9.8 представлено програмо, у якій методом Ейлера розв'язується диференціальне рівняння $y'(x) = x^2 \exp(-x) - y(x)$ із початковою умовою $y(0) = 1$. У цієї задачі є точний (аналітичний) розв'язок $y(x) = \left(\frac{x^3}{3} + 1\right) \exp(-x)$. У програмі числовий розв'язок порівнюється з аналітичним розв'язком.

Лістинг 9.8. Розв'язання диференціального рівняння методом Ейлера

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функція, що визначає диференціальне рівняння:
double f(double x,double y){
    return x*x*exp(-x)-y;
}
// Функція для розв'язання диференціального
// рівняння методом Ейлера:
double dsolve(double (*f)(double,double),double x0,
double y0,double x){
    // Кількість інтервалів:
    int n=1000;
    // Приріст по аргументу:
    double h=(x-x0)/n;
    // Початкове значення шуканої функції:
    double y=y0;
    // Обчислення значення функції в точці:
    for(int k=0;k<n;k++){
        y=y+h*f(x0+k*h,y);
    }
    // Значення шуканої функції в точці:
    return y;
}
// Функція, що визначає точний розв'язок
// диференціального рівняння:
double Y(double x){
```

```
    return (x*x*x/3+1)*exp(-x);
}
// Головна функція програми:
int main(){
    // Зміна кодування консолі:
    system("chcp 1251>nul");
    // Масив значень аргументу, для яких
    // обчислюється розв'язок диференціального рівняння:
    double x[]={0,0.5,1,3,10};
    cout<<"Розв'язок диференціального рівняння:\n";
    for(int k=0;k<5;k++){
        cout<<dsolve(f,0,1,x[k])<<" vs. "<<Y(x[k])<<endl;
    }
    // Затримка консольного вікна:
    system("pause>nul");
    return 0;
}
```

Результат виконання програми буде такий:

Результат виконання програми (із лістингу 9.8)

```
Розв'язок диференціального рівняння:
1 vs. 1
0.631701 vs. 0.631803
0.490245 vs. 0.490506
0.497815 vs. 0.497871
0.015116 vs. 0.0151787
```

Пошук розв'язку диференціального рівняння виконується за допомогою функції `dsolve()`. Як аргумент функції передається вказівник на функцію, що визначає диференціальне рівняння, початкову точку по аргументу і значення

шуканої функції в початковій точці, а також значення аргументу, для якого обчислюється значення шуканої функції.

До уваги

У тілі функції `dsolve()` діапазон від початкової точки `x0` до точки `x` розбивається на `n` інтервалів. Тому відстань між сусідніми вузловими точками змінюється зі зміною аргументу `x`. Чим більша відстань між вузловими точками, тим нижча точність обчислень.

Розв'язання диференціального рівняння методом Рунге-Кутти

Порівняно з методом Ейлера, метод Рунге-Кутти для розв'язання диференціальних рівнянь має більшу надійність. Так, у рамках методу Рунге-Кутти четвертого порядку значення шуканої функції $y(x_{k+1})$ у вузловій точці x_{k+1} обчислюється за формулою $y(x_{k+1}) = y(x_k) + \frac{h}{6}(p_1(h) + 2p_2(h) + 2p_3(h) + p_4(h))$, де уведено такі позначення: $p_1(h) = f(x_k, y(x_k))$, $p_2(h) = f\left(x_k + \frac{h}{2}, y(x_k) + \frac{h}{2}p_1(h)\right)$, $p_3(h) = f\left(x_k + \frac{h}{2}, y(x_k) + \frac{h}{2}p_2(h)\right)$ і $p_4(h) = f(x_k + h, y(x_k) + hp_3(h))$. Реалізацію цієї схеми представлено в програмному коді в лістингу 9.9. Код витримано в дусі попереднього прикладу, лише під час обчислення розв'язку використано більш "витончений" базовий алгоритм.

Лістинг 9.9. Розв'язання диференціального рівняння методом Рунге-Кутти

```
#include <iostream>
#include <cstdlib>
#include <cmath>
using namespace std;
// Функція, що визначає диференціальне рівняння:
double f(double x,double y){
```

```
    return x*x*exp(-x)-y;
  }
  // Функція для розв'язання диференціального
  // рівняння методом Рунге-Кутти:
  double dsolve(double (*f)(double,double),double x0,
double y0,double x){
    // Кількість інтервалів:
    int n=1000;
    // Змінні для запису проміжних
    // значень під час обчислень:
    double p1,p2,p3,p4;
    // Приріст по аргументу:
    double h=(x-x0)/n;
    // Початкове значення шуканої функції:
    double y=y0;
    // Обчислення значення функції в точці:
    for(int k=0;k<n;k++){
      p1=f(x0+k*h,y);
      p2=f(x0+k*h+h/2,y+h*p1/2);
      p3=f(x0+k*h+h/2,y+h*p2/2);
      p4=f(x0+k*h+h,y+h*p3);
      y=y+(h/6)*(p1+2*p2+2*p3+p4);
    }
    // Значення шуканої функції в точці:
    return y;
  }
  // Функція, що визначає точний розв'язок
  // диференціального рівняння:
  double Y(double x){
    return (x*x*x/3+1)*exp(-x);
  }
```

```
// Головна функція програми:
int main(){
   // Зміна кодування консолі:
   system("chcp 1251>nul");
   // Масив значень аргументу, для яких
   // обчислюється розв'язок диференціального рівняння:
   double x[]={0,0.5,1,3,10};
   cout<<"Розв'язок диференціального рівняння:\n";
   for(int k=0;k<5;k++){
      cout<<dsolve(f,0,1,x[k])<<" vs. "<<Y(x[k])<<endl;
   }
   // Затримка консольного вікна:
   system("pause>nul");
   return 0;
}
```

Нижче можна бачити результат виконання програми:

Результат виконання програми (із лістингу 9.9)

```
Розв'язок диференціального рівняння:
1 vs. 1
0.631803 vs. 0.631803
0.490506 vs. 0.490506
0.497871 vs. 0.497871
0.0151787 vs. 0.0151787
```

Бачимо, що точність обчислень більш ніж прийнятна.

Прикінцеві зауваження

Для цієї глави задачі для самостійного розв'язання не пропонуються в силу двох причин. По-перше, увесь матеріал глави у відомому сенсі розрахований на значний обсяг самостійної роботи. По-друге, усе-таки задачі специфічні й

передбачають для написання власного програмного коду наявність певного багажу математичних знань. Одна справа - аналізувати вже готовий програмний код, і зовсім інша - створювати програму для розв'язання математичної задачі "з нуля". З іншого боку, ті читачі, для кого цей клас задач видається цікавим, поза усяким сумнівом, легко знайдуть схожі (чи не дуже схожі) задачі для самостійного розв'язання. Єдине, що хочеться порадити на закінчення глави - подумати над тим, якими іншими способами можна було б реалізувати програми, наведені у цій главі. І вже зовсім було б добре такі програми створити і довести до "робочого" стану.

Замість епілогу

Корисні поради

Твоя служба - вірний кінь та чисте поле.
Людям служи!
з х/ф "Пропала грамота"

Книгу закінчено, приклади і задачі розібрано. Разом із тим, для більшості читачів усе це - тільки початок більшого (і, хочеться вірити, щасливого) шляху у вивченні мови C++. Тому розумно дати деякі поради "на майбутнє", які, можливо, допоможуть уникнути помилок і зберегти бажання рухатися далі в опануванні принципів C++.

- Немає універсальних книг, немає універсальних довідників, немає вичерпних ресурсів. Завжди залишається нехай найменша, але "незакрита" тема. У цьому немає нічого поганого. Навіть навпаки: наявність питань і сумнівів - хороший стимул для саморозвитку.

- Під час вивчення програмування другорядних питань не буває. Аналіз "дрібних", на перший погляд, незначних ефектів часом дає для розуміння загальної концепції набагато більше, ніж прочитання наймасивнішого довідника.

- Немає універсальних способів реалізації програми. Одну й ту саму задачу можна, як правило, розв'язувати різними способами. Для вибору найефективнішого підходу важливо розуміти хто, коли і як буде використовувати програму.

- Програмні технології розвиваються бурхливо, а переваги кінцевих користувачів програмних продуктів змінюються швидко. Потрібно спробувати передбачати тенденції. Без цього важко втриматися на ринку програмних послуг.

- На останок: мова програмування C++ потужна, гнучка, ефективна. Однак, це не єдина гарна мова програмування.

Отже, удачі! Ну і, звичайно, величезна подяка за увагу до книги ☺

Зміст

Вступ 3

Глава 1. Прості програми 7
Перша програма 7
Знайомство зі змінними 12
Знайомство з функціями 19
Знайомство з оператором циклу 22
Знайомство з умовним оператором 26
Знайомство з масивами 29
Задачі для самостійного розв'язування 31

Глава 2. Керуючі інструкції 34
Оператор циклу for 34
Оператор циклу do-while 40
Оператор вибору switch 43
Вкладені умовні оператори 50
Вкладені оператори циклу 52
Цикл по колекції 56
Генерування й перехоплення виключень 59
Інструкція безумовного переходу 65
Задачі для самостійного розв'язування 67

Глава 3. Вказівники, масиви та посилання 70
Знайомство з вказівниками 70
Масиви та вказівники 73
Знайомство з посиланнями 77
Динамічне виділення пам'яті 80
Особливості символьних масивів 83
Двомірні масиви 89
Масиви вказівників 96
Задачі для самостійного розв'язання 103

Глава 4. Функції 106
Оголошення й опис функції 106
Перевантаження функцій 111
Значення аргументів за замовчуванням 116
Рекурсія 118
Механізми передачі аргументів функціям 121
Передача вказівника як аргументу функції 126
Передача масиву як аргументу функції 129
Передача тексту у функцію 136

Вказівник як результат функції 139
Посилання як результат функції 142
Динамічний масив як результат функції 146
Вказівник на функцію 152
Задачі для самостійного розв'язання 159

Глава 5. Класи й об'єкти 163
Знайомство з класами й об'єктами 163
Відкриті та закриті члени класу 168
Перевантаження методів 171
Знайомство з конструкторами і деструкторами 178
Перевантаження операторів 186
Знайомство з успадкуванням 198
Задачі для самостійного розв'язання 206
Рекомендації для самостійної роботи 208

Глава 6. Використання класів і об'єктів 209
Вказівник на об'єкт 209
Створення масиву об'єктів 219
Масив як поле класу 223
Функтори й індексація об'єктів 228
Конструктор створення копії 232
Успадкування та закриті поля базового класу 237
Віртуальні методи та успадкування 241
Множинне успадкування 244
Доступ до об'єктів через змінну базового класу 248
Задачі для самостійного розв'язання 252
Рекомендації для самостійної роботи 253

Глава 7. Узагальнені функції та класи 254
Узагальнені функції 254
Узагальнена функція з декількома параметрами 259
Перевантаження узагальненої функції 262
Явна спеціалізація узагальненої функції 264
Узагальнені класи 267
Явна спеціалізація узагальненого класу 270
Значення параметрів за замовчуванням 275
Успадкування узагальнених класів 278
Цілочислові узагальнені параметри 284
Рекомендації для самостійної роботи 296

Глава 8. Різні задачі 297
Знайомство зі структурами 297
Узагальнені структури 301

Робота з комплексними числами 303
Клас для реалізації числових масивів 307
Контейнер для динамічного масиву 319
Контейнерний клас для реалізації множини 326
Асоціативний контейнер 329
Обробка помилок 333
Знайомство з багатопоточним програмуванням 336
Рекомендації для самостійної роботи 342

Глава 9. Математичні задачі 343
Метод послідовних наближень 343
Метод половинного ділення 347
Метод дотичних 353
Інтерполяційний поліном Лагранжа 355
Інтерполяційний поліном Ньютона 359
Обчислення інтеграла методом Сімпсона 364
Обчислення інтегралів методом Монте-Карло 367
Розв'язання диференціального рівняння методом Ейлера 371
Розв'язання диференціального рівняння методом Рунге-Кутти 374
Прикінцеві зауваження 376

Замість епілогу. Корисні поради 378

Навчальне видання

Олексій Васильєв

Програмування на C++

в прикладах і задачах

Керівник видавничого проекту *Зарицький В.І.*
Авторська редакція

Підписано до друку 14.06.2017. Формат 70×100 $^{1}/_{16}$.
Папір офсетний. Друк офсетний. Гарнітура Times New Roman.
Умовн. друк. аркушів – 31,04. Обл.-вид. аркушів – 29,93.
Тираж 300.

«Видавництво Ліра-К»
Свідоцтво № 3981, серія ДК.
03115, м. Київ, вул. Ф. Пушиної, 27, оф. 20-22
тел./факс (044) 247-93-37; 228-81-12
Сайт: lira-k.com.ua, редакція: zv_lira@ukr.net

```
#include <iostream>
#include <cstdlib>
using namespace std;
int main(){
// Зміна кодування консолі:
system("chcp 1251>nul");
// Відображення повідомлення в консолі:
cout<<"Програмуємо на C++! Олексій Васильєв "<<endl;
// Затримка консольного вікна:
system("pause>nul");
return 0;
}
```

ISBN 978-617-7507-41-2



