Міністерство освіти і науки України
Чорноморський національний університет імені Петра Могили

С. Ю. Боровльова,
А. В. Швед

БАЗОВИЙ С++

Навчальний посібник

Миколаїв – 2017

УДК 004.432(075.8)
Б 83

Рекомендовано до друку вченою радою Чорноморського національного університету ім. Петра Могили (протокол № 3 від 22.11.2016).

Рецензенти:
Кошкін К. В. – д-р техн. наук, професор, завідувач кафедри інформаційних управляючих систем та технологій, директор Інституту комп'ютерних наук Національного університету кораблебудування ім. адм. Макарова;
Рудницький В. М. – д-р техн. наук, професор, завідувач кафедри системного програмування Черкаського державного технологічного університету;
Устенко С. А. – д-р техн. наук, професор, завідувач кафедри комп'ютерної інженерії Миколаївського національного університету ім. В. О. Сухомлинського.

Б 83

Боровльова С. Ю. Базовий C++ : навчальний посібник / С. Ю. Боровльова, А. В. Швед. – Миколаїв : Вид-во ЧНУ ім. Петра Могили, 2017. – 116 с.

ISBN 978-966-336-370-7

У посібнику розглянуті базові можливості мови C++. Велику увагу приділено роботі із вказівниками, способам передачі параметрів у функцію, роботі з масивами та структурам. Посібник призначено для студентів спеціальностей "Комп'ютерні науки та інформаційні технології", "Інженерія програмного забезпечення" та "Комп'ютерна інженерія", а також може бути корисний студентам інших напрямів підготовки галузі знань 12 "Інформаційні технології".

УДК 004.432(075.8)

ISBN 978-966-336-370-7

З М І С Т

ВСТУП

Нині мова програмування С++ є однією з найбільш затребуваних мов програмування. Мова С++, розроблена Б'ярном Страуструпом у 1979 році, продовжує своє еволюціонування і дотепер, доповнюючись новими функціональними можливостями.

С++ є універсальною мовою загального призначення, що підтримує процедурну, функціональну, об'єктно-орієнтовану парадигми програмування.

Метою навчального посібника є спроба подання матеріалу з основ програмування мовою С++ у зрозумілій, простій формі, та формування у читачів базових навиків програмування. Кожне розглянуте питання супроводжується прикладами практичної реалізації. Кожен розділ навчального посібника закінчується переліком питань та завдань для самостійного виконання з метою перевірки викладеного матеріалу.

Тематично навчальний посібник складається з трьох розділів.

У першому розділі розглядаються основи мови С++: основні елементи мови; типи даних; оператори та операції мови.

Багато уваги приділено функціям, розглянуті питання виклику функцій зі змінним числом аргументів; типи аргументів функцій; способи передачі параметрів. У розділі розглядається новий стиль заголовків; класи пам'яті; простори імен. Описуються та наводяться приклади вбудованих (inline), рекурсивних та математичних функцій.

Розглянуті питання роботи із вказівниками, арифметика вказівників.

Багато уваги приділено роботі з масивами. Наведені типові алгоритми роботи з лінійними масивами: підрахунок суми/добутку елементів; підрахунок кількості елементів, що задовольняють умові; визначення кількості входжень заданого елементу до масиву; визначення максимального/мінімального елементів масиву. Розглядаються алгоритми роботи з багатомірними масивами: обробка елементів визначеного рядка/стовпчика; модифікація елементів відносно головної/побічної діагоналі.

Розглянуті питання роботи з рядками, структурами та об'єднаннями. Наводяться приклади типових функцій обробки рядків.

Другий розділ присвячено огляду бібліотечних функцій вводу-виводу. Приділена увага особливостям застосування функцій форма-

тованого введення-виведення для даних різних типів та формування управляючого рядка при форматуванні даних.

У третьому розділі розглянуті типові функції роботи з файлами послідовного та прямого доступу. Наведені приклади роботи з текстовими, двійковими та структурованими файлами.

Невеликий обсяг навчального посібника не дозволяє охопити весь спектр матеріалу з програмування мовою С++, однак автори висловлюють упевненість в тому, що наведений матеріал знайде своє місце в навчальних дисциплінах "Алгоритмічні мови та програмування", "Основи ООП", "Алгоритми та структури даних" тощо.

Посібник призначено для студентів спеціальностей "Комп'ютерні науки та інформаційні технології", "Інженерія програмного забезпечення" та "Комп'ютерна інженерія", а також може бути корисний студентам інших напрямів підготовки галузі знань 12 "Інформаційні технології".

РОЗДІЛ 1
Основи C++

1.1 Типи даних

1.1.1 Змінні і типи даних

У мові C++ передбачені вбудовані дані наступних типів:

– *Цілий* (**int**) – цілочисельні змінні (типу **int**, **long**, **short**) зберігають цілі значення, і можуть бути знаковими та беззнаковими. Знакові змінні можуть представляти як позитивні, так і негативні числа. Для цього в їх представленні один біт (найстарший) виділяється під знак. На відміну від них, беззнакові змінні містять тільки позитивні значення. Щоб вказати, що змінна буде беззнаковою, використовується ключове слово **unsigned**. За замовчуванням цілочисельні змінні вважаються знаковими (**signed**, найчастіше опускається; використовується при перетворенні типів даних).

– *Дійсний* (**float** та **double**) – для представлення чисел із плаваючою крапкою застосовують тип даних **float**. Цей тип, як правило, використовується для зберігання не дуже великих чисел. Якщо число може приймати більші значення, використовують змінні подвійної точності, тип **double**.

– *Булевий* (**bool**) – займає всього 1 байт і використовується, насамперед, у логічних операціях, тому що може приймати значення 0 (***false***, хибність) або відмінне від нуля (***true***, істина).

– *Символьний* (**char**) – (окремий випадок **int**) застосовується, коли змінна повинна нести інформацію про ASCII код. Цей тип даних часто використовується для побудови більш складних конструкцій, таких, як рядок, символьні масиви і т. п. Дані типу **char** також можуть бути знаковими й беззнаковими.

– *Відсутність типу* (**void**) – змінна типу **void** не має значення й служить для узгодження синтаксису. Наприклад, синтаксис вимагає, щоб функція повертала значення. Якщо не потрібно використовувати повернуте значення, перед іменем функції ставиться тип **void**.

– У стандарті C++11 для оголошення типу даних дозволено використовувати ключове слово **auto**, якщо компілятор може визначити тип змінної, виходячи із правої частини виразу.

Приклад:

```
auto i = 0;
```

Оскільки праворуч від знака присвоєння стоїть ціле число, компілятор може автоматично визначити тип змінної. Якщо замінити

оператор на **auto i**; то буде видана помилка компіляції, бо неможливо визначити тип змінної і.

Змінні

Правила побудови:

– Першим символом змінної С++ може бути тільки літера.

– Наступними символами ідентифікатора можуть бути літери, літери-цифри і літери-підкреслення.

– Довжина ідентифікатора необмежена (фактично довжина залежить від реалізації системи програмування).

Змінна – об'єкт програми, що займає в загальному випадку кілька байт пам'яті, покликаний зберігати дані. Щоб змінну можна було використовувати в програмі, вона повинна бути попередньо оголошена. При оголошенні змінної для неї резервується деяка область пам'яті, розмір якої залежить від конкретного типу змінної. Розмір того самого типу даних може відрізнятися на комп'ютерах різних платформ, а також може залежати від використовуваної операційної системи. Тому при оголошенні тієї або інший змінної потрібно чітко представляти, скільки байт вона буде займати в пам'яті, щоб уникнути проблем, пов'язаних з переповненням і неправильною інтерпретацією даних.

Приклад оголошення змінних:

```
int a, b = 0;
float age;
double ab = 55.4;
bool tipb = false;
char letter = 'Z';
```

1.1.2 Константи

Константи, так само як і змінні, являють собою область пам'яті для зберігання даних з тією лише відмінністю, що значення, присвоєне константі спочатку, не може бути змінене протягом виконання всієї програми. Константи бувають літеральними і типізованими, причому літеральні константи діляться на: символьні, строкові, цілі і дійсні.

– *Символьні константи* представляються окремим символом, укладеним в одинарні лапки (апострофи): 'e', '@', '<'.

– *Рядкові константи* – це послідовність символів, укладена в подвійні лапки: "*Це приклад не самої довгої строкової константи!*".

– *Цілі константи* бувають наступних форматів: десяткові; восьмирічні; шістнадцятирічні.

– *Десяткові* можуть бути представлені як послідовність цифр (від 0 до 9), що починається не з нуля, наприклад: 123; 2384.

– *Восьмирічні константи* – послідовність восьмирічних цифр (від 0 до 7), що починається з нуля, наприклад: 034; 047.

– *Шістнадцятирічний* формат констант починається із символів 0x або 0X з наступними шістнадцятирічними цифрами (0...9, A...F), наприклад: 0xf4; 0X5D. Літери при цьому можуть бути представлені в будь-якому регістрі.

– *Довгі цілі константи*, використовувані в змінних типу **long**, визначаються літерою **l** або **L** відразу після константи без проміжку: 36L, 012L, 0x52L.

– *Дійсні константи* – числа із плаваючої точкою можуть бути записані в десятковому форматі (24.58; 13.0; .71) або в експонентній формі.

Приклад:

```
10 в ступені 4 записується як 1e4;
5 в ступені 2, як 5e+2 або 5e2;
2.2 в ступені -5 буде 2.2e-5;
```

При цьому в мантисі може пропускатися ціла або дробова частина.

Приклад:

```
0.2 в ступені 4 можна записати як .2e4.
```

Усі наведені константи є типу **double**. Для запису констати типу **float** треба безпосередньо після константи вказати літеру **f** або **F.**

– *Типізовані константи* використовуються як змінні, значення яких не може бути змінене після ініціалізації. Типізована константа оголошується за допомогою ключового слова **const**, за яким іде вказівка типу константи, але, на відміну від змінних, константи завжди повинні бути ініціалізовані.

Синтаксис оголошення константи має вигляд:

const <тип> <ім'я константи> =<значення> [, < ім'я константи > = <значення>];

– *Символьні константи* в С++ займають у пам'яті 1 байт і, отже, можуть приймати значення від 0 до 255. При цьому існує ряд символів, які не відображаються під час друку, – вони виконують спеціальні дії: повернення каретки, табуляція і т. п.

Такі символи називаються символами *escape*-послідовності. Термін "*escape* послідовність" ввела компанія Epson, що стала першою фірмою, яка для керування виводом інформації на своїх принтерах стала використовувати невідображувані символи. Історично склалося так, що керуючі послідовності починалися з коду з десятковим значенням 27 (0x1B), що відповідало символу "*Escape*" кодування ASCII.

Escape – символи в програмі зображуються у вигляді зворотного слеша, за яким іде літера або символ та займають у пам'яті 1 байт.

Наприклад: '\n'.

Як приклад використання типізованих і літеральних констант обчислимо значення площі кола за відомим значенням радіуса.

Приклад:

```
#include <iostream>
using namespace std;
int main ()
{
   const double PI = 3.1415;
   const int radius = 3;
   double square = 0.0;
   square = PI * radius * radius;
   cout << square << ' \n ';
   return 0;
}
```

На початку головної функції програми оголошуються дві константи: PI і radius. Значення змінної square змінюється в ході виконання програми й не може бути представлене як константа. Оскільки значення радіуса задане явно й у тексті програми не передбачена його зміна, змінна radius оголошена як константа.

Примітка. Оператор **cout** – об'єкт вихідного потоку простору імен **std**::

```
std::cout << дані;
```

Для більш скороченої форми запису використовують запис виду:

```
using namespace std;
```

після чого застосовувати оператор можна наступним чином:

```
cout << дані;
```

Вираз **using namespace std**; відкриває простір імен бібліотеки **iostream**. У якості даних можуть бути зазначені літерали, змінні, елементи масивів, результати обчислення функцій і т. п.

Приклад:

```
#include <iostream>
using namespace std;
int main ()
{
int x = 4;  double y = 0.5;
// Виведення значення змінної x в стандартний потік cout
cout << x;
//  Функція endl здійснює перехід на новий рядок
cout << "x = " << x << "\ny = " << y << endl;
   return 0;
}
```

Константи можна визначати так само за допомогою директиви #**define**:

```
#define const_i        1            //прийшло з мови С
```

У цьому випадку константа не має ніякого типу. У момент роботи препроцесора всі const_i будуть замінені на 1. Компілятор взагалі не побачить константу const_i. Не рекомендується використовувати цю можливість у мові С++.

1.1.3 Перелічення

Перелічення дозволяють створювати нові типи даних, а потім визначати змінні цих типів. Змінні перелічення мають тип **unsigned int**.

Переліченням зручно користуватися при наявності великої кількості логічно взаємозалежних констант.

Синтаксис оголошення перелічення:

enum Name {item1[=def], item2[=def], ... itemn[=def]};

де **enum** – ключове слово (***enumerate*** – перераховувати),

Name – ім'я списку констант,

item1... **itemn** – перелік цілочисельних констант, [=def] – необов'язковий параметр ініціалізації.

Припустимо, необхідно в програмі описати роботу світлофора. Відомо, що його колір може приймати лише 3 значення: червоний (RED), жовтий (YELLOW) і зелений (GREEN). Для обробки отриманих від світлофора сигналів заведемо три константи з такими ж іменами – RED, YELLOW і GREEN, проініціалізувавши їх будь-якими неповторюваними значеннями для того, щоб надалі перевіряти, який із цих трьох кольорів горить.

Приклад:

```
const int RED = 0;
const int YELLOW = 1;
const int GREEN = 2;
```

Використовуючи перелічення, те ж саме можна зробити в один рядок:

```
enum COLOR { RED, YELLOW, GREEN };
```

Якщо значення константи перелічення не зазначене, воно на одиницю більше значення попередньої константи. За замовчуванням перша константа має значення 0.

Те ж перелічення можна було проініціалізувати іншими значеннями:

```
enum COLOR { RED=13, YELLOW=1, GREEN };
```

При цьому константа GREEN як і раніше має значення 2.

Приклад:

```
enum day_week {Sunday,Monday};
day_week day = Sunday;
```

1.1.4 Перетворення типів

У C++ існує явне й неявне перетворення типів. У загальному випадку неявне перетворення типів зводиться до участі у виразі змінних різного типу (так звана арифметика змішаних типів).

Приклад:

```
int i = 2;
float f1 = 3.4f,f2;
f2 = i + f1;
```

У правій частині оператора присвоєння необхідно знайти суму двох змінних, одна змінна типу **int**, друга – **float**. Але операції можуть виконуватись тільки над змінними одного типу. Тому спочатку перший операнд буде неявно перетворено до типу **float**, а тільки потім буде підрахована сума. Оскільки праворуч тепер стоїть змінна типу **float**, то операція присвоєння буде виконана без перетворення типів.

Якщо операція перетворення типів здійснюється над змінними базових типів, вона може викликати помилки: у випадку, наприклад, якщо результат займає в пам'яті більше місця, ніж відведено під змінну, неминуча втрата значущих розрядів.

Для явного перетворення змінної одного типу в інший перед іменем змінної в дужках вказується новий тип.

Приклад:

```
float f = 25.191 ;
int m = 0 ;
m = (int) f ;                    // явне перетворення типів
cout << "New integer: " << m << '\n' ;
```

У наведеному лістингу після оголошення відповідних змінних (цілої **m** і дійсної **f**) проводиться явне перетворення типу із плаваючої крапки (floating) до цілочисельного (integer).

Приклад неявного перетворення:

```
int m = 0;
float f = 25.191f;
m = f;                    // неявне перетворення типів
cout << "New integer: " << m << '\n';
```

На відміну від попереднього варіанта програми, у цьому випадку після оголошення і ініціалізації змінних здійснюється неявне перетворення значення змінної типу **float** до змінної типу **int**.

Результат роботи обох програм виглядає в такий спосіб:

```
New integer: 25
```

Тобто відбулося відсікання дробової частини змінної **f**.

1.1.5 Коментарі

Існує два типи коментарів: однорядковий та багаторядковий.

Приклад:

//це однорядковий коментар до кінця поточного рядка

*/*це багаторядковий коментар, який може включати декілька рядків тексту*

або операторів/*

1.1.6 Контрольні питання

1. Які вбудовані типи даних Ви знаєте?
2. Скільки пам'яті займає змінна типу **int**?
3. Який синтаксис оголошення перелічення?
4. Яким чином може відбуватись перетворення типів у С++?
5. Перелічіть типи констант в С++.
6. Які типи коментарів Ви знаєте?

1.1.7 Завдання для самостійної роботи

1. Напишіть фрагмент коду, в якому демонструється перетворення типів у С++.
2. Продемонструйте в коді використання коментарів.
3. Продемонструйте в коді використання різних типів констант.

1.2 Операції та оператори

1.2.1 Арифметичні операції та оператори присвоєння

Для виконання обчислень використовуються арифметичні операції. Операцією називається символ, який говорить про те, як треба обробити дані. При математичних обчисленнях використовуються наступні арифметичні операції: додавання (+), віднімання (-), множення (*), ділення (/) і взяття по модулю (%), тобто обчислення остачі від ділення лівого операнда на правий.

Для ефективного використання значень, що вертаються операціями, призначений оператор присвоєння (=) і його модифікації: додавання із присвоєнням (+=), віднімання із присвоєнням (-=), множення із присвоєнням (*=), ділення із присвоєнням (/=), модуль із присвоєнням (%=).

Дані для розрахунків у наведених нижче прикладах:

```
int i = 1, j = 2, m = 5, k;
```

Приклад 1. Яким буде результат виконання наступного оператора:

```
k = i / j;
```

Отримаємо `k = 0`.

Пояснення: 1 поділити на 2 дає 0.5, але оскільки результат ділення цілого на ціле є ціле, то 0.5 неявно перетворюється на ціле шляхом відкидання дробової частини, тобто після перетворення залишається 0, а потім 0 присвоюється змінній **k**.

Приклад 2. Яким буде результат виконання наступного оператора:

```
k = m / j;
```

Отримаємо `k = 2`.

Пояснення: результат ділення буде 2.5. Але оскільки обидва операнди є цілими, то в результаті отримаємо ціле, тобто дробова частина буде відкинута.

Приклад 3. Яким буде результат виконання наступного оператора:

```
float f = i / j;
```

Отримаємо `f = 0`.

Пояснення: хоч **f** оголошено як **float**, але так як і в попередньому випадку, ціле поділене на ціле дає в результаті ціле, тобто результат 0.5 буде шляхом відкидання дробової частини перетворено на 0, а тільки потім ціле буде неявно перетворено на **float**. Для отримання правильного результату в цьому випадку можна записати оператор наступним чином:

```
f = (float) i / j;
```

Тепер для ділення **float** на **int** буде виконано неявне перетворення **int** у **float**. Далі **float** на **float** дасть **float**(0.5) і потім **float** без

перетворення буде присвоєно **float**, тобто результат буде дорівнювати 0.5.

Приклад 4. Яким буде результат виконання наступного оператора:

```
k = i % j;
```

Отримаємо `k = 1`.

Пояснення: % означає знайти залишок від ділення, один на два не ділиться, отже – залишок від ділення буде дорівнювати одиниці.

Приклад 5. Яким буде результат виконання наступного оператора:

```
k = 1; k += j;
```

Отримаємо `k = 3`.

Пояснення: оператор `k + = j` можна записати як `k = k + j;`

Приклад 6. Яким буде результат виконання наступного оператора:

```
k = 1; k -= i + 4;
```

Отримаємо `k = -4`.

Пояснення: оператор `k - = i + 4;` можна записати як `k = k-(i + 4);` розкриваємо дужки, тобто `k = 1 - 1 - 4 = -4`.

Приклад 7. Яким буде результат виконання наступного коду:

```
int x = 2, y = 3, z = 6;
x = y = z;
```

Пояснення: присвоєння виконується справа наліво, тобто спочатку буде виконано `y = z;`

У результаті **y** буде дорівнювати 6.

Потім буде виконано оператор `x = y`.

У результаті x буде теж дорівнювати 6.

1.2.2 Оператори інкремента й декремента

У С++ є ефективний засіб збільшення й зменшення значення операнда на одиницю – унарні оператори інкремента (++) і декре-мента (--).

Стосовно операнду даний вид операторів може бути префіксним або постфіксним. Префіксний оператор застосовується до операнду перед використанням отриманого результату. Постфіксний оператор застосовується до операнду після використання операнда.

Відмінність префікса й постфікса можна спостерігати тільки у виразах із присвоєнням.

Приклад:

```
count++;                  // унарна операція, те ж, що ++ count ;
abc--;                    // те ж, що – - abc ;
x = b++;                  // постфікс
index = --current;        // префікс
```

Тут змінна **b** спочатку привласнюється змінної **x**, а потім збільшується на одиницю. Змінна **current** спочатку зменшується на одиницю, після чого результат привласнюється змінній **index**.

1.2.3 Оператор sizeof

Програми написані на C++ є переносимими. Переносимість означає, що програми, написані для роботи на комп'ютерах одного типу, будуть (хоча й не обов'язково!) працювати і на комп'ютерах іншого типу. Але змінні того самого типу можуть мати різний розмір залежно від того, на якому комп'ютері і в якій операційній системі вони використовуються.

Так, чиста цілочисельна змінна (тип **int** без специфікаторів **short** або **long**) при роботі в 16 розрядній операційній системі (DOS, Windows 3.1 і т. д.) займає всього 2 байта. У той же час під цю же змінну в 32 розрядній операційній системі (Windows 95, Windows NT і т. д.) виділяється вже 4 байта.

Визначити розмір змінної будь-якого типу даних (як базового, так і похідного) можна за допомогою оператора **sizeof**. Операндом **sizeof** може бути або тип, тоді ім'я типу треба обов'язково взяти в дужки, або вираз, тоді дужки можна опустити.

Приклад:

до типу – **sizeof**(**int**),

до константи – **sizeof**(5) або **sizeof** 5,

до змінної – **sizeof**(d) або **sizeof** d.

Розглянута нижче програма допоможе вам визначити розмір базових типів даних для конкретної платформи ПК.

Приклад:

```
int main()
{
   cout << "Size of int type: " << sizeof(int)<<'\n' ;
   cout << "Size of float type: " << sizeof(float)<<'\n' ;
   cout << "Size of double type: " << sizeof(double)<< '\n';
   cout << "Size of char type: "<< sizeof(char)<<'\n';
   return 0;
}
```

У результаті роботи програми за допомогою оператора **sizeof** буде обчислено, а потім виведено на екран розмір кожного з перерахованих базових типів.

1.2.4 Логічні операції

Унарні:	Бінарні:
! – Логічне НІ	&& – Логічне І
	|| – Логічне АБО

Примітка: у мові С за хибність прийнято 0, за істину – будь-яке число, окрім нуля.

Таблиця 1

A	B	!A	A && B	A || B
0	0	1	0	0
0	1	1	0	1
1	0	0	0	1
1	1	0	1	1

Логічні операції найчастіше використовуються в умовних операторах.

1.2.5 Порозрядні логічні операції

На практиці досить часто доводиться відслідковувати стан різних програмних об'єктів за допомогою прапорців. Для цієї мети можна використовувати булевські змінні. Однак якщо у вас занадто багато ознак, зручніше в якості прапорців використовувати окремі біти змінних. Для ПК на базі платформи Intel один байт являє собою вісім біт інформації, а кожний біт може приймати значення 0 або 1. Таким чином, в одному байті може бути закодоване будь-яке ціле число від 0 до 255 включно. Біти в байті рахуються справа наліво.

Розглянемо, наприклад, панель із п'яти вимикачів **sw1** ... **sw5**. Можна було б оголосити в якості прапорів 5 змінних типу **bool** (по 1 байту на кожний об'єкт). Кожний вимикач може встановлювати стан "включене" (1 – ***true***) або "виключене" (0 – ***false***).

Приклад:

```
bool s1, s2, s3, s4, s5;
```

Тоді, перевіряючи значення зазначених вище змінних, можна судити про стан того або іншого вимикача.

Беручи до уваги той факт, що біти в байті також можуть приймати значення тільки нуля або одиниці, використовуючи компактний запис, цю ж панель можна було б умістити всього в одному байті.

Порозрядні логічні операції

& – логічне І (множення)

| – логічне АБО (додавання)
^ – виключне АБО ("виняткове АБО")
~ – логічне НІ (інверсія)

Для того, щоб розробник міг вільно адресуватися до окремих бітів, використовуються так звані порозрядні логічні операції, наведені в таблиці.

Таблиця 2

A	B	A^B	A & B	A | B	~A
0	0	0	0	0	1
0	1	1	0	1	1
1	0	1	0	1	0
1	1	0	1	1	0

Повернемося до прикладу з вимикачами. Для всієї панелі відведемо однобайтну змінну, наприклад, типу **char**.

Приклад:

```
char sw = 0;
```

Присвоївши спочатку змінній **sw** значення 0 (або у двійковому виді 00000000), ми тим самим указуємо, що всі вимикачі перебувають у вимкненому стані.

Припустимо, тепер нам треба включити третій вимикач на панелі. Тобто треба зробити так, щоб змінна **sw** у двійковому виді виглядала як 00000100, або в шістнадцятирічному вигляді 0x04. Застосуємо для цього операцію "логічне АБО".

Приклад:

```
sw = sw | 0x04;                    // включити SW3
sw = sw | 0x01;                    // включити SW1
```

Припустимо, що далі по ходу програми значення байта **sw** змінювалося якимсь чином і перед нами стала задача виключити третій і перший вимикач, не маючи інформації про їхній поточний стан. Для скидання окремих біт у байті на практиці застосовують операцію "логічне І".

Приклад:

```
sw = sw & 0xFA;                    // 0xFA = 11111010
```

Тепер перший і третій біти **sw**, незалежно від того, у якому стані інші біти, будуть встановлені в нуль.

І на закінчення змінимо стан четвертого вимикача на протилежний, незалежно від його вихідного значення, застосувавши операцію "виключне АБО".

Приклад:

```
sw = sw 0x08;                      // 0x08 = 00001000
```

Слід також зазначити, що на практиці часто використовується скорочена форма запису для присвоєння результату порозрядних логічних операторів:

&=	побітове І з присвоюванням
| =	побітове логічне АБО із присвоюванням
^=	виключне побітове АБО із присвоюванням

1.2.6 Операції зсуву вліво й вправо

Для здійснення зсуву послідовності біт вліво й вправо застосовуються відповідно операції << і >> Операнд праворуч від знака операції вказує, на яку величину повинні бути зсунуті біти, задаючи тим самим кількість біт, "виведених" зі змінної, і число нульових біт, що заповнюють змінну з іншої сторони.

Приклад:

```
unsigned char a = 12;          // a = 00001100
a = a << 2;                    // a = 00110000
a = a >> 3;                    // a = 00000110
```

Слід враховувати, що при використанні правого зсуву, якщо найстарший біт дорівнює одиниці (ознака негативного числа в змінних зі специфікатором **signed**), деякі компілятори можуть не "ввести" нулі ліворуч. Щоб уникнути подібних ситуацій, рекомендується перетворювати операнд операції в беззнаковий тип (**unsigned**).

1.2.7 Оператори порівняння

<	Менше
<=	Менше або дорівнює
>	Більше
>=	Більше або дорівнює
==	Дорівнює
!=	Не дорівнює

Для того, щоб була можливість порівнювати між собою значення яких-небудь змінних, мова С++ передбачає так звані оператори порівняння – бінарні оператори виду:

<Операнд1> <Оператор_Порівняння> <Операнд2>

У результаті роботи операторів порівняння вертається логічне значення ***true*** (істина), якщо операнди дорівнюють один одному, або ***false*** (хибність) якщо ні.

1.2.8 Операція "кома"

Операція "кома" зв'язує між собою кілька виразів таким чином, що останні розглядаються компілятором як єдиний вираз.

Завдяки використанню даної операції досягається висока ефективність при написанні програм.

Приклад:

```
if (i = callFunc(), i > 7)
```

Тоді спочатку виконається виклик функції **callFunc** із присвоєнням результату змінної **i**, а потім відбудеться порівняння значення **i** із числом 7.

Ще більшої ефективності можна досягти при використанні операції "кома" в операторі циклу **for**.

1.2.9 Пріоритет і порядок виконання операцій

Як у будь-якій арифметиці, в C++ операції виконуються в певному порядку. Так, наприклад, математичні вирази обчислюються зліва направо, у той час як оператор присвоєння виконується справа наліво.

Для того щоб компілятор міг розібратися, яку ж дію здійснювати першою при виконанні операцій, останні мають важливу властивість – пріоритет. У першу чергу виконуються операції з найвищим пріоритетом.

Мова C++ надає програмісту можливість самостійно задавати порядок виконання обчислень. Із цією метою, як і в математиці, операнди групуються за допомогою дужок, які можуть бути вкладеними один в одного. Не існує обмеження на вкладеність дужок.

1.2.10 Контрольні питання

1. Наведіть приклади записів скорочених операторів присвоєння.
2. Як використовуються оператори інкремента та декремента?
3. Які проблеми можуть виникати при використанні операції зсуву вправо?
4. Як в C++ визначається істина?

1.2.11 Завдання для самостійної роботи

1. Продемонструйте в коді використання скорочених операторів присвоєння.
2. Які два оператори запрограмовані в наступному коді: `s += i++;`
3. Які два оператори запрограмовані в наступному коді: `s /= ++i - 4;`

1.3 Умовні оператори

Для здійснення розгалуження в програмі використовуються так звані умовні оператори.

1.3.1 Оператор if

Оператор **if** робить розгалуження програми залежно від результату перевірки деякої умови на істинність:

if (*умова*) **оператор1**;

Параметр *умова* може бути будь-яким виразом, але найчастіше вираз містить оператори порівняння. Якщо умова, що перевіряється, має істинне значення (***true***), виконується **оператор1**. А якщо ні (***false***), виконання програми переходить до оператора, що йде за умовним оператором.

У конструкціях мови С++ оператори можуть бути блоковими. Це означає, якщо має виконуватись не один, а група операторів, то цю групу треба взяти в фігурні дужки. Всі оператори блоку розглядаються компілятором як єдиний оператор.

Приклад:

```
#include <iostream>
using namespace std;
int main ()
{
   int b;
   cin >>b;
/* якщо умова b > 0 істина, то на екран виводиться текст "b – positive"*/
   if (b > 0) cout << "b – positive";
/* якщо умова b < 0 істина, то на екран виводиться текст "b – negative "*/
   if (b < 0) cout << "b – negative";
   return 0;
}
```

Примітка. Оператор **cin** – об'єкт вхідного потоку простору імен **std**::

```
std::cin << дані;
```

У якості даних можуть бути зазначені змінні, елементи масивів, структури.

Приклад:

```
#include <iostream>
using namespace std;
int main ()
{
   int x; double y;
```

```
// введення значення в змінну x зі стандартного потоку cin
      cin >> x;
      cin >> x >> y;          // Введення двох змінних
      return 0;
   }
```

Типові помилки при використанні оператора **if**:

– Пропуск фігурних дужок для позначення блоку виконуваних операторів.

– Використання в умовних конструкціях оператора присвоєння замість оператора порівняння (= замість ==). Хоча мова С++ дозволяє робити присвоєння в умовних операторах, тут слід бути особливо уважним – у великих проектах подібні помилки локалізуються із труднощами.

Для уникнення таких помилок краще замість конструкції `if(x == 5)` записати `if(5 == x)`.

– У С++ нуль – це ***false***, все, що не дорівнює 0 це ***true***. Тому на практиці часто зустрічаються записи виду:

```
if ( x ) cout << "x <> 0";
//якщо x не дорівнює 0 на екран виводиться текст "X<>0"
```

1.3.2 Оператор if-else

Синтаксис оператора **if** із ключовим словом **else** має такий вигляд:

if (*умова*) ***оператор1***;
else о*ператор2*;

Якщо умова, що перевіряється, має істинне значення (***true***), виконується **оператор1**. А якщо ні (***false***), то програму виконує **оператор2**.

Слід зазначити, що комбінація **if-else** дозволяє значно спростити код програми.

1.3.3 Умовний оператор ?

Замість операторів **if-else** можна використовувати умовний оператор "**?** :". Така конструкція має наступний синтаксис:

***умова* ? вираз1 : вираз2;**

За аналогією з оператором **if**, даний умовний оператор працює так: якщо умова прийняла значення ***true***, виконується *вираз1*, а якщо ***false*** – *вираз2*. Значення, що вертається, привласнюється якій-небудь змінній.

Приклад знаходження максимуму із двох цілих змінних:

```
int a = 10, b = 20, max;
max = (a > b) ? a : b;
cout << max;
```

1.3.4 Оператор switch

Ще однієї альтернативою керуючої конструкції **if-else** може служити оператор розгалуження **switch**.

Синтаксис оператора **switch**:

switch(*вираз*)
{
    case Константа1 : група операторів; [**break** ;]
    case Константа2 : група операторів; [**break** ;]
    [**default** Константа3 : група операторів;]
}

Конструкція **switch-case** являє собою своєрідний "перемикач". Оператор працює у такий спосіб.

На першому етапі аналізується вираз і здійснюється перехід до тієї гілки програми, для якої її значення збігається із зазначеним константним виразом. Далі виконується оператор або група операторів, які йдуть за відповідним **case** оператором доти, доки не зустрінеться ключове слово **break** (відбувається вихід з тіла оператора **switch-case**) або не буде досягнутий кінець блоку конструкції (відповідна фігурна дужка, що закривається). Якщо значення виразу і константних виразів, зазначених в **case**, не збігається з жодним з випадків, виконується гілка програми, описана за допомогою ключового слова **default**. Ключові слова **default** і **break** не є обов'язковими й найчастіше програмісти навмисне їх опускають.

Подібний прийом може в деяких випадках спростити код програми, однак його слід застосовувати дуже акуратно. Наведений нижче фрагмент ілюструє роботу конструкції **switch-case**. Оголошена символьна змінна **answer** служить для приймання із вхідного потоку відповіді користувача на запитання, чи продовжити роботу із програмою.

Приклад:

```
#include <iostream.h>
void main ()
{
   char answer;
   cout << "Продовжити роботу? ";
   cin >> answer;
```

```
    switch (answer)
    {
       case 'y' :
       case 'Y' :
       case 'д' :
       case 'Д': cout << "Продовжимо...\n"; break;
       default : cout << "Завершення...\n";
    }          // продовження роботи
}
```

Як видно із прикладу, користувачеві досить вибрати одну із літер – y, Y, д або Д, щоб продовжити виконання програми, або натиснути будь-яку іншу клавішу для завершення.

Якщо ж помилково пропустити оператор **break**, на екран буде виведено відразу два повідомлення без виконання змістовної частини програми:

```
Продовжимо...
Завершення...
```

Хоча оператор **switch** і допускає вкладення в себе аналогічних операторів, варто уникати подібних конструкцій, тому що такий код візуально сприймається складно.

1.3.5 Контрольні питання

1. Які оператори розгалуження Ви знаєте?
2. З якою метою використовується оператор **break**?
3. Чи обов'язково завершувати оператор **switch** оператором **default**?
4. Якого типу може бути вираз в операторі **switch**?
5. Що буде виведено на екран в результаті виконання наступного коду:

```
int x = 0;
if (x = 1) cout << "x=1";
else cout << "x=0";
```

1.3.6 Завдання для самостійної роботи

1. Продемонструйте в коді використання оператора **switch** без ключового слова **break**.
2. Продемонструйте в коді використання логічних операцій && та ||.
3. Продемонструйте в коді використання операцій зсуву вліво та вправо.

1.4 Оператори циклу

Цикл задає багаторазове проходження по тому самому коду програми (ітерації). Він має точку входження, перевірочну умову і (необов'язково) точку виходу. Цикл, що не має точки виходу, називається **нескінченним**. Для нескінченного циклу перевірочна умова завжди є дійсним значенням.

Перевірка умови може здійснюватися перед виконанням (цикли **for**, **while**) або після закінчення (**do-while**) тіла циклу.

Цикли можуть бути вкладеними один в одного довільним чином.

1.4.1 Цикл for

Оператор циклу **for** використовують, як правило, у тому випадку, коли відома кількість повторень циклу.

Синтаксис циклу **for**:

```
for ([вираз1] ; [вираз2] ; [вираз3])
        оператор;                    // Тіло циклу
```

Цикл **for** оператор працює в такий спосіб:

– спочатку виконується **вираз1**, якщо він присутній у конструкції;

– потім, на початку кожної ітерації, обчислюється **вираз2** (якщо він присутній) і, якщо отриманий результат прийняв значення ***true***, виконується *тіло циклу* (оператор або блок операторів). А якщо ні, то виконання циклу припиняється й здійснюється перехід до оператора, що розташований безпосередньо за тілом циклу. У якості **вираз2** звичайно використовують вираз логічного типу;

– після виконання *тіла циклу* наприкінці кожної ітерації обчислюється **вираз3**, якщо він є в конструкції, і здійснюється перехід до пункту обчислення **вираз2**.

Вираз1 найчастіше служить у якості ініціалізації якої-небудь змінної, що виконує роль лічильника ітерацій.

Вираз2 використовується як перевірочна умова, на практиці часто містить вираз з операторами порівняння. За замовчуванням величина **вираз2** приймає дійсне значення.

Вираз3 служить найчастіше для збільшення значення лічильника циклів або містить вираз, що впливає на перевірочну умову.

Усі три вирази не обов'язково повинні бути присутні у конструкції, однак синтаксис не допускає пропуску символу крапка з комою (**;**). Тому найпростіший приклад нескінченного циклу **for** (виконується постійно до примусового завершення програми) виглядає в такий спосіб:

```
for ( ; ; ) cout << "Нескінченний цикл...'\n'";
```

Якщо в циклі повинні синхронно змінюватися декілька змінних, які залежать від змінної циклу, обчислення їх значень можна помістити в оператор **for**, скориставшись оператором "кома".

Типова помилка програмування циклів **for** – зміна значення лічильника як у конструкції (**вираз3**), так і в тілі циклу. Це може приводити до таких негативних наслідків, як "випадання" ітерацій.

Приклад знаходження суми набору з десяти цілих чисел, починаючи з 10.

```
int s = 0;
for (int i=10; i<20; i++)
   s += i;
cout << "sum = " << s;
```

Як видно із прикладу, при ініціалізації був оголошений лічильник **i** з початковим значенням 10, що збільшується з кожною ітерацією на одиницю. Тіло циклу в цьому випадку складається з одного єдиного оператора, що додає до результуючої величини **s** значення лічильника **i** у даній ітерації.

Запишемо цей оператор з використанням усіх можливостей C++:

```
for (int s = 0, i = 10; i<20; s += i, i++);
```

АБО

```
for (int s = 0, i = 10; i < 20; s += i++);
```

Таким чином, спочатку ініціалізуються змінні **i** і **s**, а потім при кожній з 10-ти ітерації буде обчислюватися нове значення змінних **s** і **i**. Після завершення циклу на екран виводиться значення суми.

Приклад фрагменту коду знаходження суми елементів двовимірного масиву:

```
int arr[ ][3] = {{1,2,3}, {4,5,6}};
int s = 0, i, j;
for (i = 0; i < 2; i++)
        for (j = 0; j < 3; j++)
             s += arr[i][j];
cout << "\ns=" << s;
```

1.4.2 Цикли foreach

У стандарті C++11 було додано підтримку парадигми **foreach** для ітерації по набору даних (аналогічні оператори є, наприклад, в мовах C# та Java). Це корисно, якщо Вам необхідно отримати елементи масиву (контейнера) не піклуючись про індекси та кількість елементів.

Приклад:
```
int arr[] = {1,2,3};
for (int i : arr)
   cout << i << ' ';
```
На екран буде виведено послідовність 1 2 3.

Але якщо необхідно змінити поточний елемент масиву, то замість змінної в заголовку циклу треба, для збереження результату, використовувати посилання.

Приклад:
```
for (int &i : arr)
   cout << i++ << ' ';
```
Як і в попередньому прикладі, на екран буде виведено послідовність 1 2 3. Але після кожного виведення на екран значення змінної **i** буде збільшено на одиницю (постфіксний інкремент). Тому після виконання наступного циклу.
```
for (int i : arr)
   cout << i << ' ';
```
на екран буде виведено послідовність 2 3 4.

1.4.3 Цикл while

Оператор циклу **while** використовують, як правило, у тому випадку, коли невідомо, скільки ітерацій необхідно виконати. Оператор циклу **while** виконує оператор або блок доти, доки перевірочна умова (**вираз**) залишається дійсною.

Синтаксис циклу **while**:

while (*вираз***)**
оператор**;** // *Тіло циклу*

Якщо вираз є ненульовою константою, тіло циклу буде виконуватися завжди, отже, ми маємо справу з нескінченним оператором. Цикл також виявиться нескінченним, коли умова істинна і ніде далі в тілі циклу не змінюється. Якщо ж перевірочна умова повертає ***false***, здійсниться вихід із циклу й тіло оператора **while** буде пропущено.

Досить часто в якості виразу використовується оператор присвоєння. Тому що при цьому вертається деяке число, в операторі **while** фактично проводиться порівняння отриманого значення з нулем (слід нагадати, що нуль – еквівалент ***false***) з подальшим прийняттям рішення про вихід із циклу або його продовження.

Як і для оператора **for**, якщо в циклі повинні синхронно змінюватися декілька змінних, які залежать від змінної циклу, обчислення їх значень можна помістити в перевірочний вираз оператора **while**, скориставшись оператором "кома".

Приклад:

```
#include <iostream>
#include <conio.h>          // для функції getch()
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
   float s = 0, k = 11. , step = 0.1;
   while(s < k)
        s += step;
   cout << "s=" << s << '\n';
   system("pause");         // для затримки виводу на екран
   // або getch();
   return 0;
}
```

1.4.4 Цикл do-while

На відміну від оператора **while**, цикл **do-while** спочатку виконує тіло (оператор або блок), а потім уже здійснює перевірку виразу на істинність. Така конструкція гарантує, що тіло циклу буде обов'язково виконане хоча б один раз.

Синтаксис циклу **do-while**:

```
do
        оператор;                   // Тіло циклу
while (вираз);
```

Одне із часто використовуваних застосувань даного оператора – запит до користувача на продовження виконання програми.

Приклад:

```
char answer;
SetConsoleOutputCP(1251);
do
{
//Тіло програми
   cout << "Продовжити виконання?\n";
   cin >> answer ;
}
//якщо користувач натиснув клавішу Y – продовження циклу
   while ( answer=='Y' || answer=='y' );
```

Таким чином, тіло програми буде повторюватися доти, доки користувач на запитання "*Продовжувати виконання*?" не відповість введенням будь-якого символу, крім 'Y' або 'y'. При цьому, у якості змінної, використовуваної для зберігання цього значення, виступає **answer**.

1.4.5 Оператор break

Для дострокового виходу з оператору циклу, використовується оператор **break**. Даний оператор може зустрічатися в тілі циклу скільки завгодно раз і, як і у випадку **switch-case**, передає керування поза тілом конструкції. На практиці оператор **break** часто використовують для виходу з нескінченного циклу.

```
for ( ; ; )
{
    // Тіло циклу
    if (<умова1>) break;
    if (<умова2>) break;
}
```

У даному прикладі, якщо хоч один із умовних операторів поверне ***true*** відповідний оператор **break** виведе керування з тіла конструкції нескінченного циклу **for**.

1.4.6 Оператор continue

Так само, як і ключове слово **break**, оператор **continue** перериває виконання тіла циклу, але на відміну від першого, він пропонує програмі перейти на наступну ітерацію циклу.

Як приклад використання оператора **continue** пропонується програма знаходження простих чисел (числа, що діляться на 1 і самі на себе).

Приклад:

```
bool simple;                    // прапор простого числа
// для всіх чисел від 2 до 49
for (int i = 2; i < 50; i++)
{
  simple = true ;
// для всіх дільників від 2 до i-1
for (int d = 2; d < i; d++)
{
  if (i % d)                    // якщо i не ділиться націло на d
   continue;                    // завершення ітерації
  else                          // якщо i ділиться націло на d
  {
  simple = false; // число i – непросте
  break;
// вихід із внутрішнього циклу з параметром d
  }
}
```

```
if (simple)                 // якщо число просте
   cout << i;               // вивід на екран числа і
}
```

Програма організована у вигляді двох вкладених циклів таким чином, що здійснюється перебір і перевірка залишку від розподілу пари чисел, перше з яких змінюється від 2 до 49, а друге – від 2 до значення першого числа. На початку кожної ітерації в зовнішньому циклі прапор простого числа **simple** установлюється в стан ***true***. Якщо залишок від розподілу ненульовий, здійснюється продовження внутрішнього циклу по оператору **continue**. У випадку, якщо залишок від розподілу склав 0, виконується вихід із внутрішнього циклу з установкою прапора простого числа **simple** у стан ***false***. По закінченню внутрішнього циклу проводиться аналіз логічної змінної **simple** і вивід простого числа.

1.4.7 Оператор переходу goto і мітки

Мітка являє собою ідентифікатор з розташованим за ним символом двокрапки (**:**). Мітками позначають який-небудь оператор, на який надалі повинен бути здійснений безумовний перехід. Безумовна передача керування на мітку проводиться за допомогою оператора **goto**.

Оператор **goto** може здійснювати перехід (адресуватися) до міток, обов'язково розташованих в одному з ним тілі функції.

Синтаксис оператора **goto**:

goto мітка;
мітка : оператор**;**

Даний оператор – дуже потужний і небезпечний засіб керування поведінкою програми. Використовувати його потрібно вкрай обережно, тому що, наприклад, "стрибок" усередину циклу (обхід кодів ініціалізації) може привести до помилок, які важко локалізувати. За допомогою операторів **if** і **goto** можна реалізувати будь-який алгоритм, але читати й проводити налагодження коду, перенасиченого операторами переходу й мітками, надзвичайно важко. Тому використання оператора переходу вважається поганим стилем програмування.

Приклад застосування оператора **goto** для організації циклу, у якому підсумовуються всі числа від 10 до 20.

```
int i = 10;
int s = 0;
label:
s += i;
```

```
i++;
if (i < 21) goto label;
cout << "s=" << s <<'\n';
```

Як тільки виконання програми досягне оператора **goto**, керування буде передано оператору, що стоїть за міткою **label**.

Загалом кажучи, використання структурного і об'єктно-орієнтованого підходів до програмування дозволяє повністю відмовитися від застосування операторів безумовного переходу. Однак на практиці часто бувають випадки, коли **goto** значно спрощує код програми. Особливо це твердження стосується вкладених конструкцій **switch-case** і **if-else**.

1.4.8 Контрольні питання

1. Перелічіть всі типи операторів циклу.
2. З якою метою використовується в циклі оператор **break**?
3. З якою метою використовується в циклі оператор **continue**?
4. Чим відрізняється оператор **do while** від інших операторів циклу?
5. До чого призведе виконання наступного коду: **for**(;;);

1.4.9 Завдання для самостійної роботи

1. Напишіть фрагмент коду, в якому демонструється виконання в циклі операторів **break** та **continue**.

2. Наведіть приклад коду, який демонструє нескінченний цикл з використанням циклу **while**.

1.5 Функції

Розробка програмного забезпечення на практиці є досить непростим процесом. Програмісту потрібно врахувати всі тонкощі й нюанси як усього програмного комплексу в цілому, так і окремих його частин. Системний підхід до програмування ґрунтується на тому, що поставлене перед розробником завдання попередньо розбивається на менш складні завдання, які, у свою чергу, діляться ще на менш складні завдання, і так доти, поки найдрібніші завдання не будуть вирішені за допомогою стандартних процедур. Таким чином, здійснюється так звана функціональна декомпозиція.

Ключовим елементом даної моделі для рішення конкретного завдання в C++ виступає **функція**. Функцію можна представити як підпрограму або якусь процедуру, що несе закінчене смислове навантаження.

1.5.1 Прототип функції

Кожна функція, яку передбачається використовувати в програмі, повинна бути в ній оголошена. Звичайно оголошення функцій розміщують у заголовних файлах, які потім підключаються до тексту програми за допомогою директиви **#include**. Оголошення функції описує її прототип.

Прототип функції оголошується в такий спосіб:

<тип повертаємого значення> <ім'я функції > ([список параметрів]) ;

тип повертаємого значення **–** тип даних, що вертається функцією. Якщо функція не повертає значення, то тип є **void**.

Список параметрів – задає тип і ім'я кожного параметра функції, розділені комами. Допускається опускати ім'я параметра. Список параметрів може бути порожнім.

Приклад прототипів функцій:

```
int swap (int, int);
double max (double part, double par2);
void func();
```

Прототип функції може бути пропущений, якщо визначення функції знаходиться до першого її виклику з будь-якої іншої функції. Останній варіант є поганим стилем програмування, тому що бувають випадки перехресних викликів, тобто коли дві або кілька функцій викликають одна одну, у результаті чого неможливо викликати одну функцію без попереднього визначення іншої.

1.5.2 Визначення функції

Визначення функції складається з її заголовка та тіла, взятого у фігурні дужки. Якщо функція повертає значення, відмінне від типу **void**, у тілі функції обов'язково повинен бути присутнім оператор **return** з параметром того ж типу, що і повертаєме значення. Якщо повертаєме значення є **void** оператор **return** використовується без параметра або взагалі може бути опущений, тоді повернення з функції здійснюється по досягненню дужки, що закривається.

Для того, щоб функція виконала певні дії, вона повинна бути викликана в програмі. При звертанні до функції вона виконує поставлене завдання, а по закінченню роботи повертає, як результат, деяке значення.

1.5.3 Виклик функції

Виклик функції являє собою ідентифікатор функції (її ім'я), за яким у круглих дужках вказується список аргументів, розділених комами:

ім'я_функції ([аргумент 1, аргумент 2, . . . аргумент n]) ;

Кожний **аргумент** функції являє собою змінну, вираз або константу, передані в тіло функції для подальшого використання в обчислювальному процесі. Список аргументів функції може бути порожнім.

Функція може викликати інші функції (одну або декілька), а ті, у свою чергу, робити виклик третіх і т. д. Крім того, функція може викликати сама себе. Це явище в програмуванні називається **рекурсією**.

Будь-яка програма на С++ обов'язково містить у собі головну функцію **main()**. Саме із цієї функції починається виконання програми.

Приклад:

```
// Оголошення функцій:
   char inputsim ( );
   void print ( short num );

// Виклик функцій:
   char sim = inputsim ( );
   print ( 3 );

// Визначення функцій:
   char inputsim ()
{
   char sim;
   cin >> sim;
   return sim;
}
   void print ( short number )
{
```

```
   cout << "number=" << number;
}
```

Приклад обчислення квадрата числа з використанням функції:

```
   #include <iostream.h>
// містить функції введення-виводу
   using namespace sdt;
   long mySquare ( int );      // прототип функції

   void main ()
   {
// оголошення й ініціалізація змінної типу int
      int n = 5 ;
      cout << mySquare ( n );  // виклик функції
   }

//визначення функції
   long mySquare (int x)       // заголовок функції
   {
      return x * x ;           // тіло функції
   }
```

У результаті роботи програми на екран буде виведене число 25.

1.5.4 Аргументи за замовчуванням

С++ допускає при виклику функції опускати деякі її параметри. Досягається це вказівкою в оголошенні функції значень параметрів за замовчуванням. Для параметрів за замовчуванням існує обов'язкове правило: усі параметри, розташовані праворуч від параметра, який має значення за замовчуванням, повинні також мати значення за замовчуванням.

Приклад помилкового оголошення функції:

```
void showint (int i = 1, bool flag, char symbol = '\n');
```

Оскільки змінна **i** має значення за замовчуванням, то і змінна flag теж повинна мати значення за замовчуванням.

При виклику функції, параметри якої мають значення за замовчуванням, необхідно дотримуватись наступного правила: якщо опущено значення одного із параметрів, які мають значення за замовчуванням, то треба опустити і всі параметри, які розташовані правіше цього параметра.

Приклад, функції, прототип якої, може при виклику мати різний вигляд залежно від ситуації:

```
void showint (int i, bool flag = true, char symbol = '\n');
```

```
showint (1, false, 'a');
showint (2, false);
showint (3);
```

У першому випадку всі три аргументи задані явно, тому робота функції здійснюється у звичайному режимі. У другому виклику у функцію передається два параметри із трьох, причому замість останнього аргументу підставляється значення за замовчуванням, а саме символ **'\n'**. Третій варіант звертання до функції передає тільки один цілочисельний параметр, а в якості інших аргументів використовуються значення за замовчуванням: логічна змінна зі значенням ***true*** і символьна змінна зі значенням **'\n'**.

Наступний виклик функції:

```
showint(1, , 'a');
```

викличе помилку компіляції.

Величина, що вказується в аргументах за замовчуванням, може бути не тільки константним виразом – вона може бути глобальною змінною або значенням, що вертається деякою функцією.

1.5.5 Локальні й глобальні змінні

Змінні можуть бути оголошені як усередині тіла якої-небудь функції, так і за межами кожної з них. Змінні, оголошені усередині тіла функції, називаються **локальними**. Такі змінні розміщуються в стеці програми й діють тільки усередині тієї функції, у якій оголошені. Як тільки керування вертається функції, що викликала дану функцію, пам'ять, виділена під локальні змінні, звільняється.

Кожна змінна характеризується областю дії, областю видимості й часом життя.

Під **областю дії** змінної розуміють область програми, у якій змінна доступна для використання.

Із цим поняттям тісно зв'язане поняття області видимості змінної. Якщо змінна виходить із області дії, вона стає невидимою. З іншого боку, змінна може перебувати в області дії, але бути невидимою. Змінна перебуває в **області видимості**, якщо до неї можна одержати **доступ** (за допомогою операції дозволу видимості, у тому випадку, якщо вона безпосередньо не видима).

Часом життя змінної називається інтервал виконання програми, протягом якого вона існує.

Локальні змінні мають своєю областю видимості функцію або блок, у яких вони оголошені. У той же час область дії локальної змінної може виключати внутрішній блок, якщо в ньому оголошена змінна з тим же іменем. Час життя локальної змінної визначається часом виконання блоку або функції, у якій вона оголошена.

Це означає, наприклад, що в різних функціях можуть використовуватися змінні з однаковими іменами зовсім незалежно одна від одної.

У розглянутому нижче прикладі змінні з іменем **x** визначені відразу у двох функціях – в **main** і в **sum**, що, однак, не заважає компілятору розрізняти їх між собою.

Приклад:

```
int sum (int, int );            // прототип функції sum
void main ()
{
      int x = 2, b = 4;         // локальні змінні
      cout << sum (x, y);
}
int sum (int a, int b)        // заголовок функції
{
      int x = a + b;
// локальну змінну x видно тільки в тілі функції sum
      return x;
}
```

У програмі здійснюється обчислення суми двох цілочисельних змінних за допомогою виклику функції **sum**.

Глобальні змінні, як вказувалося раніше, оголошуються поза тілом будь-якої із функцій і діють протягом виконання всієї програми. Такі змінні доступні в кожній із функцій програми, яка описана після оголошення глобальної змінної. Звідси випливає висновок, що імена локальних і глобальних змінних не повинні збігатися. Якщо глобальна змінна не проініціалізована явно, вона ініціалізується значенням 0.

Область дії глобальної змінної збігається з областю видимості й простирається від точки її опису до кінця файлу, у якому вона оголошена. Час життя глобальної змінної – постійний, тобто збігається із часом виконання програми.

Загалом кажучи, на практиці програмісти намагаються уникати використання глобальних змінних і застосовують їх тільки у випадку крайньої необхідності, тому що вміст таких змінних може бути змінено усередині тіла будь-якої функції, що може призвести до серйозних помилок при роботі програми.

Приклад :

```
int test = 200;          // оголошення глобальної змінної test
void printTest ();       // прототип функції printTest
void main ()
```

```
   {
   int test = 10;          // оголошення локальної змінної test
   printTest () ;          // вивід на друк Глобальна: 200
// вивід на друк Локальна: 10
   cout << "Локальна: " << test << ' \n ' ;
   }
// заголовок функції printtest
   void printTest ()
   {
   cout << "Глобальна: " << test << ' \n ';
   }
```

Спочатку оголошується глобальна змінна **test**, якій присвоюється значення 200. Далі оголошується локальна змінна з тим же іменем **test**, але зі значенням 10. Виклик функції **printTest** з **main** фактично здійснює тимчасовий вихід з тіла головної функції. При цьому всі локальні змінні стають недоступні, і **printTest** виводить на друк глобальну змінну **test**. Після цього керування програмою вертається у функцію **main**, де конструкцією **cout** виводиться на друк локальна змінна **test**.

У C++ допускається оголошувати локальну змінну не тільки на початку функції, а взагалі в будь-якому місці програми. Якщо оголошення відбувається всередині якого-небудь блоку, змінна з таким же іменем, оголошена поза тілом блоку, "ховається".

Змінимо попередній приклад для того, щоб продемонструвати процес приховання локальної змінної:

```
   int test = 200;         // оголошення глобальної змінної test
// прототип функції printTest
   void printTest ();
   int main()
   {
   int test = 10;          // оголошення локальної змінної test printtest () ;
                           // вивід на друк Глобальна: 200
// вивід на друк Локальна: 10
   cout << "Локальна: " << test << ' \n ';
   {
// додаємо новий блок із ще однієї локальної змінної Test
   int test = 5;
// вивід на друк Локальна: 5
   cout << " Локальна: " << test << ' \n ';
   }
// вивід на друк Локальна: 10
```

```
   cout << " Локальна: " << test << ' \n ';
   }
// заголовок функції Printtest
   void printTest ()
   {
   cout << "Глобальна: " << test << ' \n ';
   }
```

1.5.6 Операція ::

Як було показано вище, оголошення локальної змінної приховує глобальну змінну з таким же іменем. Таким чином, усі звернення до імені глобальної змінної в межах області дії локального оголошення викликають звернення до локальної змінної. Однак C++ дозволяє звертатися до глобальної змінної з будь-якого місця програми за допомогою використання операції області видимості. Для цього перед іменем змінної ставиться префікс у вигляді подвійної двокрапки (**::**).

Приклад:

```
   int turn = 5;              // оголошення глобальної змінної turn
   void main ()
   {
   int turn = 70 ;            // оголошення локальної змінної
// вивід локального значення 70
   cout << turn << ' \n ';
// вивід глобального значення 5
   cout << :: turn << ' \n ';
   }
```

З розглянутого прикладу видно, що були оголошені глобальна й локальна змінні з іменем **turn**, які пізніше були виведені на друк.

1.5.7 Класи пам'яті

Існує п'ять модифікаторів змінних, що визначають область видимості і час дії змінних: **auto**, **register**, **extern**, **static**, **volatile**.

1. Автоматичні змінні

Модифікатор **auto** (друге використання цього ключового слова) використовується при описі локальних змінних. Оскільки для локальних змінних даний модифікатор використовується за замовчуванням, на практиці його найчастіше опускають.

Модифікатор **auto** застосовується тільки до локальних змінних, які видно тільки в блоці, в якому вони оголошені. При виході із блоку такі змінні знищуються автоматично.

2. Регістрові змінні

Модифікатор **register** пропонує компілятору спробувати розмістити зазначену змінну в регістрах процесора. Якщо така спроба кінчається невдало, змінна поводиться як локальна змінна типу **auto**. Розміщення змінних у регістрах оптимізує програмний код по швидкості, тому що процесор оперує зі змінними, що перебувають у регістрах, набагато швидше, ніж з пам'яттю. Але у зв'язку з тим, що число регістрів процесора обмежене, кількість таких змінних може бути дуже невеликою.

Приклад:

```
void main ()
{
// оголошення локальної регістрової змінної
register int reg;
. . .
}
```

Модифікатор **register** застосовують тільки до локальних змінних. Спроба вживання даного модифікатора (так само як і модифікатора **auto**) до глобальних змінних викличе повідомлення про помилку. Змінна існує тільки в межах блоку, що містить її оголошення.

3. Зовнішні змінні й функції

Якщо програма складається з декількох модулів, деякі змінні можуть використовуватися для передачі значень із одного файлу в інший. При цьому деяка змінна оголошується глобальною в одному модулі, а в інших файлах, у яких вона повинна бути видима, проводиться її оголошення з використанням модифікатора **extern**. Якщо оголошення зовнішньої змінної проводиться в блоці, вона є локальною.

На відміну від попередніх модифікаторів, цей модифікатор повідомляє, що початкове оголошення змінної проводиться в якомусь іншому файлі.

Приклад використання зовнішньої змінної:

```
// файл myheader.h
// прототип функції changeFlag
   void changeFlag ();
// файл myfunction.cpp
   #include "stdafx.h"
// оголошення зовнішньої глобальної змінної flag
   bool flag;
   void changeFlag ()       //заголовок функції changeFlag
// змінює значення прапора на протилежне
{
```

```
        flag = ! flag;
}

// Головний модуль програми
    #include <iostream.h> // містить функції введення-ввиводу
// підключення заголовного файлу myheader.h
    #include "myheader.h"
// оголошення зовнішньої глобальної змінної flag
    extern bool flag;
    void main ()
    {
    changeFlag () ;         // виклик функції changeFlag
// вивід на печатку значення прапора
    if ( flag ) cout << " flag = true\n";
    else
       cout << " flag = false\n ";
    }
```

Спочатку у файлі **myheader.h** оголошується функція **changeFlag**. Далі у файлі **myfunction.cpp** оголошується глобальна логічна змінна **flag** і визначається реалізація тіла функції **changeFlag** і нарешті в головному модулі підключається заголовний файл **myheader.h** і змінна **flag** описується як зовнішня (**extern**). Оскільки опис функції **changeFlag** включається в головний модуль директивою **#include "myheader.h"**, дана функція доступна в тілі функції **main**.

4. Статичні змінні

Статичні змінні багато в чому схожі на глобальні змінні. Для опису статичних змінних використовується модифікатор **static**. Якщо така змінна оголошена глобально, то вона ініціалізується при запуску програми, а її область видимості збігається з областю дії, і простирається від точки оголошення до кінця файлу. Якщо ж статична змінна оголошена усередині функції або блоку, то вона ініціалізується при першому вході у відповідну функцію або блок. Значення змінної зберігається від одного виклику функції до іншого. Таким чином, статичні змінні можна використовувати для зберігання значень змінних протягом часу роботи програми. Статичні змінні не можуть бути оголошені в інших файлах як зовнішні.

Якщо статична змінна не проініціалізована явно, так, як і глобальна змінна, вона ініціалізується значенням 0.

Приклад реалізації лічильника викликів деякої функції:

```
#include <iostream.h>
int count ();
```

```
void main ()
{
   for (int i = 0; i < 5; i++ )
      cout << count () <<'\n';
}
int count ()
{
   static int counter = 0;
   counter ++;
   return counter;
}
```

У результаті виконання цього коду на екран буде виведено п'ять рядків, перший містить число від 1, останній – 5. Чому так? Головна функція в циклі (5 разів підряд) викликає функцію **count**, яка містить статичну змінну **counter**. Початкова ініціалізація цієї змінної нулем виконується тільки один раз, при першому входженні в тіло функції. В тілі значення змінної буде збільшено на одиницю при кожному виклику функції. Оскільки значення змінної зберігається між викликами функції, на друк буде виведена саме така послідовність чисел.

5. Змінні класу volatile

У тих випадках, коли необхідно передбачити можливість модифікації змінної периферійним пристроєм або іншою програмою, використовують модифікатор **volatile**. У зв'язку із цим компілятор не намагається оптимізувати програму шляхом розміщення змінної в регістрах.

Приклад оголошення таких змінних наведений нижче:

```
volatile short stest;
volatile const int ctest;
```

Як видно із прикладу, змінна **ctest** з модифікатором **volatile** у той же час може бути оголошена як константа. У цьому випадку її значення не зможе мінятися в програмі, але може модифікуватися залежно від зовнішніх факторів.

1.5.8 Новий стиль заголовків

Історично склалося так, що в мові С++ при підключенні заголовних файлів використовувався той же синтаксис, що й у мові С для сумісності з розробленим на той момент програмним забезпеченням. Однак при стандартизації мови цей стиль був змінений і тепер замість заголовних файлів (як це було в С) вказуються деякі стандартні ідентифікатори, по яких компілятор сам знаходить необхідні файли. Визначені ідентифікатори являють собою ім'я заголовка в кутових дужках без вказівки розширення (. h).

Приклад включення заголовків у стилі C++:

```
#include <iostream>
#include <stdlib>
#include <new>
```

Крім цього, для включення в програму бібліотек функцій мови С відповідно до нового стандарту заголовок перетвориться в такий спосіб: відкидається розширення **.h** і до імені заголовка додається префікс **c.** Таким чином, наприклад, заголовок **<string.h>** заміняється заголовком **<cstring>**. Якщо ж використовуваний компілятор не підтримує оголошення заголовків у новому стилі, можна як і раніше використовувати заголовки в стилі мови С, хоча це й не рекомендується стандартом C++.

1.5.9 Простори імен

Визначення функцій і змінних у заголовних файлах нерозривно пов'язано з поняттям простору імен. Це поняття з'явилося порівняно недавно. До введення поняття простору всі оголошення ідентифікаторів і констант, зроблені в заголовному файлі, розміщувались компілятором у глобальному просторі імен. Такий стан речей приводив до виникнення маси конфліктів, пов'язаних з використанням різними об'єктами однакових імен. Найчастіше непорозуміння виникали, коли в одній програмі використовувалися бібліотеки, розроблені різними виробниками. Введення поняття простору імен дозволило значно знизити кількість подібних конфліктів імен. Коли в програму включається заголовок нового стилю, його вміст розміщується не в глобальному просторі імен, а в простір імен **std**. Якщо в програмі потрібно визначити деякі ідентифікатори, які можуть перевизначити вже наявні, просто заведіть свій власний, новий простір імен. Це досягається шляхом використання ключового слова **namespace**. Синтаксис оголошення власного простору імен має вигляд:

```
namespace ім'я_простору_імен
{
   // оголошення
}
```

Таким чином, оголошення всередині нового простору імен будуть перебувати тільки всередині видимості певного імені простору імен, запобігаючи тим самим виникненню конфліктів.

Приклад створення простору імен:

```
namespace NewNameSpace
{
   int x, b, z;
   void someFunction ( char smb );
}
```

Для того, щоб указати компілятору, що слід використовувати імена з конкретного іменного простору (у нашому випадку з **NewNameSpace**), можна скористатися операцією дозволу видимості:

```
NewNameSpace :: x = 5;
```

Однак, якщо в програмі звернення до власного простору імен проводяться досить часто, такий синтаксис викликає певні незручності. У якості альтернативи можна скористатися інструкцією **using**, синтаксис якої має дві форми:

using namespace ім'я_простору_імен;

using ім'я_простору_імен :: ідентифікатор;

При використанні першої форми компілятору повідомляється, що надалі необхідно використовувати ідентифікатори із зазначеного іменного простору аж до того моменту, поки не зустрінеться наступна інструкція **using**. Наприклад, указавши в тілі програми

```
using namespace NewNameSpace;
```

можна прямо працювати з відповідними ідентифікаторами:

```
x = 0; y = z = 4;
someFunction ('A');
```

На практиці часто після включення в програму заголовків явно вказується використання ідентифікаторів стандартного простору імен:

```
using namespace std;
```

Друга форма запису пропонує компілятору використовувати зазначений простір імен лише для конкретного ідентифікатора. Таким чином, визначивши

```
using namespace std;
using NewNameSpace :: z;
```

можна використовувати ідентифікатори стандартної бібліотеки С++ і цілу змінну **z** із простору імен **NewNameSpace** без використання операції області видимості:

```
z = 12;
```

Слід розуміти, що вказівка нового простору імен інструкцією **using namespace** скасовує видимість стандартного простору **std**, тому для одержання доступу до відповідних ідентифікаторів з **std** буде потрібно щоразу використовувати операцію дозволу видимості **std ::**.

Простір імен не може бути оголошений всередині тіла якої-небудь функції, однак може оголошуватися всередині інших просторів. При цьому для доступу до ідентифікатора внутрішнього простору необхідно вказати імена всіх вищих іменованих просторів.

Приклад оголошення простору імен:

```
namespace Highest
{
```

```
namespace Middle
{
   namespace Lowest
   {
      int nattr;
   }
}}
```

Використання оголошеної змінної **nattr** буде виглядати:

```
Highest :: Middle :: Lowest :: nattr = 0;
```

Допускається оголошення декількох іменованих просторів з тим самим іменем, що дозволяє розділити його на кілька файлів. Незважаючи на це, вміст усіх частин буде зберігатися в тому самому просторі імен.

Щоб оголошення змінних і функцій у деякому просторі імен були більш впорядкованими, рекомендується в межах опису простору імен повідомляти тільки прототипи функцій, поміщаючи визначення тіла функції окремо. При цьому слід явно вказувати, до якого простору імен належить функція.

Приклад:

```
namespace Nspace
{
   char z ;
   int i ;
   void fund ( char flag );
}
void Nspace :: fund ( char flag )
{
   // тіло функції
}
```

Крім вищесказаного, допускається оголошення неіменованих просторів імен. У цьому випадку просто опускається ім'я простору після ключового слова **namespace**.

Приклад:

```
namespace
{
   char cbyte ;
   long Ivalue ;
}
```

Звертання до оголошених елементів проводиться по їх імені, без якого-небудь префікса. Неіменовані простори імен можуть бути використані тільки в тому файлі, в якому вони оголошені.

Стандарт мови C++ передбачає визначення псевдонімів простору імен, які посилаються на конкретний простір імен. Найчастіше псевдоніми використовуються для спрощення роботи з довгими іменами просторів. Наступний приклад ілюструє створення більш короткого псевдоніма і його використання для доступу до змінної.

Приклад:

```
namespace A_Very_Long_Name_Of_Namespace
{
   float y;
}
A_Very_Long_Name_Of_Namespace :: y = 0.0;
namespace Neo = A_Very__Long_Name_Of_Namespace;
Neo :: y = 13.4;
```

1.5.10 Вбудовані (inline) функції

У результаті роботи компілятора кожна функція представляється у вигляді машинного коду. Якщо в програмі виклик функції зустрічається кілька раз, у місцях таких викликів генеруються коди виклику вже реалізованого екземпляра функції. Однак виконання викликів вимагає деякої витрати часу. Таким чином, якщо тіло функції невеликого розміру і звернення до неї в програмі відбуваються досить часто, на практиці можна вказати компілятору замість викликів функції у відповідних місцях генерувати все її тіло. Здійснюється це за допомогою ключового слова **inline**. Тим самим збільшується продуктивність реалізованого коду, хоча, звичайно, розмір програми може збільшуватися. Компілятори різних фірм накладають свої обмеження на використання вбудованих функцій, тому перед **використанням inline** функцій необхідно звернутися до інструкції компілятора.

Ключове слово **inline** повинне передувати першому виклику функції, що вбудовується (наприклад, вказуватись в її прототипі).

Приклад:

```
// Прототип функції, що вбудовується
   inline int sum ( int, int );
   void main ()
   {
      int a = 2, b = 6, c = 3;
// Виклики функції, що вбудовується
      cout << sum ( a, b ) << endl;
// генерують усе тіло функції
      cout << sum ( b, 3 ) << endl;
      cout << sum ( a, 3 ) << endl;
   }
```

```
int sum ( int x, int y)
{
   return x + y;
}
```

У наведеному прикладі в кожному місці виклику функції **sum** буде згенеровано код тіла всієї функції.

1.5.11 Рекурсивні функції

Як уже згадувалося раніше, функція може викликати сама себе. При цьому говорять, що виник рекурсивний виклик. Рекурсія буває:

– простою – якщо функція в тілі містить виклик самої себе;

– непрямою – якщо функція викликає іншу функцію, а та в свою чергу викликає першу.

При виконанні рекурсії програма зберігає в стеці значення всіх локальних змінних функції і її аргументів, для того, щоб надалі після повернення з рекурсивного виклику відновити їх збережені значення.

Застосовувати рекурсію треба з обережністю, тому що її використання для функцій, що містять велику кількість змінних або занадто велике число рекурсивних викликів, може викликати переповнення стеку. Слід також пам'ятати, що при використанні рекурсивного виклику розробник зобов'язаний передбачити механізм повернення в функцію, яка викликала поточну, щоб не утворити нескінченний цикл.

Деякі завдання на практиці можуть бути простіше і наочніше вирішені саме з використанням рекурсивних функцій.

Приклад знаходження факторіала із застосуванням рекурсії:

```
long int fact ( long x);
void main ()
{
   int count = 1;
   while ( count && count < 31)
   {
      cout << "Input number: ";
      cin >> count;
      cout << fact ( count ) << ' \n ';
   }
}
long int fact ( long x )
{
   if ( x == 0 || x == 1 )
      return 1;
   else
      return x * fact ( x –1 );
}
```

1.5.12 Математичні функції

Прототипи стандартних математичних функцій визначені в заголовному файлі **math.h**. Розглянемо деякі з них, найбільше часто вживані в повсякденній роботі.

Таблиця 3

Синтаксис функції	Значення, що повертається
double pow (**double** x, **double** y);	число **x** у ступені **y**, **x ≥ 0**
double log (**double** x); **float** logf (**float** x); **long double** logl (**long double** x);	натуральний логарифм числа, **x > 0**
double log10 (**double** x); **float** log10f (**float** x); **long double** log10l(**long double** x);	десятковий логарифм числа, **x > 0**
double sqrt (**double** x); **float** sqrtf (**float** x); **long** double sqrtl (**long double** x);	квадратний корінь числа, **x ≥ 0**
int abs (**int** x); **double** fabs (**double** x); **long** labs (**long** x); **float** fabsf (**float** x); **long double** fabsl (**long double** x);	абсолютне значення числа **x**
double fmod(**double** x, **double** y);	залишок від ділення **x** на **y**
double acos (**double** x); **float** acosf (**float** x);	арккосинус числа, **-1 ≤ x ≤ 1**
double asin (**double** x); **float** asinf (**float** x);	арксинус числа, **-1 ≤ x ≤ 1**
double atan (**double** x); **float** atanf (**float** x);	арктангенс числа **x**
double atan2(**double** x, **double** y); **float** atan2f (**float** x, **float** y);	арктангенс **y/x**
double cos (**double** x); **float** cosf (**float** x);	косинус числа, радіани **x**
double sin (**double** x); **float** sinf (**float** x);	синус числа, радіани **x**
double tan(**double** x); **float** tanf (**float** x);	тангенс числа, радіани **x**
double cosh (**double** x); **float** coshf (**float** x);	гіперболічний косинус числа **x**
double sinh (**double** x); **float** sinhf (**float** x);	гіперболічний синус числа **x**
double tanh (**double** x); **float** tanhf (**float** x);	гіперболічний тангенс **x**

Кути тригонометричних функцій вказуються в радіанах.

Приклад програми, що здійснює переклад градусів у радіани і вивід значення синуса для введеного в градусах числа.

```
#include <iostream>
#include <cmath>
#include <windows.h>
void main ()
{
   SetConsoleOutputCP(1251);
   double angle, PI = 3.14159;
   char s [ 80 ];
   cout << "Введіть кут у градусах :\n ";
   cin >> angle;
   cout<< "Значення сінуса : ";
   cout << sin ( angle * PI / 180 ) << '\n';
}
```

На жаль, у С++ немає готової реалізації функції зведення аргументу в квадрат. Цю функцію можна реалізувати, наприклад, у такий спосіб:

```
inline double sqr ( double x )
{
return x * x;
}
```

1.5.13 Функції округлення

Найчастіше потрібно скористатися округленим значенням тієї або іншої змінної. С++ пропонує набір функцій для рішення цього завдання. Залежно від конкретної ситуації може знадобитися функція, що округляє значення аргументу в більшу або меншу сторону. Розглянемо найбільше часто використовувані варіанти викликів.

Для округлення числа в меншу сторону використовується функція **floor** і її різновиди для різних типів аргументів. Округлення в більшу сторону проводиться за допомогою функції **ceil**.

Таблиця 4

Синтаксис функції	Значення, що вертається
double floor (**double** x); **long double** floorl (**ong double** x);	округлення числа в меншу сторону **x**
double ceil (**double** x); **long double** ceil (**long double** x);	округлення числа в більшу сторону **x**

Однак у реальності проблема вибору, у яку ж сторону робити округлення, покладається на програму, що розробляється.

Приклад:

```
#include <iostream>      // містить функції вводу/ввиводу
#include <math.h>        // містить математичні функції
double round ( double num )
{
// округлення числа в меншу сторону
double val = floor ( num );
double frac = modf ( num, val );    // обчислення залишку
if (frac < 0.5)
// якщо залишок менше 0.5, округляємо в меншу сторону
num = val;
else
// якщо залишок більше 0.5, округляємо в більшу сторону
num = val + 1.0;
return num;
}
```

Для роботи обох варіантів функції необхідно задіяти заголовний файл **math.h** прототип, що **містить** функції **floor** і **modf**.

1.5.14 Перевантаження функцій

Перевантаження функцій – це один з типів поліморфізму, що забезпечуються в С++. У С++ кілька функцій можуть мати те саме ім'я.

Функції, які мають однакове ім'я, але відрізняються або кількістю параметрів, або їх типом або і тим і іншим, називаються перевантаженими.

Перевантажити функції, які відрізняються тільки типом повертаємого значення, не можна. Перевантажені функції дають можливість спростити програми, допускаючи звертання за одним іменем для виконання близьких за змістом дій.

Щоб перевантажити деяку функцію, потрібно просто оголосити, а потім визначити всі необхідні варіанти її виклику. Компілятор автоматично вибере правильний варіант виклику на підставі числа і типу використаних аргументів.

Приклади перевантажених функцій:

– Функції, що відрізняються типом параметрів:

```
void sum(int x, int y);
void sum(double x, double y);
void sum(float x, float y);
```

– Функціїї, що відрізняються кількістю параметрів:

```
int fun(int x, char s);
int fun(char s, int x, float y);
```

– Функції, що відрізняються кількістю та типом параметрів:

```
int fun(char s, int x);
int fun(char s, int x, float y);
```

1.5.15 Контрольні питання

1. Коли виділяється пам'ять під глобальну статичну змінну?
2. Коли виділяється пам'ять під локальну статичну змінну?
3. Чи присвоюється початкове значення за замовчуванням локальній змінній?
4. Який синтаксис має протип функції?
5. Чи є обов'язковою наявність прототипу функції?
6. Що таке параметри функції та чим вони відрізняються від аргументів?
7. Яке правило діє при оголошенні функції, параметри якої мають значення за замовчуванням?
8. Яке правило діє при виклику функції, параметри якої мають значення за замовчуванням?
9. Які функції називаються перевизначеними.
10. Перелічіть переваги та недоліки **inline** функцій.

1.5.16 Завдання для самостійної роботи

1. Напишіть фрагмент коду, який демонструє особливості роботи із локальною статичною змінною.
2. Наведіть приклад коду, який демонструє використання ключового слова **auto**.
3. Напишіть фрагмент коду, в якому міститься оголошення функції, параметри якої мають значення за замовчуванням, та наведіть всі можливі варіанти виклику такої функції.
4. Наведіть приклад коду, який демонструє визначення функції, яка не повертає значення.
5. Продемонструйте в коді використання перевантажених функцій.

1.6 Поняття вказівника

Будь-який об'єкт програми, будь то змінна базового або похідного типу, займає в пам'яті певну область пам'яті. Місце розташування об'єкта в пам'яті визначається його **адресою**. При оголошенні змінної для неї резервується місце в пам'яті, розмір якого залежить від типу даної змінної, а для доступу до вмісту об'єкта служить його ім'я (ідентифікатор). Для того щоб дізнатись адресу конкретної змінної, служить унарна операція взяття адреси. При цьому перед іменем змінної ставиться знак амперсанта (**&**). Наступний нижче приклад програми виведе на друк спочатку значення змінних **x** і **y**, а потім їх адреси.

Приклад:

```
#include <iostream.h>
void main ()
{
int x = 10, y = 20;
// виводить на екран Value x = 10
cout << "Value x = " << x << endl;
// виводить на екран Adress x = 0x0012FF7C
cout << "Adress x = " << &x << endl;
// виводить на екран Value y = 20
cout << "Value y = " << y << endl;
// виводить на екран Adress y = 0x0012FF78
cout << "Adress y = " << &y << endl;
}
```

Як видно із прикладу, локальні змінні розташовуються у стеці у зворотному порядку (стек росте в напрямку молодших адрес). Результат може відрізнятися навіть при повторному запуску програми, тому що неможливо вгадати, по якій адресі почнуть розміщатися змінні. Важливо інше: різниця в адресах першої й другої змінної завжди буде однакова й складе 4 байта.

Потужним засобом розробника програмного забезпечення на С++ є можливість здійснення безпосереднього доступу до пам'яті. Для цієї мети передбачається спеціальний тип змінних – вказівники.

Вказівник (**pointer**) – це змінна, яка в якості значення зберігає адресу іншої змінної. Вказівник може посилатися на змінну (базового або похідного типу) або на функцію. Найбільша ефективність застосування вказівників у розробці додатків досягається при використанні їх з масивами й символьними рядками.

Синтаксис оголошення вказівника:

тип_об'єкта* ідентифікатор ;

Тут **тип_об'єкта** визначає тип даних, на які посилається вказівник з іменем **ідентифікатор**. Символ 'зірочка' (*****) повідомляє компілятор, що оголошена змінна є вказівником і не залежно від того, скільки пам'яті потрібно відвести під сам об'єкт, для вказівника резервується два або чотири байти залежно від моделі пам'яті, що використовується.

Оскільки вказівник являє собою посилання на деяку область пам'яті, йому може бути присвоєна тільки адреса деякої змінної (або функції), а не саме її значення. У випадку некоректного присвоєння компілятор видасть відповідне повідомлення про помилку.

Приклад оголошення і ініціалізації вказівника:

```
char symbol = 'Y';
char *psymbol = &symbol;
long capital = 304L;
long *plong;
plong = &capital;
```

У наведеному фрагменті оголошується символьна змінна **symbol** і ініціалізується значенням **'Y'**, потім визначається вказівник на символьний тип даних **psymbol**, значення якого призначається рівним адресі змінної **symbol**. Слідом за цим оголошується змінна **capital** типу **long** і вказівник на цей же тип **plong**, після чого проводиться ініціалізація вказівника адресою змінної **capital**.

Зверніть увагу на те, що символ "*" в операторі оголошення змінної вказує, що ця змінна є вказівником. Слід уникати запису оператора в рядку 4. Краще відразу проіцізіалізувати вказівник `long* plong =NULL;`

1.6.1 Розіменування вказівників

Вказівники допомагають здійснювати безпосередній доступ до пам'яті. Для того, щоб одержати (прочитати) **значення**, записане в деякій області, на яку посилається вказівник, використовують операцію **розіменування** (*****). При цьому використовується ім'я вказівника із зірочкою перед ним:

```
double d1, d2 = 10;
double *ptr = &d2;
d1 = *ptr;
cout << d1;
```

У наведеному фрагменті оголошуються дві змінні типу double: **d1** і **d2** і вказівник (**ptr**) на тип **double**, проініціалізований адресою змінної **d2**. Після цього за допомогою непрямого доступу до змінної **d1** присвоюється значення, що зберігається за адресою, зазначеною

в **ptr**, тобто фактично значення змінної **d2**, що й підтверджує вивід на друк.

Ще раз зверніть увагу на те, що символ "*" в операторі оголошення змінної вказує, що ця змінна є вказівником. В усіх інших випадках наявність "*" – це операція розіменування.

1.6.2 Порожній вказівник

На практиці досить широко застосовується так званий **порожній вказівник** (типу **void**), який може вказувати на об'єкт будь-якого типу. Для отримання доступу до об'єкта, на який посилається вказівник **void**, його необхідно попередньо привести до того ж типу, що і тип самого об'єкта.

Приклад:

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
   char A = 'q';
   void *ptr;
   ptr = &A;
   *(char*) ptr = 'a';
   cout << "ptr=" << *(char*)ptr << "\na="<<A;
   system("pause");
   return 0;
}
```

1.6.3 Арифметика вказівників

До вказівників (крім вказівників на змінні типу **void**) можуть застосовуватися арифметичні операції:

– вказівники можна віднімати один від одного, тим самим, визначаючи кількість елементів (того ж типу, на який указують вказівники), розташованих між ними. Це корисно при операціях з масивами і символьними рядками;

– від вказівника можна віднімати (або додавати до нього) якенебудь число (ці операції мають сенс тільки для масивів);

– до вказівників можна застосовувати операції інкремента або декремента (ці операції мають сенс тільки для масивів);

– вказівники можна використовувати в операціях порівняння.

Для зміни порожнього вказівника він повинен бути попередньо приведений до якого-небудь типу (не **void**).

Компілятор, знаючи тип вказівника, обчислює розмір змінної цього ж типу, після чого модифікує адресу, що зберігається у вказівнику, відповідно до заданої арифметичної операції, але з урахуванням обчисленого для даного типу розміру. Це означає, що якщо оголошений вказівник типу **double**, на змінну, що займає в пам'яті 8 байт, операція, наприклад, інкремента вказівника збільшить значення адреси не на один, а на вісім байт:

Приклад 1:

Оголошено вказівник

```
double *ptr=NULL;
```

якимсь чином у вказівник **ptr** було записано адресу 100, після виконання оператора інкремента

```
ptr++;
```

ptr буде містити адресу `100 + sizeof(double)`
(або 108 для наших комп'ютерів)

Приклад 2:

Оголошено вказівник

```
int *ptr=NULL;
```

якимсь чином у вказівник **ptr** було записано адресу 100, після виконання оператора інкремента

```
ptr++;
```

ptr буде містити адресу адресу 100 + **sizeof**(**int**) ;
(або 104 для наших комп'ютерів)

Приклад 3:

```
double d;
double *ptr1 = &d;
cout << "\nptr1=" << ptr1;                    //0012ff50
ptr1++;
cout << "\nptr1+1=" << ptr1<<'\n';           // 0012ff58
int a = 4;
int *ptr1 = &a,*ptr2 = &a;
if (ptr1 == ptr2) cout<<"yes";          //буде надруковано yes
```

1.6.4 Константні вказівники та вказівники на константу

При оголошенні вказівників можна використовувати ключове слово **const**. Застосування даного специфікатора трактується компілятором у такий спосіб.

Синтаксис вказівника на константу:

const тип* ідентифікатор ;

Розіменоване значення незмінне. Спроба модифікації вміст вказівника на константу приведе до повідомлення про помилку, тобто не можна написати *ptr=3;

Синтаксис константного вказівника:

тип* const ідентифікатор ;

Завжди вказує на ту саму адресу. Застосування будь-якої арифметичної операції до константного вказівника, приведе до повідомлення про помилку.

Приклад:

```
double f, f1;
//обов'язково присвоїти початкове значення
double *const ptr = &f;
ptr = &f1;                                  // не можна
```

1.6.5 Застосування до вказівників оператора sizeof

Як і до будь-якої змінної або типу даних, до вказівників можна застосовувати операцію визначення розміру **sizeof**. Вище зазначалося, що розмір вказівника може приймати одне із двох значень: два або чотири байти, що дозволяє вказівнику адресувати $2^{(2\times8)}$ = 65 Кбайт або $2^{(4\times8)}$ = 4 Гбайта пам'яті відповідно. На розмір вказівника (2 або 4 байта) впливає обрана модель пам'яті і ряд інших причин.

Приклад:

```
long X = 546213L ;
long* px = &X ;
// виводить на екран розмір вказівника 2 або 4 байта
cout << sizeof ( px ) << '\n' ;
```

1.6.6 Вказівники на вказівники

Вказівники можуть самі посилатися на інші вказівники. При цьому в пам'яті, на яку посилається вказівник, утримується не значення, а адреса якого-небудь об'єкта. Сам об'єкт може також бути вказівником і т. д.

Синтаксис вказівника на вказівник:

тип ідентифікатор ;**

При оголошенні вказівник на вказівник може ініціалізуватися адресою об'єкта. Число символів "зірочка" (*) при оголошенні говорить про "порядок" вказівника.

Щоб одержати доступ до значення, такий вказівник повинен бути розіменований відповідну кількість разів (по числу символів ' * ').

Приклад:

```
// оголошення змінної типу char
char s = 'H' ;
// оголошення вказівника типу char і його ініціалізація
char* ps = &s ;
```

```
// оголошення вказівника на вказівник і його ініціалізація
   char** pps = &ps ;
// оголошення вказівника третього порядку і його ініціалізація
   char*** ppps = &pps ;
// модифікує значення змінної s
   ***ppps = 'L' ;
// виводить на екран значення змінної s
   cout << ***ppps ;
```

1.6.7 Вказівники на функції

У С++ вказівники можуть посилатися на функції. Ім'я функції саме по собі представляє константний вказівник на цю функцію, тобто містить адресу входу в неї. Однак можна задати свій власний вказівник на дану функцію:

тип (*ім'я_вказівника) (список_типів_аргументів)

Приклад:

```
bool ( *funcPtr ) ( char, long );
```

оголошує вказівник **funcPr**, що посилається на функцію, яка повертає логічне значення, та приймає в якості параметрів одну символьну і одну цілу довгу змінну.

Виклик функції через вказівник здійснюється так, начебто ім'я вказівника є просто іменем функції, що викликається. Тобто, після імені вказівника вказується список аргументів, очікуваних функцією. Так, для наведеного вище вказівника на функцію **funcPtr** виклик може виглядати, наприклад, у такий спосіб:

```
bool flag;
char symbol = 'x';
long number = 202L;
flag = funcPtr ( symbol, number );
```

Вказівники на функції використовуються часто в якості аргументів інших функцій. У такий спосіб створюються універсальні функції, що значно спрощують текст складної програми (наприклад, чисельне рішення рівнянь, диференціювання, інтегрування). Деякі бібліотечні функції, як параметр, також приймають вказівники на функції.

Приклад функції (**common**), яка приймає в якості параметра вказівник на іншу функцію. Таким чином, результат роботи функції залежить від того, яка функція передається в якості параметра. Головна вимога – сігнатури функцій, які передаються в якості параметра, повинні співпадати.

Приклад:

```
#include <iostream>
```

```
using namespace std;
void common(int(*arifmetic)(int a,int b),int a, int b)
{
   cout << arifmetic(a,b) << '\n';
}

int sum(int a, int b)
{
return a+b;
}

int subtraction(int a, int b)
{
return a-b;
}

int _tmain(int argc, _TCHAR* argv[])
{
   common(sum,2,3);
   common(subtraction,2,3);
   system("pause");
   return 0;
}
```

1.6.8 Посилання

Посилання – особливий тип даних, що є схованою формою вказівника, який при використанні автоматично розіменовується. Іншими словами, він може використовуватися просто як інше ім'я, або псевдонім об'єкта. При оголошенні посилання перед іменем змінної ставиться знак амперсанта, а сама вона повинна бути відразу проініціалізована іменем того об'єкта, на який посилається:

тип &ім'я_посилання = ім'я_змінної;

Тип об'єкта, на який вказується посилання, може бути будь-яким. Оголошення неініціалізованого посилання викличе повідомлення компілятора про помилку (крім ситуації, коли посилання оголошується як **extern**). Будь-яка зміна значення посилання спричинить зміну того об'єкта, на який дане посилання вказує.

Приклад:

```
int v = 0;          // оголошення змінної типу int
// оголошення посилання на змінну типу int і її ініціалізація
int &ref = v;
ref += 10;          // те ж, що й v += 10
```

Після виконання наведеного фрагмента значення обох змінні **v** і **ref** буде дорівнювати 10. Використання посилань не пов'язане з додатковими витратами пам'яті.

Посилання не можна перепризначати. Спроба перепризначити наявне посилання якій-небудь іншій змінній приведе до присвоєння оригіналу об'єкта значення іншої змінної.

Приклад:

```
// оголошення змінних a і b типу char їх ініціалізація
   char a = 'A', b = 'B';
// оголошення посилання на змінну a і її ініціалізація
   char &refa = a;
// присвоєння посиланню refa значення змінної b
   refa = b;
   cout << refa;              // виводить на екран символ 'B'.
```

Крім того, слід врахувати, що посилатися можна тільки на сам об'єкт. Не можна оголосити посилання на тип об'єкта. Ще одне обмеження, що накладається на посилання, полягає в тому, що вони не можуть вказувати на нульовий об'єкт, тобто на об'єкт, який має значення **null**. Таким чином, якщо є ймовірність того, що об'єкт в результаті роботи додатка стане нульовим, від посилання слід відмовитися на користь застосування вказівника.

1.6.9 Функції. Передача параметрів за посиланням та за значенням

Параметри у функцію можуть передаватися одним із наступних способів:

- за значенням;
- за посиланням.

При передачі аргументів **за значенням** компілятор створює тимчасову копію об'єкта, який повинен бути переданий, і розміщує її в області стекової пам'яті, призначеної для зберігання локальних об'єктів. Функція, що викликається, оперує саме із цією копією, не впливаючи на оригінал об'єкта.

Приклад:

```
void swap(int x, int y)
{
   int temp = x;
   x = y;
   y = temp;
   cout << "x=" << x << "y=" << y << '\n';
}
```

```
int _tmain(int argc, _TCHAR* argv[])
{
   int a = 3, b = 4;
   swap(a,b);
   cout << "a=" << a << "b=" << b << '\n';
   system("pause");
   return 0;
}
```

У результаті виконання оператора

```
cout<<"x="<<x<<"y="<<y<<'\n';
```

у функції **swap** на друк буде виведено:

```
x = 4 y = 3
```

а після повернення у функцію **_tmain** в результаті виконання оператора

```
cout << "a=" << a << "b=" << b << '\n';
```

на друк буде виведено:

```
a = 3 b = 4
```

оскільки у функцію передавались не змінні **a** та **b** , а їх копії.

Якщо ж необхідно, щоб функція модифікувала оригінал об'єкта, використовується передача параметрів **за посиланням**. При цьому в функцію передається не сам об'єкт, а тільки його адреса. Таким чином, всі модифікації в тілі функції переданих їй за посиланням аргументів впливають на об'єкт. Беручи до уваги той факт, що функція може повертати лише єдине значення, використання передачі адреси об'єкта виявляється досить ефективним способом роботи із великим за обсягом числом даних. Крім того, тим, що передається адреса, а не сам об'єкт, суттєво заощаджується стекова пам'ять.

У C++ передача за посиланням може здійснюватися двома способами:

- використовуючи безпосередньо посилання;
- за допомогою вказівників.

Синтаксис передачі з використанням посилань має на увазі застосування як аргумент посилання на тип об'єкта. **Наприклад**, функція

```
double glue ( long& x, int& y );
```

одержує два посилання на змінні типу **long** і **int**. При передачі у функцію параметра за посиланням компілятор автоматично передає у функцію адресу змінної, зазначеної в якості аргумента. Ставити знак амперсанта перед аргументом у виклику функції не потрібно. Наприклад, для попередньої функції виклик з передачею параметрів за посиланням виглядає в такий спосіб:

```
c = glue (a, b);
```

Приклад прототипу функції при передачі параметрів через вказівник:

```
void setnumber ( int*, long* );
```

Тоді виклик функції має наступний вигляд:

```
setnumber (&n, &a);
```

Приклад функції, яка приймає в якості параметра дві змінні, та міняє їх місцями (параметри передаються з використанням посилань):

```
void swap(int &x, int &y)
{
   int temp = x;
   x = y;
   y = temp;
   cout << "x=" << x << "y=" << y << '\n';
}

int _tmain(int argc, _TCHAR* argv[])
{
   int a = 3,b = 4;
   swap(a, b);
   cout << "a=" << a << "b=" << b << '\n';
   system("pause");
   return 0;
}
```

Приклад функції, яка приймає в якості параметра дві змінні, та міняє їх місцями (параметри передаються з використанням вказівників):

```
void swap (int *x, int *y)
{
   int temp = *x;
   *x = *y;
   *y = temp;
   cout << "x=" << *x << "y=" << *y << '\n';

}
int _tmain(int argc, _TCHAR* argv[])
{
   int a = 3, b = 4;
   swap (&a, &b);
   cout << "a=" << a << "b=" << b << '\n';
   system("pause");
   return 0;
}
```

Якщо передати параметри за значенням, зміни не будуть збережені, бо в функцію буде передано копії змінних, а не їх адреси.

Крім того, функції можуть повертати не тільки значення деякої змінної, але й вказівник або посилання. Наприклад, функції, прототип яких:

```
int* count ( int );
int& increase ( );
```

повертають вказівник і посилання відповідно на цілу змінну типу **int**. Слід мати на увазі, що повернення посилання або вказівника з функції може привести до проблем, якщо змінна, на яку робиться посилання, вийшла з області видимості.

Приклад:

```
int& func () { int x ; return x ; }
```

У цьому випадку спроба повернути посилання на локальну змінну **x** приведе до помилки, яка з'ясується тільки в ході виконання програми.

Ефективність передачі адреси об'єкта замість самої змінної відчутна і у швидкості роботи, особливо, якщо використовуються великі об'єкти, зокрема масиви.

Якщо потрібно у функцію передати досить великий об'єкт, однак його модифікація не передбачається, на практиці використовується передача константного вказівника. Даний тип виклику припускає використання ключового слова **const**, наприклад, функція

```
int* const fname ( int* const number )
```

приймає і повертає вказівник на константний об'єкт типу **int**. Будь-яка спроба модифікувати такий об'єкт у межах тіла функції, що викликається, викличе повідомлення компілятора про помилку.

Приклад, що ілюструє використання константних вказівників:

```
#include <iostream>
using namespace std;
int* const call ( int* const );
/* прототип функції, яка приймає й повертає константний вказівник типу int*/
void main ()
{
int x = 13;                    // оголошення змінної x, її ініціалізація
// оголошення вказівника на змінну x, його ініціалізація
int* px = &x;
call ( px );                   // виклик функції call
system("pause");
}
```

```
int* const call ( int* const x )      // заголовок функції
{
   int k = 88;
// виводить на екран значення змінної, на яку вказує вказівник x
   cout << *x;
//x = &k;                  // не можна модифікувати об'єкт!
   *x = 99;                 //все добре
   return x;
}
```

Замість наведеного вище синтаксису константного вказівника як альтернативи при передачі параметрів можна використовувати константні посилання, що мають той же зміст, що й константні вказівники.

Приклад:

```
/* заголовок функції, яка приймає й повертає константне посилання типу int */
const int& call ( const int& x )
{
/* виводить на екран значення змінної, на яку посилається посилання */
cout << x ;
// x++ ;                    // не можна модифікувати об'єкт!
return x ;
}

void main ()
{
SetConsoleOutputCP(1251);
int y = 13 ;                // оголошення змінної x, її ініціалізація
// оголошення посилання на змінну x і її ініціалізація
int& rx = y ;
// виклик функції call
cout << "\nПовертаємо значення=" << call ( rx ) ;
system("pause");
}
```

1.6.10 Контрольні питання

1. Дайте визначення вказівника.
2. Для чого використовується розіменування вказівника?
3. Синтаксис оголошення вказівників.
4. Що таке ім'я функції в C++?
5. Чи можна передати функцію в якості параметра?

6. Які арифметичні операції можна виконувати над вказівниками?
7. Що таке константний вказівник?
8. Що таке вказівник на константу?
9. Якими способами можна передавати параметри у функцію?

1.6.11 Завдання для самостійної роботи

1. Напишіть фрагмент коду, який демонструє 3 способи передачі параметрів у функцію.
2. Наведіть приклад коду, який демонструє передачу функції в якості параметру.
3. Наведіть приклад коду, який демонструє арифметику вказівників.

1.7 Масиви

У повсякденному житті постійно доводиться зустрічатися з однотипними об'єктами. Як і багато інших мов високого рівня, C++ надає програмісту можливість роботи з наборами однотипних даних – **масивами**. Окрема одиниця таких даних, що входять у масив, називається **елементом масиву**. У якості елементів масиву можуть виступати дані будь-якого типу (один тип даних для кожного масиву), а також вказівники на однотипні дані. Масиви бувають **одновимірними** і **багатовимірними**.

Оскільки всі елементи масиву мають один тип, вони також мають однаковий розмір. Використання масиву в програмі передує його оголошення, що резервує під масив певну кількість пам'яті. При цьому вказується тип елементів масиву, ім'я масиву і його розмір. Розмір повідомляє компілятор, яка кількість елементів буде розміщена в масиві.

Синтаксис оголошення одновимірного масиву:

тип ідентифікатор [кількість_елементів] ;

Приклад оголошення

```
int array [ 20 ] ;
```

зарезервує в пам'яті місце для розміщення двадцяти цілих елементів.

Елементи масиву в пам'яті розташовуються безпосередньо один за одним. На рисунку показане розташування одномірного масиву двобайтних елементів (типу **short**) у пам'яті.

Звернення до елементів масиву може здійснюватися одним із двох способів:

- за номером елементу в масиві (через його індекс);
- за допомогою вказівника.

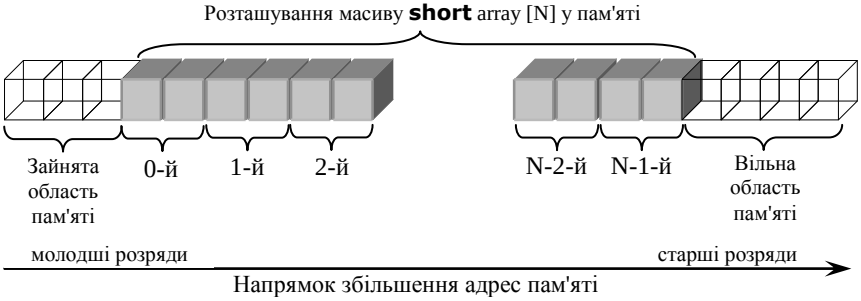

Рис. 1

При зверненні через індекс за іменем масиву у квадратних дужках вказується номер елемента, до якого потрібно виконати доступ. Слід пам'ятати, що в С++ елементи масиву нумеруються, починаючи з 0. Перший елемент масиву має індекс 0, другий – індекс 1 і т. д. Таким чином, запис типу:

```
x = array [13];
y = array [19];
```

виконує присвоєння змінній **x** значення 14-го елемента, а змінній **y** – значення 20-го елемента масиву.

Доступ до елементів масиву через вказівник полягає в наступному. Ім'я масиву, що оголошується, асоціюється компілятором з адресою його найпершого елемента (з індексом 0), а точніше – ім'я масиву – це константний вказівник на перший елемент масиву. Таким чином, можна присвоїти вказівнику адресу нульового елемента, використовуючи ім'я масиву.

Приклад:

```
// оголошення й ініціалізація масиву символів array
char array [ ] = { 'W', 'O', 'R', 'L', 'D' };
char* parray = array;          // parray указує на array [ 0 ]
```

Розіменуванням вказівника **parray**, можна отримати доступ до змісту **array [0]**;

```
// оголошення й ініціалізація символьної змінної
char letter = *parray;
cout << letter << '\n';        // виводить на екран W
```

Оскільки в С++ вказівники та масиви тісно взаємопов'язані, збільшуючи або зменшуючи значення вказівника на масив, програміст отримує можливість доступу до всіх елементів масиву шляхом відповідної модифікації вказівника.

Приклад:

```
parray += 3;                   // збільшує адресу на 3 байта
cout << *parray << '\n';       // виводить на екран L
parray ++;                     // збільшує адресу на 1 байт
cout << *parray << '\n';       // виводить на екран D
```

Таким чином, після проведених арифметичних операцій вказівник **parray** буде посилатися на елемент масиву з індексом 4. До цього ж елемента можна звернутися іншим способом:

```
letter = *( array + 4 ) ;      // еквівалент letter = array [ 4 ] ;
```

А от оператор **array**++; викличе помилку, бо оскільки **array** константний вказівник, то не можна змінити адресу, яку він містить.

Присвоєння значень одного масиву значенням іншого масиву виду **array [ ] = another [ ]** або **array = another** теж є неприпустимим.

Для виконання такої операції програмісту необхідно вживати певні дії (при доступі до елементів по індексу найчастіше використовується циклічне присвоєння). Оголошення виду

```
char ( * parray) [ 10 ] ;
```

визначає вказівник **parray** на масив з 10 символів (**char**). Якщо ж вилучити дужки, компілятор зрозуміє запис як оголошення масиву з 10 вказівників на тип **char**.

Приклад використання масиву.

```
// оголошення масиву цілих, що містить 5 елементів
   short num [5];
// заповнення всіх елементів масиву в циклі
   for (int i = 0; i < 5; i++)
              num [ i ] = i;
// вивід значення з 3-го по 5-е
   for (int i = 2; i < 5; i++)
      cout << num [ i ] <<'\n';
```

1.7.1 Ініціалізація масивів

Ініціалізацію масивів, що містять елементи базових типів, можна робити при їх оголошенні. При цьому безпосередньо після оголошення необхідно за знаком рівності (=) перелічити значення елементів у фігурних дужках через кому (,).

Приклад:

```
int temp [ 12 ] = { 2, 4, 7, 11, 12, 12, 13, 12, 10, 8, 5, 1 };
```

проініціалізує масив температур **temp** відповідними значеннями. Так, елемент **temp[0]** одержить значення 2, елемент **temp[1]** – значення 4 і т. д. до елемента **temp[11]** із присвоєнням йому значення 1.

Якщо в списку ініціалізації значень зазначено менше, ніж оголошено в розмірі масиву, останні елементи будуть проініціалізовані 0. У цьому випадку іноді після останнього значення в ініціалізуючому виразі для наочності ставлять кому:

```
int Temp [12] = {2, 4, 7, };
```

Таким чином, елементи **temp[0]**, **temp[1]** і **temp[2]** отримають значення із списку ініціалізації, всі інші будуть мати значення 0.

При оголошенні одномірного масиву з одночасною його ініціалізацією дозволяється опускати значення розміру, що звичайно вказується у квадратних дужках. При цьому компілятор самостійно підрахує кількість елементів у списку ініціалізації і виділить під них необхідну область пам'яті.

Приклад 1:

```
// виділення в пам'яті місця для зберігання шести об'єктів типу int
int even [ ] = { 0, 2, 4, 6, 8, 10 };
```

Якщо далі в програмі буде потрібно визначити, скільки елементів є в масиві, можна скористатися наступним виразом:

```
int size = sizeof ( even ) / sizeof ( even [ 0 ] );
```

Тут вираз **sizeof(even)** визначає загальний розмір, що займає масив **even** у пам'яті (у байтах), а вираз **sizeof(even[0])** повертає розмір (теж у байтах) одного елемента масиву.

Приклад 2. Введення елементів масиву з клавіатури та виведення їх на екран.

```
#include <iostream>
#include "windows.h"

using namespace std;

int _tmain(int argc, _TCHAR* argv[])
{
int a[10];                          // оголошення масиву
// введення елементів масиву з клавіатури
for (int i = 0; i < 10; i++)
   cin >> a[i];
cout << "\n Mass:\n";
// виведення елементів масиву на екран
for (int i = 0; i < 10; i++)
   cout << a[i];
system("pause");
return 0;
}
```

Примітка. Інструкція `cout << a[i];` виведе елементи масиву без роздільника між ними.

Наприклад, задано масив `int a[3] = {0, 1 ,2};`

1. Код:

```
for(int i=0; i<3;i++)
      cout << a[i];
```

Результат: 012

2. Код:

```
for(int i=0; i<3;i++)
      cout << a[i] << " ";
```

Результат: 0 1 2

3. Код:
```
for(int i=0; i<3;i++)
     cout << setw (3) << a[i];
```
Результат: 0 1 2

Для застосування маніпулятора **setw(n)** необхідно підключити бібліотеку **#include <iomanip>**, де **n** – мінімальна кількість позицій для **a[i]**

4. Код:
```
for(int i=0; i<3;i++)
    cout << a[i] << endl;
```
Результат:
```
0
1
2
```
5. Код:
```
for(int i=0; i<3;i++)
    cout << "a[" << i + 1 << "] – " << a[i] << endl;
```
Результат:
```
A[1] – 0
A[2] – 1
A[3] – 2
```

Приклад 4. Ініціалізація елементів масиву випадковими числами в діапазоні [0, N].

```
#include <iostream>
#include <iomanip>
#include "windows.h"
#include "time.h"          // для функції time()

using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
 int m[30];
 int N = 10;
 srand(time(NULL)); //ініціалізація датчика випадкових чисел
for (int i = 0; i <= 29; i++)
{
 m[i] = rand() % (N + 1); //заповнення масиву
 cout << setw(4) << m[i]; // вивід елементів масиву
}
system("pause");
return 0;
}
```

Приклад 5. Ініціалізація елементів масиву випадковими числами в діапазоні [a, b].

```
int _tmain(int argc, _TCHAR* argv[])
{
 int m[30];
 int b = 100;
 int a = -100;
 srand(time(NULL));
 for (int i = 0; i <= 29; i++)
  {
   m[i] = rand() % (b – a + 1) + a;
   cout << setw(4) << m[i];
  }
 system("pause");
 return 0;
}
```

1.7.2 Багатовимірні масиви

Багатовимірний масив розмірності **N** можна представити як одномірний масив з масивів розмірності (**N**-1). Таким чином, наприклад тривимірний масив – це масив, кожний елемент якого представляє двовимірну матрицю.

Приклад оголошення багатомірних масивів:

```
// двовимірний масив 6×9 елементів
char mat2d [6] [9];
// тривимірний масив 4×2×8 елементів
unsigned long arr3d [4] [2] [8];
```

Вираз **a[i][j]**, що представляє двовимірний масив, переводиться компілятором в еквівалентний вираз:

```
*( *( a + i ) + j )
```

Елементи двовимірного масиву **a[i][j]** розташовуються в оперативній пам'яті так, як показано на рисунку – спочатку елементи нульового рядка (i = 0), потім першого (i = 1), другого (i = 2) і т. д.

Багатомірні масиви ініціалізуються в порядку найшвидшої зміни самого правого індексу (задом наперед): спочатку відбувається присвоєння початкових значень усім елементам останнього індексу, потім попереднього і т. д. до самого початку.

Приклад:

```
int mass [2] [2] [3] = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12};
```

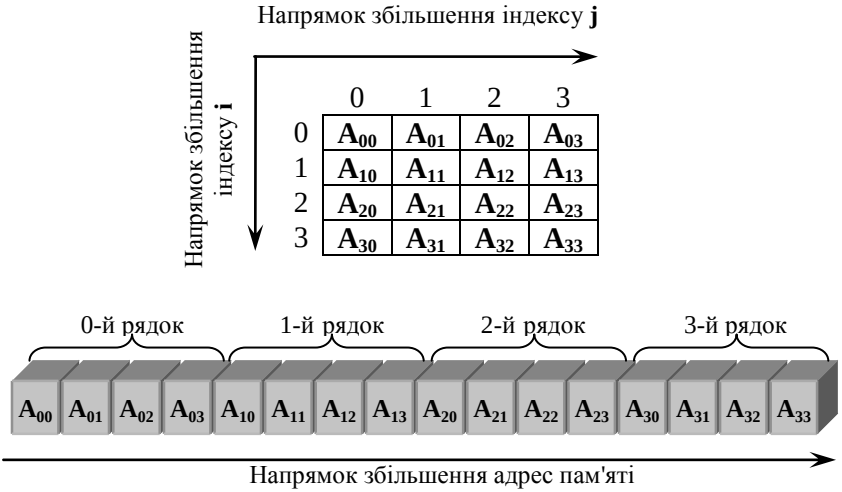

Рис. 2

У цьому випадку елементи масиву будуть приймати наступні значення:

mass [0] [0] [0] = 1 ; mass [0] [0] [1] = 2 ; mass [0] [0] [2] = 3 ;
mass [0] [1] [0] = 4 ; mass [0] [1] [1] = 5 ; mass [0] [1] [2] = 6 ;
mass [1] [0] [0] = 7 ; mass [1] [0] [1] = 8 ; mass [1] [0] [2] = 9 ;
mass [1] [1] [0] = 10; mass [1] [1] [1] = 11; mass [1] [1] [2] = 12;

Щоб не заплутатися, для наочності можна групувати дані за допомогою проміжних фігурних дужок:

```
int mass [2] [2] [3] = { { { 1, 2, 3 }, { 4, 5, 6 } },
                         { { 7, 8, 9 }, { 10,11, 12 }}};
```

Для багатомірних масивів при ініціалізації дозволяється опускати тільки величину першої розмірності:

```
int mat [ ] [3] = {{9, 8, 7}, {6, 5, 4}, {3, 2,1 }};
```

Доступ до елементів багатомірного масиву через вказівники здійснюється набагато складніше. Оскільки, наприклад, двовимірний масив **mat[i][j]** може бути представлений як одномірний (**mat[i]**), кожний елемент якого також є одномірним масивом (**mat[j]**), вказівник на двовимірний масив **pmat**, посилаючись на елемент масиву **mat[i][j]**, по суті, вказує на масив **mat[j]** у масиві **mat[i]**. Таким чином, для доступу до вмісту пам'яті, на яку вказує вказівник **pmat** прийдеться розіменовувати двічі.

Приклад:

```
char arr [3] [2] = { 'W', '0', 'R', 'L', 'D',' ! ' };
```

```
char* parr = (char*) arr;
parr += 3;
char letter = *parr;
cout << letter;
```

У наведеному прикладі оголошується масив символів розмірністю 3×2 і вказівник **parr** на нього (фактично – вказівник на **arr[0][0]**).

У рядку

```
char* parr = (char*) arr;
```

ідентифікатор **arr** вже є вказівником на елемент із індексом 0, однак, оскільки масив двовимірний, потрібно його повторне розіменування.

Збільшення **parr** на 3 приводить до того, що він вказує на елемент масиву, значення якого – символ "**L**" (елемент **arr[1][1]**). Далі здійснюється вивід значення елемента масиву, на яке вказує **parr**.

Розглянемо приклад, у якому на екран виводиться двовимірний масив розміром **n×m** у вигляді таблиці. Звернення до елементів масиву здійснюється за допомогою індексів – номера рядка **i** та номера стовпця **j**.

Приклад:

```
const n = 3, m = 4;                    // розмір масиву
// оголошення й ініціалізація двовимірного масиву
int mat [n] [m] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
// для всіх номерів рядків і від 0 до n
for (int i = 0; i < n; i++)
{
cout << " i = " << i <<'\t';     // вивід на екран номер рядка
// для всіх номерів стовпців j від 0 до m
for (int j = 0; j < m; j++)
// вивідна екран елемент масиву з i рядка й j стовпця
    cout << mat [ i ] [ j ]<<' \t ';
// перехід на початок наступного рядка
    cout<<' \n ';
}
```

На екран буде виведено:

```
i=0   1     2       3       4
i=1   5     6       7       8
i=2   9     10      11      12
```

Розглянемо приклад, у якому демонструється, що ідентифікатор **mat+j** є вказівником на **j** рядок і що двовимірний масив є масивом одномірних масивів.

Приклад:

```
for (int i = 0; i < n; i++)
   cout <<"i="<< i <<' \t '<< mat + i <<' \n ';
```

На екран буде виведено, наприклад:

```
i=0     0x0012FF48
i=1     0x0012FF58
i=2     0x0012FF68
```

Наведений фрагмент коду виводить на екран значення вказівників на рядки. Різниця між адресами становить 4*4 байта = 16 байт.

Розіменування вказівників на рядки дозволяє одержати вказівники на елементи двовимірного масиву, що показане в наступному прикладі:

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
   const int n = 3, m = 4;            // розмір масиву
// оголошення й ініціалізація двовимірного масиву
   int mat [n] [m] = { {1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
   cout << &mat[0][0]<<"\n";
   for (int i = 0; i < n; i++)
   {
      cout<<hex;
// вивід на екран номер рядка
      cout << "\n i = "<< i << '\n';
// для всіх номерів стовпців j від 0 до m
      for (int j = 0; j < m; j++)
/* вивід на екран значення вказівника на елемент масиву з i рядка й j
стовпця */
      cout << " j= "<<j<<"=="<< *(mat + i) + j <<"\n";
      cout << '\n';
   }
   system("pause");
   return 0;
}
```

На екран буде виведено, наприклад:

```
i=0   0x0012FF48   0x0012FF4C   0x0012FF50   0x0012FF54
i=1   0x0012FF58   0x0012FF5C   0x0012FF60   0x0012FF64
i=2   0x0012FF68   0x0012FF6C   0x0012FF70   0x0012FF74
```

Розіменування вказівників на елементи двовимірного масиву дозволяє одержати значення самих елементів, що показано в наступному прикладі:

```
// для всіх номерів рядків i від 0 до n
   for ( int i = 0; i < n; i++)
```

```
   {
// вивід на екран номер рядка
   cout << " i= " << i << ' \t ';
// для всіх номерів стовпців j від 0 до m
   for ( int j = 0; j < m; j++)
// вивід на екран елемент масиву з i рядка й j стовпця
      cout<< *(*(mat + i) + j) << ' \t ';
   cout << ' \n ' ;
   }
```

Оскільки елементи двовимірного масиву розташовуються в пам'яті в тому ж порядку, що й виводяться на екран, скористаємося цією обставиною в наступному прикладі для рішення тієї ж задачі.

Приклад:

```
#include <iostream>
using namespace std;
int _tmain(int argc, _TCHAR* argv[])
{
   const int n = 3, m = 4;            // розмір масиву
// оголошення й ініціалізація двовимірного масиву
   int mat [n] [m] = {{1,2,3,4}, {5,6,7,8}, {9,10,11,12}};
   int *parr=&mat[0][0];
   for (int i = 0; i < n*m; i++)
   {
      cout << *parr++ << '\t';
      if ((i + 1) % m == 0)
           cout << '\n';
   }
   system("pause");
   return 0;
}
```

Таким чином, кожен рядок масиву буде виведено з нового рядка на моніторі.

1.7.3 Динамічне розміщення масивів

У програмі кожна змінна може розміщатися в одному із трьох місць: в області даних програми, у стеці або у вільній пам'яті (так звана купа).

Кожній змінній в програмі пам'ять може виділятись або статично, тобто в момент завантаження програми, або динамічно – у процесі виконання програми. Дотепер обумовлені масиви оголошувалися статично, і, отже, зберігали значення всіх своїх елементів у стековій

пам'яті або області даних програми. Якщо кількість елементів масиву невелика, таке розміщення виправдане. Однак досить часто виникають випадки, коли в стековій пам'яті, що містить локальні змінні і допоміжну інформацію (наприклад, точки повернення із вкладених функцій), недостатньо місця для розміщення всіх елементів великого масиву. Ситуація ще погіршується, якщо масивів великого розміру повинно бути багато. Тут на допомогу приходить можливість використання для зберігання даних динамічної пам'яті.

1.7.4 Виділення й звільнення пам'яті в мові С

Щоб надалі можна було розмістити в пам'яті деякий динамічний об'єкт, для нього необхідно попередньо виділити в пам'яті відповідне місце. По закінченню роботи з об'єктом виділену для його зберігання пам'ять потрібно звільнити.

Виділення динамічної пам'яті під об'єкт в мові С здійснювалось за допомогою наступних функцій: **malloc** і **calloc**. Звільнення виділених ресурсів пам'яті проводиться функцією **free**.

1.7.5 Функція malloc

Функція **malloc** підключається в одному із заголовних файлів **stdlib.h** або **alloc.h.**

Синтаксис функції:

```
void *malloc ( size_t size );
```

Дана функція покликана виділити в пам'яті блок розміром **size** байт із кучі.

У випадку успішного резервування блоку пам'яті функція **malloc** повертає вказівник на тільки що виділений блок.

При невдалому результаті операції з пам'яттю функція повертає **NULL** – зарезервований ідентифікатор зі значенням **'\0'** (абсолютне значення **0x00**). При цьому вміст блоку залишається незмінним. Якщо в якості аргументу функції зазначене значення 0, функція повертає **NULL**.

Як видно із синтаксису, дана функція повертає вказівник типу **void**, однак, оскільки на практиці найчастіше доводиться виділяти пам'ять для об'єктів конкретного типу, доводиться приводити тип отриманого порожнього вказівника до необхідного типу. Наприклад, виділення пам'яті під три об'єкти типу **int** і визначення вказівника на початок виділеного блоку можна зробити в такий спосіб:

```
int *pint = (int*) malloc (3*sizeof (int));
```

1.7.6 Функція calloc

На відміну від **malloc**, функція **calloc** крім виділення області пам'яті під масив об'єктів ще робить ініціалізацію елементів масиву нульовими значеннями.

Синтаксис функції:

```
void *calloc (size_t num, size_t size);
```

Аргумент **num** вказує, скільки елементів буде зберігатися в масиві, а параметр **size** повідомляє розмір кожного елемента в байтах. Наведений тут тип **size_t** (оголошений у заголовному файлі **stddef.h**) є синонімом типу **unsigned char**, що повертається оператором **sizeof**.

1.7.7 Функція free

Функція звільнення пам'яті **free** у якості єдиного аргументу приймає вказівник на блок пам'яті.

Синтаксис функції:

```
void free ( void *block );
```

Дана функція підключається в заголовному файлі **stdlib.h** або **alloc.h**. Тип об'єктів блоку, що видаляється, може бути довільним, на що вказує ключове слово **void**.

Розглянемо приклад виділення динамічної пам'яті для десяти об'єктів типу **int**, заповнення виділеної області числами від нуля до дев'яти і звільнення ресурсу пам'яті.

Приклад:

```
#include <iostream.h>
#include <stdlib.h>
#include <windows.h>
int main ()
{
// оголошення вказівника на область пам'яті
int* preg;
// виділення області пам'яті
if ((preg = (int*) malloc (10*sizeof(int))) == NULL)
{
   cout<< "\n Недостатньо пам'яті \n";
// завершення програми, якщо не вистачає пам'яті для виділення
   return 1;
}
// заповнення виділеного блоку пам'яті
for (int i=0 ; i<10 ; i++ ) *( preg + i ) = i ;
// вивід заповненого блоку
for (i = 0; i < 10; i++)
```

```
        cout << *(preg + i) << ' \n';
    free (preg) ;            // звільнення блоку пам'яті
    }
```

Приклад динамічного виділення пам'яті під двовимірний масив **array[n][m]** змінних типу **int** виглядає в такий спосіб:

```
int* array = (int*) malloc (n*m*sizeof (int));
```

1.7.8 Виділення і звільнення пам'яті в мові С++

У мові С++ для динамічного виділення пам'яті краще використовувати оператори **new** і **delete**.

На відміну від функцій роботи з динамічною пам'яттю **malloc**, **calloc** і **free**, запозичених в С++ зі стандарту ANSII для сумісності, нові оператори динамічного розподілу пам'яті **new** і **delete** мають додаткові можливості, перераховані нижче.

– Функції **malloc** і **calloc** повертають порожній вказівник, який надалі потрібно приводити до заданого типу. Оператор **new** повертає вказівник на тип, для якого виділялася пам'ять, і додаткових перетворень уже не потрібно.

– Операція **new** припускає можливість використання разом з бібліотечною функцією **set_new_handler**, що дозволяє користувачеві визначити свою власну процедуру обробки помилки при виділенні пам'яті.

– Оператори **new** і **delete** можуть бути перевантажені для того, щоб вони, наприклад, могли приймати додаткові параметри або виконувати специфічні дії з урахуванням конкретної ситуації роботи з пам'яттю.

Оператори **new** і **delete** мають дві форми:

– керування динамічним розміщенням у пам'яті одиничного об'єкта;

– динамічне розміщення масиву об'єктів.

Синтаксис операторів **new** і **delete** при роботі з одиничними об'єктами наступний:

тип_об'єкта *ім'я = new тип_об'єкта;
delete ім'я;

При керуванні життєвим циклом масиву об'єктів синтаксис операторів **new** і **delete** має вигляд:

тип_об'єкта *ім'я = new тип_об'єкта [кількість];
delete [] ім'я;

Тут параметр **кількість** в операторові **new[]** характеризує кількість об'єктів типу **тип_об'єкта**, для яких проводиться виділення області пам'яті. У випадку успішного резервування пам'яті змінна

вказівник **ім'я** посилається на початок виділеної області. При видаленні масиву його розмір вказувати не потрібно.

Форма оператора **delete** повинна обов'язково відповідати формі оператора **new** для даного об'єкта: якщо виділення пам'яті проводилося для одного об'єкта (**new**), звільнення пам'яті також повинне здійснюватися для одного об'єкта (**delete**). У випадку застосування оператора **new[]** (для масиву об'єктів), в остаточному результаті вивільнення пам'яті повинно бути зроблене з використанням оператора **delete[]**. Якщо пам'ять була виділена оператором **new[],** а звільнена **delete** без дужок, то буде звільнена пам'ять, розподілена під перший елемент масиву. Вся інша пам'ять буде зайнята до кінця роботи додатку. Ніяких попереджень надруковано не буде. Такий ефект називається "витік пам'яті". Аналогічний ефект отримаємо, якщо пам'ять за допомогою оператора **new** розподілена, а відповідного оператора **delete** не було**.** Такі помилки дуже складні в пошуку. Це одна із головних проблем мови С++.

Приклад:

```
#include <iostream.h>
#include <windows.h>
void main ()
{
const n = 100;                  // розмір масиву – константа типу int
// оголошення вказівника на змінну типу int
int* pint;
// виділення пам'яті для масиву з n об'єктів типу int
pint = new int [n];
// ініціалізація масиву
for (int i = 0; i < n; i++)
     *(pint + i) = n – i;
// вивід масиву на екран
for (i = 0; i < n; i++)
     cout << *( pint + i ) << '\t';
delete [ ] ptr;                 // видалення масиву з пам'яті
const m =100000;                // розмір масиву
// оголошення вказівника типу long double
double* plong;
// якщо пам'ять виділена
if (plong = new long double [m])
{
   cout<<" \n Готово ! \n ";
 delete [ ] plong;              // видалення масиву з пам'яті
```

```
}
else                                  // якщо пам'ять не виділена
{
   cout<<" \n Невдача... \n ");
}}
```

У головній функції програми оголошено цілочисельний вказівник **ptr**, якому відразу оператором **new** привласнюється адреса виділеного блоку динамічної пам'яті для **n** значень типу **int**. Далі область пам'яті заповнюється в циклі числами від **n** до 1, а в наступному циклі здійснюється вивід її вмісту, після чого виділений блок динамічної пам'яті звільняється (оператор **delete[]**).

Наступний крок – спроба виділення пам'яті для великого масиву типу **long double**. Якщо результат операції **new[]** не нульовий, вказівнику **plong** буде привласнене значення адреси зарезервованої пам'яті з виводом напису "Готово!" і наступним вивільненням області. У випадку невдалого виділення виводиться повідомлення "Невдача...".

1.7.9 Масиви як параметри функцій

У тіло функції в якості аргументів можна передавати значення, що зберігаються в масивах. При виклику функції, параметр типу масиву перетворюється компілятором у вказівник на тип масиву. Наприклад, якщо аргумент масив має тип **unsigned long**, при виклику він буде перетворений в **unsigned long***. Таким чином, зміна у функції значення будь-якого елемента масиву, що є аргументом, обов'язково вплине і на оригінал. Масиви відрізняються від інших типів тим, що їх не можна передавати за значенням – усередину тіла функції попадає тільки адреса масиву.

Якщо треба запобігти зміні значень елементів масиву, то треба оголосити параметр с ключовим словом **const**.

Приклади передачі одновимірних масивів в якості параметрів.

Приклад 1:

```
void fun(const int arr[],int size)
{
   cout <<"\nspisok arr\n";
   for (int i = 0; i < size; i++)
      cout <<'\n'<<arr[i];
}
int _tmain(int argc, _TCHAR* argv[])
{
 int arr[] = {1,2,3};
```

```
    fun(arr,3);
    system("pause");
    return 0;
  }
```

Зверніть увагу на те, що розмір масиву передається в функцію як параметр. Немає можливості визначити в функції розмірність масиву, який був переданий в якості параметра.

Приклад 2:

```
void fun(int *arr,int size)
{
   cout << "\nspisok arr\n";
   for(int i = 0; i < size; i++)
      cout << '\n' << arr[i];
}

int _tmain(int argc, _TCHAR* argv[])
{
     int arr[] = {1,2,3};
     fun(arr,3);
      system("pause");
      return 0;
}
```

Зверніть увагу на те, що виклик функції не змінився, змінився тільки синтаксис заголовку функції. Це можливо тому, що ім'я масиву перетворюється компілятором на вказівник.

Приклад 3:

```
void fun(int arr[],int size)
{
   arr++;
}

int _tmain(int argc, _TCHAR* argv[])
{
   int arr[]={1,2,3};
// arr++;
   fun(arr,3);
   system("pause");
   return 0;
}
```

Зверніть увагу на те, що в функції розміщено оператор **arr**++; Він не є помилковим, бо, як вказано в попередньому коментарі, ім'я

масиву перетворюється на вказівник, а не на константний вказівник. В основній програмі аналогічний оператор стоїть в коментарі, бо він є помилковим, компілятор вважає ім'я масиву константним вказівником, а відповідно неможливо змінити адресу яку він містить.

Приклад передачі двовимірних масивів в якості параметрів.

Приклад 4:

```
void fun(int arr[][3],int i1,int i2)
{
   for(int i=0; i<i1;i++)
      for(int j=0;j<i2;j++)
         cout<<"\ni="<<i<<"J="<<j<<" znach arr="<<arr[i][j];
}

int _tmain(int argc, _TCHAR* argv[])
{
        int arr[][3]={{1,2,3},{4,5,6}};
        fun(arr,3);
        system("pause");
        return 0;
}
```

При оголошенні масиву в параметрах функції всі розміри, окрім першого, треба обов'язково вказати.

1.7.10 Типові алгоритми обробки масивів

1. Алгоритми роботи з лінійними масивами

До найбільш поширених умов, які використовуються при роботі з одномірними масивами належать:

Вибірка	Умова
парні елементи масиву	`a[i] % 2 == 0`
непарні елементи масиву	`a[i] % 2 != 0`
елементи масиву, які кратні числу **t**	`a[i] % t == 0`
елементи масиву, які некратні числу **t**	`a[i] % t != 0`
елементи масиву з парними індексами	`(i+1) % 2 == 0`
елементи масиву з непарними індексами	`(i+1) % 2 != 0`
позитивні елементи масиву	`a[i] > 0`
негативні елементи масиву	`a[i] < 0`

Приклад 1. Вивести елемент масиву, який є позитивним, кратним 3-м, і має парний індекс:

```
if ((a[i] > 0) && (a[i] % 3 == 0) && ((i+1) % 2 == 0))
  cout << a[i];
```

Приклад 2. Вивести індекс елементу масиву, якщо елемент масиву є непарним числом, а його індекс парний:

```
if ((a[i] % 2 != 0) && ((i+1) % 2 == 0))
   cout << i;
```

До типових операцій обробки одномірних масивів належать:

1. Визначення суми / добутку / середнього арифметичного елементів масиву.
2. Підрахунок кількості елементів масиву, що задовольняють умові.
3. Визначення мінімального / максимального елементів масиву.
4. Підрахунок кількості входжень заданого елементу до масиву.

Приклад 3. Пошук мінімального елементу та його індексу.

```
int _tmain(int argc, _TCHAR* argv[])
{
  const int n = 10;
  int a[n] = { -7, 11, 7, 3, 4, -5, 6, 7, -13, 9 };
// min_i індекс мінімального елементу
  int min_i = 0;
  for (int i = 1; i < n; i++)
// умова пошуку мінімального елементу
  if (a[i] < a[min_i])
  { min_i = i; }
  cout << "Min element a[" << min_i << "] = "<< a[min_i];
  system("pause");
  return 0;
}
```

Алгоритм пошуку мінімального / максимального елементів масиву:

`int min_i = 0;`	Вважаємо, що мінімальний елемент масиву знаходиться на 0 позиції. Запам'ятовуємо його індекс в змінній **min_i**
`for (int i = 1; i < n; i++)`	Переглядаємо в циклі решту елементів масиву
`if (a[i] < a[min_i])`	Якщо поточний елемент масиву **a[i]** менший за опорний елемент (елемент з індексом **min_i**)
`{ min_i = i; }`	тоді запам'ятовуємо індекс поточного елементу – це індекс мінімального елементу
`cout << "Min element a[" << min_i << "] = " << a[min_i];`	Вивід індексу та значення мінімального елементу масиву

Для визначення максимального елементу буде інша умова: поточний елемент **a**[**i**] має бути більший за опорний елемент

Приклад 4. Підрахунок кількості позитивних елементів та їх добутку.

```
int _tmain(int argc, _TCHAR* argv[])
{
const int n = 10;
int a[n] = { -7, 11, 7, 3, 4, -5, 6, 7, -13, 9 }; /*
оголошення та ініціалізація змінної count для підрахунку кількості елементів*/
int count = 0;
/* оголошення та ініціалізація змінної mul для підрахунку добутку
елементів*/
int mul = 1;
for (int i = 0; i < n; i++)
 if (a[i] > 0)
  {
// підрахунок кількості елементів, що задовольняють умові
      count++;
// підрахунок добутку елементів, що задовольняють умові
      mul *= a[i];
  }
// вивід на екран
  cout << "Count =" << count << endl << "Mul =" << mul;
  system("pause");
  return 0;
}
```

Примітка. При підрахунку суми або кількості елементів – початкове значення змінної має дорівнювати 0; при підрахунку добутку – початкове значення змінної має дорівнювати 1.

Приклад 5. Визначення кількості входжень заданого елементу до масиву.

```
int _tmain(int argc, _TCHAR* argv[])
{
 const int n = 10;
 int a[n] = { -7, 11, 7, 3, 4, -5, 6, 7, -13, 9 };
 int count = 0;  //ініціалізація лічильника
 int key;        // оголошення змінної-ключа key
 cout << " Enter – ";
 cin >> key;     // введення значення змінної-ключа
 for (int i = 0; i < n; i++)
```

```
// якщо елемент масиву = змінній-ключу,
            if (a[i] == key)
            {
// тоді підрахувати кількість входжень
            count++;
            cout << "a[" << i << "]" << endl;
            }
// вивід кількості входжень змінної key в масиві a
     cout << "Count =" << count << endl;
     system("pause");
     return 0;
   }
```

2. Алгоритми роботи з двовимірними масивами

При роботі з квадратними матрицями для виділення елементів масиву відносно діагоналі використовують умови:

Вибірка	Умова
елементи головної діагоналі	`i == j`
елементи побічної діагоналі	`j == n-1-i`
елементи над головною діагоналлю	`j > i`
елементи під головною діагоналлю	`j < i`

де **i** – номер рядка, **j** – номер стовпчика

Приклад 6. Підрахунок суми елементів парних рядків.

```
   int _tmain(int argc, _TCHAR* argv[])
   {
     const int n = 10, m = 10;
     int b[n][m];
     srand(time(NULL));
     for (int i = 0, sum; i < n; i++)
      {
        if (i % 2 == 0)                    // для парних рядків
        {
            sum = 0;  // обнулити суму для кожного рядка
            for (int j = 0; j < m; j++)
             {
// ініціалізація масиву випадковими числами від -10 до 10
             b[i][j] = rand() % 21 – 10;
             sum += b[i][j];               // підрахунок суми
             cout << setw(4) << b[i][j];
             }
cout << setw(10) << "Рядок:" << i << setw(10) << "Сума:" << sum;
```

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

Примітка. У ***прикладі 6*** ми визначали суму елементів для кожного рядка, відповідно зовнішній цикл – це цикл за рядками; внутрішній – за стовпчиками. Перед обрахунком суми для кожного рядка змінна **sum** має дорівнювати 0: **sum** = 0.

Приклад 7. Підрахунок середнього арифметичного елементів кожного стовпчика.

```
   int _tmain(int argc, _TCHAR* argv[])
   {
    const int n = 5, m = 10;
    int b[n][m];
    srand(time(NULL));
    for (int i = 0, sum = 0; i < n; i++)
     {
      for (int j = 0; j < m; j++)
       {
// ініціалізація масиву випадковими числами від -10 до 10
          b[i][j] = rand() % 21 – 10;
          cout << setw(4) << b[i][j];
       }
      cout << endl;
     }
    double avg;
    for (int j = 0; j < m; j++)
     {
        avg = 0;
        for (int i = 0; i < n; i++)
            avg += b[i][j];
        avg /= n;
cout << setw(10) << "Стовпчик:"<< j << setw(10) <<"Avg:"
<< avg;
     }
    system("pause");
    return 0;
   }
```

Примітка. У ***прикладі 7*** ми визначали середнє арифметичне для кожного стовпчика, відповідно зовнішній цикл – це цикл за стовпчиками; внутрішній – за рядками.

Приклад 8. Обнулити елементи масиву над головною діагоналлю.

```
int _tmain(int argc, _TCHAR* argv[])
{
 const int n = 5, m = 5;
 int b[n][m];
 srand(time(NULL));
 for (int i = 0, sum = 0; i < n; i++)
  {
   for (int j = 0; j < m; j++)
    {
// ініціалізація масиву випадковими числами від -10 до 10
     b[i][j] = rand() % 21 – 10;
     cout << setw(4) << b[i][j];
    }
   cout << endl;
  }
 for (int i = 0; i < n; i++)
  {
   for (int j = 0; j < m; j++)
    {
// умова для виділення елементів над головною діагоналлю
     if (j > i)
      {
       b[i][j] = 0; // обнуляємо відповідні елементи
      }
// вивід елементів рядка i
     cout << setw(4) << b[i][j];
  }
     cout << endl; // перехід на новий рядок
    }
 system("pause");
 return 0;
}
```

Приклад 9. Сформувати масив 6 х 6, як показано на рисунку 3.

Рис. 3

Визначаємо умови заповнення:

Умова 1	Рядок 0	(i == 0)
Умова 2	Останній рядок	(i == n-1)
Умова 3	Парні стовпчики	(j % 2 == 0)

```
int _tmain(int argc, _TCHAR* argv[])
{
const int n = 6, m = 6;
 char b[n][m];
 for (int i = 0; i < n; i++)
 {
 for (int j = 0; j < m; j++)
  {
/* записуємо складену умову: якщо нульовий рядок або останній рядок
або парний стовпчик */
    if ((i == 0) || (i == n – 1) || (j % 2 == 0))
     {
        b[i][j] = '*'; // тоді записати символ *
     }
     else
        {
         b[i][j] = ' '; // інакше записати пробіл
        }
        cout.width(2);
        cout << b[i][j];
   }
      cout << endl;
 }
system("pause");
return 0;
}
```

1.7.11 Контрольні питання

1. Синтаксис оголошення масиву.
2. Як компілятор розглядає ім'я масиву?
3. Як визначити розмір одновимірного масиву?
4. Як передати одновимірний масив у функцію?
5. Як передати багатовимірний масив у функцію?
6. Як заборонити зміну в функції елементів масиву, переданого в якості параметру?
7. Як присвоїти вказівнику адресу одновимірного масиву?
8. Як присвоїти вказівнику адресу двовимірного масиву?
9. Як звернутись до елементу одновимірного масиву без використання індексу?
10. Як звернутись до елементу двовимірного масиву без використання індексу?

1.7.12 Завдання для самостійної роботи

1. Напишіть фрагмент коду, який демонструє звернення до елементу одновимірного та двовимірного масиву (з використанням та без використання індексу).
2. Продемонструйте в коді передачу в функцію одновимірного масиву в якості параметра.
3. Продемонструйте в коді передачу в функцію двовимірного масиву в якості параметра.

1.8 Рядки та операції з ними

1.8.1 Масиви символів в С++

У стандарт С++ включена підтримка декількох наборів символів. Традиційний 8-бітовий набір символів називається "вузькими" символами. Крім того, включена підтримка 16-бітових символів, які називаються "широкими". Для кожного із цих наборів символів у бібліотеці є своя сукупність функцій.

Як і в С , для символьних рядків в С++ не існує спеціального рядкового типу. Замість цього рядки в С++ представляються як масиви елементів типу **char**, що закінчуються **термінатором рядка** – символом з нульовим значенням (**'\0'**). Рядки, що закінчуються нуль-термінатором, часто називають ASCII рядками. Символьні рядки складаються з набору символьних констант, взятих в подвійні лапки:

```
"Це рядок символів..."
```

У таблиці наведений набір констант, що застосовуються в С++ у якості символів.

Таблиця 5

Прописна літера	від **'A'** до **'Z'**, від **'А'** до **'Я'**
Мала літера	від **'a'** до **'z'**, від **'а'** до **'я'**
Цифра	від **'0'** до **'9'**
Порожнє місце	горизонтальна табуляція **'\9'**, переклад рядка (код ASCII 10), вертикальна табуляція (код ASCII 11), переклад форми (код ASCII 12), повернення каретки (код ASCII 13)
Символи пунктуації	**! " # $ % & ' () * +, – . / : ;** **< = > ? @ [\ ] _ { | } ~**
Керуючий символ	усі символи з кодами від 0 до 1F і символ з кодом 7F
Пробіл	символ пробілу (код ASCII 32)
Шістнадцятирічна цифра	від **'0'** до **'9'**, від **'A'** до **'F'**, від **'а'** до **'f'**

При оголошенні рядкового масиву необхідно брати до уваги наявність термінатора наприкінці рядка, відводячи тим самим під рядок на один байт більше:

/ оголошення рядка розміром 10 символів, включаючи термінатор. Реальний розмір рядка: 9 символів + термінатор. */*

```
char buffer [10];
```

Рядковий масив може при оголошенні ініціалізуватись початковим значенням. При цьому компілятор автоматично обчислює розмір майбутнього рядка і додає в кінець нуль-термінатор.

Приклад:

```
// оголошення й ініціалізація рядка
   char Wednesday [] = "Середа";
// що рівносильне
   char Wednesday [] = { 'С', 'е', 'р', 'е', 'д', 'а', '\0'};
```

1.8.2 Контрольні питання

1. Що таке рядок мови С?
2. Чим відрізняється рядок від масиву символів?
3. Як визначити розмір рядка?

1.8.3 Завдання для самостійної роботи

1. Продемонструйте в коді роботу не менше трьох функцій роботи з рядками.

1.9 Структури

1.9.1 Визначення структури

Масиви – набір однотипних даних. Структури – набір різнотипних або однотипних даних.

Синтаксис оголошення структури:

struct ім'я_структури{
    тип_змінна ім'я_змінна1;
    тип_змінна ім'я_змінна2;
    . . .
    тип_змінна ім'я_змінnan;
};

Приклад оголошення структури:

```
struct Person{
   char name[20];
   int vozvrast;
   float rost;
};
```

Зверніть увагу на те, що оголошення структури завершується крапкою з комою.

1.9.2 Використання структури

Для використання структури після її оголошення, треба створити об'єкт структури.

Синтаксис створення об'єкта структури:

<ім'я_структури> < ім'я_змінної>;

Приклад:

Person student1, student2;

Створити об'єкт структури можна одночасно з оголошенням структури:

```
struct Person{
   char name[20];
   int vozvrast;
   float rost;
} student1, student2;
```

1.9.3 Ініціалізація об'єкта структури

Присвоєння початкових значень членам структури можна під час оголошення структури:

```
struct Person{
```

```
    char name[20];
    int vozvrast;
    float rost;
} student1 = {"Петров", 20, 1.75};
```

Або під час створення об'єкта структури:

```
Person student1 = {"Петров", 20, 1.75};
```

1.9.4 Доступ до елементів структури

Доступ до елементів структури здійснюється за допомогою використання "крапки".

Приклад:

```
student2.vozvrast = 22;
student2.rost = 1.68;
student2.name = "Женя";                    // не правильно
/*один рядок можна присвоїти іншому тільки за допомогою функції strcpy */
strcpy(student2.name,"Женя");
```

1.9.5 Розмір структури

Розмір структури можна визначити за допомогою оператора **sizeof**.

Приклад:

```
int raz = sizeof(studern2);
```

Оголосити структуру можна і всередині головної функції.

1.9.6 Масиви структур

У програмі можна оголосити не тільки окремий об'єкт структури, а і масив об'єктів.

Синтаксис оголошення масиву об'єктів:

Ім'я_структури ім'я_масиву[кіл_елементів];

Приклад:

```
Person p[20];
```

Доступ до елементів масиву здійснюється наступним чином:

```
p[0].vozvrast = 19;
p[0].rost = 1.68;
strcpy(p[0].name,"Женя");
```

1.9.7 Структура в структурі

Можна оголосити одну структуру в іншій структурі.

Приклад:

```
struct Dot{
```

```
    float x;
    float y;
};
struct Rectan{
    Dot a;
    Dot b;
    char color[10];
};
```

Використати такі структури можна наступним чином:

```
Rectan r;
strcpy(Rectan.color,"red");
r.a.x = 10;
r.a.y = 20;
r.b.x = 100;
r.b.y = 200;
```

1.9.8 Вказівник на структуру

Можна оголосити не об'єкт структури, а вказівник на об'єкт.

Приклад:

```
Person *p;
```

При цьому створюється вказівник на об'єкт, але сам об'єкт ще не існує, бо під нього не відведена пам'ять. Відвести пам'ять можна за допомогою оператора **new**.

```
p = new Person;
```

Доступ до елементів об'єкта в цьому випадку здійснюється з допомогою операції ->

Приклад:

```
p->rost = 1.68;
p->vozvras = 21;
strcpy(p->name,"Аня");
```

Вказівнику можна присвоїти адресу вже існуючого об'єкта структури.

Приклад:

```
Person *p;
Person pers;
p = &pers;
```

1.9.9 Структура як аргумент функції

Об'єкт структури можна передавати у функцію в якості параметра. Передача параметра може бути за значенням або за посиланням.

Приклад передачі структури за значенням:

```
void outPerson(Person p){
   cout << "Ім'я " << p.name;
   cout << "Вік " << p.vozvrast;
   cout << "Зріст "<<p.rost;
}
void main()
{
   Person friend;
   friend.vozvrast = 22;
   friend.rost = 1.68;
   strcpy(friend.name,"Женя");
   outPerson(friend);
}
```

Недоліки такого способу: зайвий час та зайве використання пам'яті – передається копія об'єкта, який може бути досить великим за обсягом.

Приклад передачі структури за посиланням та з використанням вказівника:

```
void vv1(Person &p)
{
   cin >> p.rost;
   cin >> p.vozvrast;
}
void vv2(Person *p)
{
   cin >> p->rost;
   cin >> p->vozvrast;
}
void main()
{
   Person p1,p2;
   vv1(p1);
   vv2(&p2);
}
```

Переваги такого способу: передається тільки адреса об'єкта, не витрачається зайвий час та пам'ять. Але в цьому випадку, при зміні об'єкта у функції, ці зміни будуть збережені. Якщо треба заборонити зміну об'єкта, треба використати в оголошені параметру ключове слово **const**.

Приклад:

```
void vv1(const Person &p);
```

Виклик функції при цьому не зміниться.

1.9.10 Контрольні питання

1. Синтаксис оголошення структури.
2. Як створити об'єкт структури?
3. Яким чином можна звернутись до полів структури?
4. Як створити вказівник на об'єкт структури?
5. Яким чином можна звернутись до полів структури, якщо створено вказівник на об'єкт структури?

1.9.11 Завдання для самостійної роботи

1. Напишіть фрагмент коду, який демонструє оголошення структури та створення об'єкта структури.
2. Напишіть фрагмент коду, який демонструє оголошення структури з одночасним створенням об'єкта та його ініціалізацією.
3. Напишіть фрагмент коду, який демонструє передачу в функцію об'єкта структури за значенням.
4. Напишіть фрагмент коду, який демонструє передачу в функцію об'єкта структури за посиланням.

1.10 Об'єднання

1.10.1 Визначення об'єднання

Об'єднання служать для розміщення в одній і тій же області пам'яті за тою самою адресою даних різних типів.

Синтаксис оголошення об'єднання:

union <ім'я_об'єднання>
{
    тип_змінна ім'я_змінна_1;
    тип_змінна ім'я_змінна_2;
    . . .
    тип_змінна ім'я_змінна_n;
}

Приклад:

```
union D
{
   int x;
   float y;
   char s[10];
}
```

Приклад оголошення змінної:

```
D data;
```

Приклад доступу до полів:

```
data.x = 10;
cout << data.x;
strcpy(data.s,"Hello");
cout << data.s;
```

Програміст сам повинен стежити за типом даних у цей момент.

Приклад:

```
data.x = 10;
data.x++;
cout << data.x;
strcpy(data.s,"Hello");
sata.x++;                  // ПОМИЛКА
cout << data.x;
```

1.10.2 Контрольні питання

1. Синтаксис оголошення об'єднання.
2. Наведіть приклад оголошення та ініціалізації об'єднання.

3. Яким чином можна звернутись до полів об'єднання (за іменем, за вказівником)?

4. Як дані розташовуються в пам'яті, якщо вони є елементами об'єднання?

5. У чому полягає відмінність між типами даних структури та об'єднання.

6. Як визначити скільки пам'яті займає об'єднання?

1.10.3 Завдання для самостійної роботи

1. Напишіть фрагмент коду, який демонструє оголошення об'єднання.

2. Напишіть фрагмент коду, який оголошує та ініціалізує об'єднання, що має чотири поля – тип трикутника та довжини його сторін. Визначити площу трикутника.

РОЗДІЛ 2
Бібліотечні функції вводу-виводу

Засоби вводу/виводу безпосередньо до мови С++ (так само як і С) не входять. У програмах на мові С++ можна рівноправно використовувати дві бібліотеки вводу-виводу: стандартну бібліотеку С (форматований ввід-вивід) та бібліотеку класів, створену спеціально для С++ (потоковий ввід-вивід).

2.1 Функції форматованого введення-виведення

Для форматованого введення-виведення використовується стандартна бібліотека функцій введення-виведення:

#include <stdio.h>

2.1.1 Функція printf

Для виведення даних у вікно програми використовується стандартна функція **printf**():

printf(<рядок_опису_форматів> [, <список_даних>]);

Функція **printf**() повертає кількість виведених символів. Значення **EOF** повідомляє про помилку.

Формати, що записані в рядку опису формату, відповідають значенням зазначеним у списку даних: перший формат відповідає першому значенню у списку даних, другий формат – другому значенню і т. п.

Кількість специфікацій формату має дорівнювати кількості виводимих змінних.

Специфікація формату:

% [прапорець] [ширина] [.точність] [h|l|L] символ_формату

У **[квадратних дужках]** вказані необов'язкові поля.

Таблиця 6

Прапорець	Призначення
-	притискає виводиме значення до лівої границі виділеного поля (за замовченням – до правої)
+	обов'язковий режим виведення знаку (навіть для позитивних значень)
проміжок	для позитивних чисел замість знака "+" виводиться проміжок

Прапорець	Призначення
0	встановлює режим виводу, при якому старші незначущі позиції заповнюються нулями
#	впливає на формат чисел у восьмирічній (префікс 0), шістнадцятирічній (префікс **0x** або **0X**) системах числення та дійсних чисел; для форматів **f**, **e**, та **E** – як роздільник між цілою та дробовою частинами відображається десятична крапка (якщо дробова частина 0 – відображається лише ціла частина); для форматів **g** та G завжди відображається десятична крапка.

Ширина – мінімальна кількість позицій, призначених для відображення даних. Якщо ширина менша за виводиме значення, дані будуть виведені у повному обсязі.

Точність – кількість позицій, що відводяться для відображення дробової частини числа. Для цілочисельних значень поле задає обов'язкову кількість цифр – якщо виводиме значення менше за точність, то перед його старшим розрядом додається необхідна кількість нулів. Для дійсних чисел поле точність визначає кількість цифр у дробовій частині числа.

Приклад:

Виводиме значення	Формат	Результат
тип int число –5	%8.4d	–0005
тип float число 3.14159265	%10.4f	3.1416

h, l, L – модифікатори, що визначають тип аргументів

Таблиця 7

h	тип **short** або **unsigned short**
l	тип **long** або **unsigned long**
L	тип **long double**

Таблиця 8

Символи формату

Символ формату	Тип виводимого значення
c, C	**char**; одиничний літерал (символ)
d, i	**int**; цілі зі знаком
o	**int**; без знакові цілі у восьмирічній системі числення
u	**int**; цілі зі знаком
x, X	**int**; беззнакові цілі у шістнадцятирічній системі числення
f	**float**; дійсні числа типу **float** (фіксована крапка)
lf	**double**; дійсні числа типу **double** (фіксована крапка)
e, E	**double**; дійсні числа в експоненційній формі (плаваюча крапка)
g, G	**double**; дійсні числа
P	**void**; вказівник
%	знак **%**

У рядкових значеннях такі символи як апостроф, знак питання можуть набиратись і не використовуючи зворотній слеш. Керуючі символи можуть бути відображені як у символьному вигляді, так і представлені у восьмирічній або шістнадцятирічній системі:

\0xxx	відображення символу кодом **xxx** у восьмирічній системі;
\xhh	відображення символу кодом **hh** у шістнадцятирічній системі;
\xHH	відображення символу кодом **HH** у шістнадцятирічній системі;

Таблиця 9

Керуючі символи (*Escape* – послідовності)

Chr	Dec	Hex	Значення
'\a'	'\07'	'\x07'	Звуковий сигнал
'\b'	'\08'	'\x08'	Повернення на одну позицію назад (Backspace)
'\f'	'\14'	'\x0c'	Перехід на нову сторінку
'\n'	'\12'	'\x0a'	Перехід на новий рядок
'\r'	'\15'	'\x0d'	Повернення каретки (повернення на початок рядка)
'\t'	'\11'	'\x09'	Горизонтальна табуляція
'\v'	'\13'	'\x0b'	Вертикальна табуляція
'\\'	'\134'	'\x5c'	Зворотна коса риска
'\''	'\47'	'\x27'	Апостроф
'\"'	'\42'	'\x22'	Подвійні лапки
'\?'	'\77'	'\x77'	Знак питання

Приклад:

```
int k = -25;
unsigned int j = 7;
float x = -4.55;
double y = 3.125E-2;
```

оператором

```
printf("%d%u%f%lf", k, j, x, y);
```

буде надруковано наступний рядок

```
-257-4.5500000.031250
```

В управляючому рядку "**%d%u%f%lf**" формати розташовані без проміжків, відповідно на екрані і числа будуть надруковані без проміжків.

Приклад застосування функції **printf** ()

1. Використання тексту:

```
printf("k=%d j=%u x=%f y=%lf", k, j, x, y);
```

Результат

```
k=-25 j=7 x=-4.550000 y=0.031250
```

2. Використання пробілів:

```
printf("k=%d %u %f %lf", k, j, x, y);
```

Результат

```
k=-25 7 -4.550000 0.031250
```

3. Використання керуючих символів – знак табуляції (\t) :

```
printf("\n\t%d\t%u\t%f\t%lf\n", k, j, x, y);
```

Результат

```
-25    7    -4.550000      0.031250
```

4. Використання керуючих символів – перехід на новий рядок (\n) :

```
printf("%d\n%u\n%f\n%lf\n", k, j, x, y);
```

Результат

```
-25
7
-4.550000
0.031250
```

5. Виведення числа в експоненційній формі:

```
printf("Y = %e", y);
```

Результат

```
Y = 3.125000e-002
```

6. Виведення числа зі встановленим флагом "-":

Без використання флагу:

```
printf("Y = %5.3e k=%10.5d", y, k);
```

Результат

```
Y = 3.125e-002 k=  -00005
```

З використанням флагу:

```
printf("Y = %5.3e k=%-10.5d", y, k);
```

Результат

```
Y = 3.125e-002 k=-00005
```

Функція **sprintf()** виводить інформацію не на стандартне обладнання виводу, а в символьний рядок BUFFER:

sprintf(BUFFER, <рядок_опису_форматів> [, <список_даних>]);

Приклад застосування функції **sprintf()**:

```
int k = 15;
char str[80];
sprintf (str,"мені %d років", k);
printf("%s", str);
```

Результат

```
мені 15 років
```

2.1.2 Функція scanf

Для форматованого виведення даних використовується функція **scanf():**

scanf(<рядок_опису_форматів> [, <список_даних>]);

Функція **scanf**() повертає кількість успішно зчитаних елементів.

Список параметрів **scanf()** складається із двох частин: рядка формату і списку даних. ***Рядок опису форматів*** відповідає специфікації формату визначеного для функції **printf** і визначає, яким чином повинні бути інтепретовані дані, що вводяться.

Список даних містить адреси змінних, до яких будуть записані значення що вводяться. Адрес змінної розраховується за допомогою унарної операції **&**. Кількість специфікацій формату має дорівнювати кількості змінних, зазначених у списку даних.

Після натискання <ENTER> дані передаються функції **scanf**() у вигляді набору символів. Функція визначає, які символи відповідають типу, заданому вказівником формату, а які слід ігнорувати. Функція ігнорує знаки: пробіли, символи табуляції, знаки нового рядка, крім тих випадків, коли тип визначений, як рядок.

Цифрові символи перетворюються в число і заносяться за адресою змінної. Якщо цифрові символи передують нецифровим, сканування триває, поки не зустрінеться нецифровий символ або пробіл. Якщо нецифрові символи передують цифровим, функція припиняє роботу і вся надрукована послідовність ігнорується. Якщо ввід здійснюється через буфер, то символи при цьому залишаться в буфері і будуть передані при наступному введенні.

При введенні рядка функція ігнорує пробіли, пробіл інтерпретується як роздільник (тобто буде зчитано значення до пробілу). Признак кінця рядка функція додає автоматично.

Функція забезпечує можливість зазначати множину символів дозволених для введення.

Приклад:

Вводиме значення	Формат
введення цифр тільки із діапазону від 0 до 9	%[0-9]
введення символів, що не є цифрами	%[^0-9]
введення тільки символів A, B і C	%[ABC]

Приклад застосування функції **scanf**():

```
int m, n, x;
double y;
char ch1 = '&';
```

1. Введення цілих

```
scanf("%d%d", &m, &n);
```

2. Введення дійсного числа

```
scanf("%lf", &y);
```

3. Введення символа

```
scanf("%c ", &ch1);
```

Функція **sscanf**() здійснює введення даних з масиву (рядка символів), що адресується параметром BUFFER:

sscanf(BUFFER, <рядок_опису_форматів> [, <список_даних>]);

Приклад:

```
int x;
float y;
sscanf("5 6.7", "%d%f", &x, &y);
printf("%d %f", x, y);
```

Результат

```
5 6.700000
```

2.1.3 Контрольні питання

1. Яка стандартна бібліотека застосовується при форматованому введенні-виведенні даних?
2. Запишіть специфікацію формату функції **printf**?
3. За що відповідають параметри **флаг, ширина** і **точність**? Чому вони вказані у квадратних дужках?
4. Як вказати кількість значущих розрядів цілої і дробової частини дійсного числа з фіксованою крапкою?
5. Для чого застосовуються керуючі символи (*Escape* – послідовності)? Наведіть приклади.
6. Який символ формату застосовується при введенні/виведенні: символа, цілого зі знаком, дійсного в експоненційній формі?
7. Як встановити діапазон дозволених вводимих символів для функції **scanf**?

2.1.4 Завдання для самостійної роботи

1. Напишіть фрагмент коду, в якому на друк виводиться дійсне з 3 розрядами чілої частини і 5 розрядами дробової частини, вирівняне по лівому краю.
2. Напишіть фрагмент коду в якому користувач може ввести ціле без знакове та лише символи "a, e, u, +", а на друк виводить декрементоване ціле у восьмирічній системі числення.
3. Напишіть фрагмент коду, в якому користувач вводить беззнакове ціле; на екран виводиться значення квадратного кореня введеного числа та символ, ASCII код якого відповідає введеному цілому.
4. Що буде виведено на екран, після виконання оператора:

а) `printf("\n\t%c\t%o%d \t%x\n\n%f\t%lf\n",k,j,x,y,o,f);`

б) `sscanf("5 a 0 6.7", "%d%c%o%f", &x, &y, &z, &c);`

5. Дослідити призначення і можливості функції **cprintf**.

РОЗДІЛ 3
Файли

3.1 Текстові файли

Текстові файли належать до файлів послідовного доступу. Текстові файли містять як символьні, так і числові дані. Одиниця виміру інформації в текстових файлах – рядки довільної довжини. Кожний рядок закінчується парою символів 0D0A – "повернення каретки" та "перехід на новий рядок".

3.1.1 Ініціалізація та відкриття текстового файлу

Для ініціалізації текстового файлу необхідно створити вказівник на структуру файлу типу **FILE** та відкрити його за допомогою функції **fopen** у відповідному режимі.

ФУНКЦІЯ fopen

Функція **fopen**() відкриває файл, ім'я якого задається першим параметром (**fname)**, і повертає вказівник на потік, пов'язаний з файлом:

FILE ***fopen**(const char *fname, const char *mode);

АБО

FILE ***fopen**(ім'я_файлу, "режим");

Таблиця 11

Значення параметру mode

Режим	Призначення
"r" або "rt"	Відкриває текстовий файл для зчитування даних
"w" або "wt"	Створює текстовий файл для запису даних
"a" або "at"	Дописує дані в існуючий текстовий файл

Максимальна кількість одночасно відкритих файлів задається параметром **FOPEN_MAX**, визначеним у файлі "**stdio.h**".

Приклад:

```
FILE *f1;          // оголошення вказівника на файл
// спроба відкрити файл для запису
if ((f1 = fopen("input.txt", "w")) == NULL)
 {
  printf("Не вдається відкрити файл.\n");
  exit(1);              // завершення з кодом помилки 1
 }
```

АБО

```
FILE *f1;
```

```
if ((f1 = fopen("input.txt", "w")) == NULL)
  {
   perror("Помилка відкриття! \n ");
   exit(1);
  }
```

Приклад запису шляху до файлу:

```
FILE *f1, *f2, *f3;
// f1 – вказівник на файл output.txt, який відкрито для зчитування
f1 = fopen("output.txt", "rt");
f2 = fopen("d:/my/file.cpp", "w");
// За правилами С
f3 = fopen("d:\\my\\file.cpp", "w");
```

3.1.2 Зчитування даних з текстового файлу

ФУНКЦІЯ fscanf

Функція **fscanf**() працює аналогічно до функції **scanf**(), але зчитує інформацію із потоку, заданого вказівником **stream**:

int fscanf(FILE *stream, const char *format, ...);

АБО

int fscanf(вказівник_на_потік, const char *format, ...);

При успішному завершенні функція повертає кількість аргументів, яким були присвоєні значення; якщо досягнут кінець файлу або виникла помилка – значення **EOF**; якщо не було записано жодного символу – нуль.

3.1.3 Запис даних до текстового файлу

ФУНКЦІЯ fprintf

Функція **fprintf**() виводить дані в потік, що адресується вказівником **stream.** Решта параметрів аналогічна параметрам функції **printf()**:

int **fprintf**(FILE *stream, const char *format, ...);

АБО

int **fprintf** (вказівник_на_потік, const char *format, ...).

При успішному завершенні функція повертає кількість записаних символів. При помилці – від'ємне число.

3.1.4 Закриття файлу

ФУНКЦІЯ fclose

Функція **fclose**() закриває файл, пов'язаний із потоком **stream:**

int **fclose**(FILE *stream);

При успішному завершенні функція **fclose**() повертає нуль; в іншому випадку – значення **EOF**.

Помилка виникає і при закритті вже закритого файлу.

Приклад програми, що записує до файлу числа від 1 до 10 та їх квадрати; після чого зчитує дані з файлу і виводить на екран:

```
#include <windows.h>
#include "math.h"
int _tmain(int argc, _TCHAR* argv[])
{
   FILE *f;                    // оголошення вказівника на файл
   int j, k;
   double d;
   char name_f[]="input.txt"; //ім'я файлу
//відкриття файлу з іменем name_f для запису
   f = fopen(name_f,"w");
   for( j = 1; j < 6; j++)
      {
//запис даних (цілого та дійсного чисел) до файлу
        fprintf(f, "%d %lf\n",j, sqrt(double(j)));
//вивід на екран
        printf("%d %lf\n", j, sqrt(double(j)));
      }
   fclose(f);                  //закриття файлу
   f = fopen(name_f,"rt");     //відкриття файлу для читання
   for(j = 1; j < 6; j++)
      {
// зчитування даних (цілого k та дійсного d) з файлу
        fscanf(f,"%d %lf", &k ,&d);
        printf("%d %lf\n", k, d);
      }
   fclose(f);                  // закрити файл
   system("pause");
   return 0;
}
```

Результат

```
Запис до файлу
1 1.000000
2 1.414214
3 1.732051
4 2.000000
5 2.236068
```

```
Зчитування з файлу
1 1.000000
2 1.414214
3 1.732051
4 2.000000
5 2.236068
```

3.1.5 Додаткові функції роботи із файлами

Функція **feof**() перевіряє чи був досягнутий кінець файлу, що пов'язаний з потоком **stream:**

int **feof** (FILE * stream);

При успішному завершенні функція **feof**() повертає ненульове значення; в іншому випадку – значення 0.

Функція **remove**() видаляє файл з ім'ям **fname:**

int **remove** (char * fname);

При успішному завершенні функція **remove**() повертає нуль; в іншому випадку – макрос **ERRNO** з кодом помилки.

Функція **rename**() переіменовує файл з ім'ям **old** у файл з ім'ям **new:**

int **rename** (char *old, char *new);

При успішному завершенні функція **rename**() повертає нуль; в іншому випадку – макрос **ERRNO** з кодом помилки.

3.1.6 Контрольні питання

1. Яке представлення даних у текстових файлах? Чим являє собою інформація, що в них зберігається.
2. Яку послідовність дій треба виконати, щоб відкрити текстовий файл для читання/запису?
3. Яка функція застосовується для відкриття файлів?
4. Які значення може приймати параметр **mode** функції **fopen**?
5. Скільки одночасно може бути відкрито файлів?
6. Які функції застосовуються для запису/зчитування даних з текстових файлів? Наведіть приклади застосування.
7. Яка функція застосовується для закриття файлів? Що ця функція повертає при виникненні помилки?
8. Як обробити помилку при відкритті файлу?

3.1.7 Завдання для самостійної роботи

1. Напишіть фрагмент коду, в якому відкривається файл для дозапису даних, з можливістю обробляти помилку відкриття файлу.

2. Напишіть фрагмент коду, в якому з файлу **in.txt** зчитуються три значення: символ, ціле, дійсне (записані в рядок). У файл **out.txt** зчитані значення виводяться кожне в окремому рядку.

3. Напишіть фрагмент коду, в якому зчитуються з текстового файлу **in.txt** рядки наступного формату:

```
число знак_операції число
```

Вивести в файл **out.txt** рядок формату :

```
число знак_операції число = результат_операції
```

Кількість рядків невідома. В якості знаку операції розглянути: +, -, *.

3.2 Двійкові файли

Двійкові (бінарні) файли належать до файлів прямого доступу. Інформація в таких файлах зберігається у вигляді послідовності байтів. Двійкові файли містять дані різної природи: байти графічної, текстової, числової, аудіо та іншої інформації. Одиниця виміру інформації в двійкових файлах – порції байтів зазначеної довжини.

Двійкові файли як і текстові строюються за допомогою функції **fopen**(). При цьому використовується один із наступних режимів роботи із файлом.

Таблиця 12

Значення параметру mode

Режим	Призначення
"rb"	Відкриває двійковий файл для зчитування даних
"wb"	Створює двійковий файл для запису даних
"ab"	Дописує дані в існуючий двійковий файл

3.2.1 Зчитування даних з двійкового файлу ФУНКЦІЯ fread

Функція **fread**() зчитує з потоку **stream**, **count** об'єктів (порцій даних) довжиною **size** байт до масиву **buf**:

size_t **fread**(void *buf, size_t size, size_t count, FILE *stream);

Функція повертає кількість зчитаних об'єктів.

Приклад зчитування з файлу 5 чисел типу **float**:

```
FILE *f1;                  // оголошення вказівника на файл
float b[5] = { 1.1, 2.2, 3.3, 4.4, 5.5 };
//  вказівник f1 посилається на двійковий файл з іменем file
f1 = fopen("file", "rb");
//  якщо вдалося зчитати блок з 5-ти дійсних чисел
if (fread(b, sizeof(float), 5, f1) != 5)
{
   if (feof(f1))          // перевірка чи досягнутий кінець файлу
       printf("Достроковий кінець файлу.");
   else printf("Помилка відкриття файлу.\n");
}
```

3.2.2 Запис даних до двійкового файлу ФУНКЦІЯ fwrite

Функція **fwrite**() записує в потік **stream**, **count** об'єктів (порцій байт) довжиною **size** з масиву **buf**:

size_t **fwrite**(const void *buf, size_t size, size_t count, FILE *stream);

Функція повертає кількість записаних до файлу об'єктів.

Приклад запису до файлу 1 числа типу **float**:

```
FILE *f1;
float f = 34.3;
f1 = fopen("file", "wb");
fwrite(&f, sizeof(float), 1, f1);
```

3.2.3 Функция fseek

Функция fseek() встановлює вказівник поточної позиції файлу, пов'язаного з потоком **stream**, у відповідність до значення параметрів **delta** і **pos**:

fseek(FILE *stream,delta,pos);

delta – величина зсуву у байтах, на значення якого необхідно перемістити вказівник файлу;

pos – позиція, відносно якої відбувається зсув вказівника:

Константа	Числове значення	Напрямок руху
SEEK_SET	0	від початку
SEEK_CUR	1	від поточної позиції
SEEK_END	2	від кінця файлу

3.2.4 Структуровані файли

Структурований файл – це окремий випадок двійкового файлу. В якості порції даних в таких файлах використовують структури.

Для ініціалізації, відкриття, зчитування та запису даних до структурованих файлів застосовуються ті ж самі функції, які використовуються при роботі з двійковими файлами.

Структуровані файли дозволяють за одне звернення записати/зчитати всі поля структури, що значно пришвидшує роботу з файлом.

Приклад запису до файлу структури, що містить 2 поля – ціле (b.n, 4 байта) і дійсне (b.r, 4 байта):

```
#include <stdio.h>
#include <windows.h>
#include "math.h"

int _tmain(int argc, _TCHAR* argv[])
{
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
FILE *f1;                    //оголошення вказівника на файл
struct {                           //опис структури даних
```

```
      int n;
      float r;
     } b;
//  відкриття файлу для запису
   f1 = fopen("file1","wb");
   printf("Запис до файлу");
   for (int j = 1; j < 6; j++)
    {
// ініціалізація полів структури
          b.n = j;
          b.r = sqrt(float(j));
// запис 1 елементу структури до файлу
      fwrite(&b, sizeof(b), 1, f1);
          printf("\n%d %f", b.n, b.r);
    }
   fclose(f1);                              // закрити файл
   printf("\n");
//  відкриття файлу для зчитування
   f1 = fopen("file1","rb");
   printf("Зчитування з файлу\n");
   int k = 0; // 0 – для 1-го блоку
   while (!feof(f1)) // доки не досягли кінця файлу
      {
// встановити вказівник перед зчитуваним блоком
          fseek(f1, (k)*sizeof(b), SEEK_SET);
// зчитати блок даних розміром sizeof(b) – 1 елемент структури b
          if (fread(&b, sizeof(b), 1, f1))
// вивести результат на екран
                printf("%d %f\n", b.n, b.r);
                k+=2;                       // пропустити запис
      }
   system("pause");
   return 0;
   }
```

Результат

```
Запис до файлу
1 1.000000
2 1.414214
3 1.732051
4 2.000000
5 2.236068
```

```
Зчитування з файлу
1 1.000000
3 1.732051
5 2.236068
```

3.2.5 Контрольні питання

1. Як дані зберігаються у двійкових (бінарних) файлах?
2. Які значення може приймати параметр **mode** функції **fopen** при роботі з бінарними файлами?
3. Які функції застосовуються для запису та зчитування даних у бінарних файлах? Наведіть їх синтаксис та приклади застосування.
4. Призначення та аргументи функції **fseek**?
5. Що використовується в якості порції даних в структурованих файлах?

3.2.6 Завдання для самостійної роботи

1. Продемонструйте в коді як за 1 команду зчитати з файту 10 цілих чисел; записати до файлу значення масиву, що містить 20 дійсних чисел.
2. Продемонструйте в коді як встановити вказівник на 5-тий, передостанній елемент файлу; перемістити вказівник на три позиції вперед від поточного значеня.
3. Напишіть фрагмент коду, в якому з бінарного файлу **in** зчитується 10 цілих. В файл **out** записується значення середнього арифметичного введених чисел і виводиться рядок формату:

```
Середнє арифметичне = значення_ср_арифметичного
```

4. Напишіть фрагмент коду, в якому зі структурованого файлу **in** зчитується інформація про 10 людей: їх прізвище та вік. У файл **out** записати прізвище та вік наймолодшого.

СПИСОК ЛІТЕРАТУРИ

1. Глушаков С. В. Язык программирования С++ : учебный курс / С. В. Глушаков, А. В. Коваль, С. В. Смирнов ; худож. оформитель А. С. Юхтман. – Харьков : Фолио ; М. : ООО "Издательство АСТ", 2001. – 500 с.
2. Мейерс С. Наиболее эффективное использование С++. 35 новых рекомендаций по улучшению ваших программ и проектов / С. Мейерс ; пер. с анг. – М. : ДМК "Пресс", 2000. – 304 с.
3. Страуструп Б. Программирование : Принципы и практика с использованием С++ : 2-изд. / Б. Страуструп ; пер. с англ. – М. : ООО "И. Д. Вильямс", 2016. – 1328 с.
4. Страуструп Б. Язык програмирования С++ : В 2-х ч. / Б. Страуструп ; пер с англ. – К. : "ДиаСофт", 1993. – 294 с.
5. Шилдт Г. Полный справочник по С++ / Г. Шилдт ; пер с англ. – М. : "И. Д. Вильямс", 2006. – 800 с.
6. Шилдт Г. Теория и практика С++ / Г. Шилдт ; пер с англ. – СПб. : BHV-Санкт-Петербург, 1996. – 416 с.
7. Эккель Б. Филисофия С++. Введение в стандартный С++ : 2-е изд. / Б. Эккель. – СПб. : Питер, 2004. – 572 с.
8. Эккель Б. Практическое программирование / Б. Эккель, Ч. Эллисон. – СПб. : Питер, 2004. – 608 с.

ДЛЯ НОТАТОК

Навчальне видання

С. Ю. Боровльова,
А. В. Швед

БАЗОВИЙ С++

Навчальний посібник

Редактори, технічні редактори *Д. Стригіна.*
Комп'ютерна верстка, дизайн обкладинки *А. Іщенко.*
Друк, фальцювально-палітурні роботи *С. Волинець.*

Підп. до друку 24.04.2017.
Формат 60 × 84 $^{1}/_{16}$. Папір офсет.
Гарнітура «Times New Roman». Друк ризограф.
Ум. друк. арк. 6,74. Обл.-вид. арк. 3,93.
Тираж 300 пр. Зам. № 5165.

54003, м. Миколаїв, вул. 68 Десантників, 10.
Тел.: 8 (0512) 50-03-32, 8 (0512) 76-55-81, e-mail: rector@chmnu.edu.ua.
Свідоцтво суб'єкта видавничої справи ДК № 3460 від 10.04.2009.