Міністерство освіти і науки України
Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»

# Інформатика. Частина 2.

## Програмування та алгоритмічні мови

**Конспект лекцій**

Рекомендовано Методичною радою КПІ ім. Ігоря Сікорського
як навчальний посібник для здобувачів ступеня бакалавра
за освітніми програмами «Електронні компоненти і системи»,
«Електронні прилади та пристрої»
спеціальності G5 Електроніка, електронні комунікації, приладобудування та радіотехніка

Укладачі: К. С. Клен, О. М. Коваленко, О. О. Абакумова

Електронне мережеве навчальне видання

Київ
КПІ ім. Ігоря Сікорського
2026

УДК 621.314
К37

Укладачі: *Клен Катерина Сергіївна*, д-р техн. наук, доц.
*Коваленко Олександр Миколайович*
*Абакумова Олена Олегівна*, канд. філос. наук, доц.

Рецензент *Найда Сергій Анатолійович*, д-р техн. наук, проф.
КПІ ім. Ігоря Сікорського

Відповідальний редактор *Вербицький Євген Володимирович*, д-р техн. наук, проф.

*Гриф надано Методичною радою КПІ ім. Ігоря Сікорського*
*(протокол № 8 від 04.06.2026 р.)*
*за поданням вченої ради факультету електроніки*
*(протокол № 05/2026 від 27.05.2026 р.)*

К37 **Інформатика. Частина 2.** Програмування та алгоритмічні мови [Електронний ресурс] : конспект лекцій : навч. посіб. для здобувачів ступеня бакалавра за освіт. програмами «Електронні компоненти і системи», «Електронні прилади та пристрої» спец. G5 Електроніка, електронні комунікації, приладобудування та радіотехніка / КПІ ім. Ігоря Сікорського ; уклад.: К. С. Клен, О. М. Коваленко, О.О. Абакумова. – Електрон. текст. дані (1 файл). – Київ : КПІ ім. Ігоря Сікорського, 2026. – 313 с.

У навчальному посібнику викладено інформацію щодо основ програмування мовою С.++

Конспект лекцій містить теоретичні відомості до 15 лекцій з контрольними запитаннями та список використаної літератури.

Навчальний посібник призначений для здобувачів ступеня бакалавра за освітніми програмами «Електронні компоненти і системи», «Електронні прилади та пристрої» спеціальності G5 Електроніка, електронні комунікації, приладобудування та радіотехніка.

УДК 621.314

Реєстр. № НП 25/26-440. Обсяг 14,23 авт. арк.

Національний технічний університет України
«Київський політехнічний інститут імені Ігоря Сікорського»
проспект Берестейський, 37, м. Київ, 03056
https://kpi.ua
Свідоцтво про внесення до Державного реєстру видавців, виготовлювачів
і розповсюджувачів видавничої продукції ДК № 5354 від 25.05.2017 р.

# ЗМІСТ

## Вступ

Навчальна дисципліна «Інформатика. Частина 2. Програмування та алгоритмічні мови» належить до циклу професійної та практичної підготовки бакалаврів за спеціальністю G5 Електроніка, електронні комунікації, приладобудування та радіотехніка, освітніми програмами «Електронні компоненти і системи», «Електронні прилади та пристрої». Дисципліна читається протягом одного семестру (2) і є базовою для подальшого навчання.

В процесі вивчення курсу студенти вивчають основи програмування мовою С++, яка призначена для розробки високопродуктивного програмного забезпечення та дуже популярна серед програмістів. Вивчивши С++, студенти отримують фундаментальні знання, які дозволять їм у подальшому опанувати будь-які аспекти сучасного програмування, навички проєктування та відлагодження пристроїв на мікроконтролерах та інших приладів та систем сучасної електроніки.

Метою навчальної дисципліни є передача студентам знання прикладних основ розробки програмного забезпечення сучасними програмними засобами мови С++; формування у студентів досвіду практичного використання програмних засобів обчислювальної техніки та програмування з використанням сучасних можливостей мови С++ у відповідності до професійних потреб; розробка прикладного програмного забезпечення для розв’язання інженерних завдань та для мікропроцесорних систем електроніки.

## Лекція №1. Змінні і основні типи даних

Пізнавши історію появи мов С та С++, легше зрозуміти концепції, що лежать в їх основі, а також відповісти на питання, чому протягом вже не одного десятиріччя мова С залишається популярною серед програмістів, а її більш молодша «родичка» С++ не поступається їй у популярності.

У 1972 році співробітник фірми Bell Laboratories Деніс Рітчі створив нову алгоритмічну мову – С. В її основу було закладено багато особливостей мови Assembler. Мова С є універсальною, придатною для розв'язання будь-якого типу задач, хоча спочатку була задумана як мова системного програмування (у 1973 році на мові С Деніс Рітчі реалізував операційну систему Unix) [1 – 3].

У середині 80-х років Б'ярн Страуструп розробив мову «С з класами», що надалі стали називати мовою С++. Ця мова дозволяє працювати не тільки зі змінними, але і з їх адресами, розміщувати дані як у пам'яті, так і в регістрах, використовувати непряму адресацію (задання адреси комірки, в якій зазначена адреса даного), автоматично змінювати адресу. Об'єктний код, що формується компіляторами мови С++, займає приблизно стільки ж пам'яті, скільки і відповідна програма на Assembler.

Мову С++ можна розглядати як надмножину мови С, бо вона зберігає усі можливості, що надає мова С, і доповнює їх засобами об'єктно-орієнтованого програмування. С++ є універсальною алгоритмічною мовою, яка використовується для розробки системних та складних прикладних програм. Це не тільки найпоширеніша мова програмування, але й мова спілкування програмістів, оскільки більшість програм алгоритмів написані на С++.

Мова С++ є мовою високого рівня і основою багатьох систем програмування: Borland C++, Visual C++, Borland C++ Builder. Найбільш популярною з них вважається Borland C++ Builder. За допомогою цієї системи візуального об'єктно-орієнтованого програмування як користувач-початківець, так і програміст-професіонал мають можливість створювати інтерфейс користувача до прикладних програм різноманітних класів, що виглядає однаково професійно.

## Алфавіт, лексеми, синтаксис мови

У природній мові спілкування виділяють чотири основні елементи: символ, слово, словосполучення та речення. Подібні елементи існують і в алгоритмічній мові, тільки слова мають назву лексеми, словосполучення – вирази, а речення – оператори. Лексеми створюються із символів, вирази – із лексем та символів, оператори – з символів, виразів і лексем.

Алфавіт мови C++ включає [1 – 4]:

- великі (A – Z) і малі (a – z) літери латинського алфавіту та символ підкреслення ( _ );

- арабські цифри від 0 до 9;

- знаки арифметичних дій +, -, *, /, %, ++, --;

- знаки побітових операцій <<, >>, &, |, ~, ^;

- знаки відношень <=, ==, !=, >, >=;

- знаки логічних операцій &&, ||, !;

- розділові знаки , ; : пропуск;

- спеціальні знаки ., =, ->, ?, \, $, #, ', ";

- символи дужок (, ), [, ], {, }.

Інші символи, а також літери кирилиці не використовуються для побудови базових елементів мови або для їх розділу, але вони можуть застосовуватись у символьних константах та коментарях.

**Лексеми**, тобто базові елементи мови з певним самостійним значенням, складаються із символів алфавіту. До них відносять ідентифікатори, ключові слова, знаки операцій, константи, роздільники (дужки, крапка, кома, символи пропуску). Межі лексем визначаються іншими лексемами-роздільниками або знаками операцій.

**Ідентифікатором**, тобто ім’ям програмного об’єкта, називають будь-яку послідовність літер латинського алфавіту, цифр і символу підкреслення за умови, що першою стоїть літера або символ підкреслення, а не цифра. Існує два різновиди ідентифікаторів:

- стандартні, наприклад, імена всіх вбудованих у мову функцій;

- користувальницькі.

Характерно, що мова С++ чутлива до регістру літер, тому компілятор розпізнає великі і малі літери латинського алфавіту як різні символи. Це дає можливість створювати ідентифікатори, що однаково читаються, але відрізняються написом одного або декількох символів. Наприклад, ідентифікатори «Sigma», «sigma» і «sigMa» вважаються різними.

Ідентифікатори можуть мати будь-яку довжину, але значимими є не більше 31 символу від початку ідентифікатора, а в деяких компіляторах це обмеження ще більш суворе (не більше 8 символів). Імена програмних об’єктів створюються на етапі оголошення даних, після цього їх можна використовувати в різних операторах програми.

**Ключовими (службовими) словами** називають ряд зарезервованих ідентифікаторів, що вживаються для побудови конструкцій мови і мають фіксоване значення. За смисловим навантаженням службові слова поділяються на такі основні групи:

- специфікатори типів – char, int, long, typedef, short, float, double, enum, struct, union, signed, unsigned, void;

- кваліфікатори типів – const і volatile;

- класи пам’яті – auto, extern, register, static;

- для побудови операторів – for, while, do, if, else, switch, case, continue, goto, break, return, default, sizeof.

У табл. 1.1 наведено список основних ключових слів мови С++.

Таблиця 1.1 Ключові слова С++

| | | | | |
|---|---|---|---|---|
| asm | delete | goto | register | throw |
| auto | do | if | return | try |
| break | double | inline | short | typedef |
| case | else | int | signed | typename |
| catch | enum | long | sizeof | union |
| char | explicit | new | static | unsigned |
| class | extern | operator | struct | virtual |
| const | float | private | switch | void |
| continue | for | protected | template | volatile |
| default | friend | public | this | while |

Як роздільники лексем застосовують такі символи: пропуск, табуляція, символ нового рядка, коментар. Між будь-якими двома лексемами допускається довільна кількість символів-роздільників. Крім того, деякі лексеми («*», «+», «,», « », «(», «->» тощо) самі є роздільниками і відділяти їх від інших лексем символами-роздільниками необов’язково.

**Типи даних**

Обробка даних різного типу є головною метою будь-якої програми. Кожне з даних характеризується класом пам’яті, ім’ям, типом і значенням. Імена дозволяють ідентифікувати дані, тобто відрізняти їх між собою. Програміст обирає тип кожної величини, що використовується для подання реальних об’єктів. Тип задає множину можливих значень даних і способи їх зберігання, перетворення та використання.

Обов’язкове оголошення типу даних дозволяє компілятору робити перевірку допустимості різних конструкцій програми.

Усі типи даних мови С++ можна розділити на основні (базові) і складені [2]. Основні типи визначені для представлення цілих, дійсних, символьних і логічних даних. На основі цих типів вводиться опис складених типів, до яких належать масиви, перелічення, функції, структури, посилання, вказівники, об’єднання і класи.

Основні типи даних (див. табл. 1.2) часто називають арифметичними, тому що їх можна використовувати в арифметичних операціях. Для опису основних типів мови С++ використовують такі службові слова:

- int (цілий);
- char (символьний);
- bool (логічний);
- float (дійсний);
- double (дійсний з подвійною точністю);
- void (порожній, не має значення).

Типи int, char, bool називають цілими, а типи float та double – дійсними з плаваючою крапкою. Код, що формує компілятор для обробки цілих величин, відрізняється від коду для величин з плаваючою крапкою.

Для уточнення внутрішнього подання та діапазону значень стандартних типів мова C++ використовує чотири специфікатори типу:

- short (короткий);
- long (довгий);
- signed (знаковий);
- unsigned (беззнаковий).

Таблиця 1.2. Базові типи даних для ПК (платформа Intel)

| **Тип** | **Розмір, байт** | **Значення** |
|---|---|---|
| bool | 1 | true або false |
| unsigned short int | 2 | від 0 до 65 535 |
| short int | 2 | від -32 768 до 32 767 |
| unsigned long int | 4 | від 0 до 4 294 967 295 |
| long int | 4 | від -2 147 483 648 до 2 147 483 647 |
| int (16 розрядів) | 2 | від -32 768 до 32 767 |
| int (32 розряди) | 4 | від -2 147 483 648 до 2 147 483 647 |
| unsigned int (16 розрядів) | 2 | від 0 до 65 535 |
| unsigned int (32 розряди) | 4 | від 0 до 4 294 967 295 |
| char | 1 | від 0 до 255 |
| float | 4 | від 1.2e-38 до 3.4e38 |
| double | 8 | від 2.2e-308 до 1.8e308 |
| long double | 10 | від 3.4e-4932 до 3.4e+4932 |

У табл. 1.2 наведено діапазони значень та розміри основних типів даних (для 16-розрядного і 32-розрядного процесорів). Розмір однакового типу даних може відрізнятися на комп’ютерах різних платформ, а також може залежати від застосованої операційної системи. Тому при оголошенні тієї чи іншої змінної потрібно чітко уявляти, скільки байт вона буде займати в пам’яті комп’ютера, щоб запобігти проблемам, пов’язаним з переповненням і неправильною інтерпретацією даних. Діапазони кожного з типів (див. табл. 1.2) повинні бути перевірені для конкретного комп’ютера.

## Змінні

Кожна програма потребує виконання різноманітних обчислень, для здійснення яких використовуються вирази, що складаються з операндів, знаків операцій і дужок. Операнди задають дані для обчислень, а операції задають дії, які необхідно виконати над цими даними. Операнд є, у свою чергу, виразом, що в окремому випадку може бути константою або змінною.

**Змінна** – це іменована область пам'яті, у якій зберігаються дані визначеного типу [5]. Змінна має ім'я, розмір та інші атрибути, такі як видимість, час існування тощо. Ім'я змінної служить для звертання до області пам'яті, у якій зберігається її значення. Перед використанням будь-яка змінна повинна бути описана, при цьому для неї резервується деяка область пам'яті, розмір якої залежить від конкретного типу змінної. Під час виконання програми змінна може приймати різні значення.

Наведемо загальний вигляд опису змінних:

[клас пам'яті] [const] тип ім'я [ініціювання];

де необов'язковий клас пам'яті може приймати одне зі значень – auto, extern, static чи register (у посібнику при описі синтаксису об'єктів програмування необов'язкові частини синтаксичних конструкцій мови подано у квадратних дужках «[ ]»); модифікатор const вказує, що змінна не може змінювати своє значення, у цьому випадку її називають типізованою (іменованою) константою або просто константою.

**Ініціювання** – це присвоювання змінній при описі початкового значення, яке записується зі знаком рівності – = значення або в круглих дужках – (значення). Зазначимо, що константа повинна бути ініційована при описі. Один оператор може містити опис декількох змінних одного типу, розділяючи їх комами, наприклад:

const int n = 20, m = 5, k = 4; – ініціювання констант n, m, k цілого типу;

float h = 17.5, d(5.5), sum; – опис дійсних змінних h, d, sum, ініціювання h і d;

char sf = 'f', st[ ] = "Мудрість лише в істині."; – ініціювання символьних змінних.

Якщо тип значення, що ініціюється, не збігається з типом змінної, то виконуються перетворення типу. Кожна змінна повинна мати своє ім’я, причому в одному блоці не може бути двох змінних з однаковим ім’ям.

**Областю дії ідентифікатора змінної** є частина програми, в якій його можна використовувати для доступу до зв’язаної з ним області пам’яті. Залежно від області дії змінна може бути локальною або глобальною.

**Локальна змінна** визначена всередині блока (нагадаємо, що блок розташований між фігурними дужками). Область її дії обмежена початком опису змінної та кінцем блока, включаючи усі вкладені блоки. Змінну, визначену поза будь-яким блоком, називають **глобальною**, і областю її дії вважається файл, у якому вона визначена від початку опису до його кінця.

**Клас пам’яті** визначає час існування та область видимості програмного об’єкта, тобто змінної. Якщо клас пам’яті не зазначений явно, то він визначається компілятором, виходячи, з контексту оголошення.

**Час існування змінної** може бути постійним (протягом виконання програми) і тимчасовим (протягом виконання блока).

**Областю видимості ідентифікатора** називають частину тексту програми, з якої можна здійснити звичайний доступ до зв’язаної з ідентифікатором області пам’яті. Найчастіше область видимості збігається з областю дії. Винятком є ситуація, коли у вкладеному блоці описана змінна з таким же ім’ям. У цьому випадку зовнішня змінна у вкладеному блоці невидима, хоча він і входить до її області дії. Проте до цієї змінної, якщо вона глобальна, можна звернутися, застосовуючи операцію доступу до області видимості – “::“.

Клас пам’яті задають такі специфікатори:

- **auto** – автоматична змінна, для якої пам’ять виділяється у стеку і за необхідності ініціюється кожного разу при виконанні оператора, що містить її визначення. Звільнення пам’яті відбувається при виході з блока, де описана змінна. Час її існування – з моменту опису до кінця виконання блока. Для глобальних змінних цей специфікатор не використовується, а для локальних він приймається за замовчуванням, тому задавати його явно великого сенсу немає;

- **extern** означає, що змінна визначена в іншому місці програми (в іншому файлі або далі по тексту) і використовується для створення змінних, доступних в усіх модулях програми, де вони оголошені. При ініціюванні змінної у тому ж операторі, специфікатор extern ігнорується;

- **static** – статична змінна, що має постійний час існування. Її ініціюють один раз при першому виконанні оператора, що містить визначення змінної. Залежно від розташування оператора, описані статичні змінні можуть бути глобальними і локальними. Глобальні статичні змінні видимі тільки у тому модулі, в якому вони описані;

- **register** – аналогічний до специфікатора auto, але пам'ять виділяється по можливості в регістрах процесора і за відсутності такої можливості у компілятора змінні обробляються як auto.

Наведемо фрагмент програми з використанням розглянутих вище понять:

```
int d;                  //1 – глобальна змінна d
int main()
{
    int b;              //2 – локальна змінна b
    extern int y;       //3 – змінна y визначена в іншому місці програми
    static int s;       //4 – локальна статична змінна s
    d = 1;              //5 – присвоєння значення глобальної змінної
    int d;              //6 – локальна змінна d
    d = 10;             //7 – присвоєння значення локальної змінної
    ::d = 3;            //8 – присвоєння значення глобальної змінної
    return 0;
}
int y = 4;              // 9 – визначення і ініціалізація змінної y
```

У цьому прикладі глобальна змінна d визначена поза всіма блоками. Пам'ять для неї виділяється в сегменті даних на початку роботи програми, областю дії є вся програма. Область видимості – вся програма, крім рядків 6-8, тому що в першому з них визначається локальна змінна з тим же ім'ям, область

дії якої починається з початку її опису і закінчується при виході з блока. Змінні b і s – локальні, область їх видимості – блок, але час існування різний: пам'ять під b виділяється в стеку при вході у блок і звільняється при виході з нього, а змінна s розташована у сегменті даних та існує увесь час роботи програми. Якщо початкове значення змінних явно не задається, компілятор присвоює глобальним і статичним змінним нульове значення відповідного типу. Автоматичні змінні не ініціюються. Початкове ініціювання змінних не є обов'язковим, проте все ж його бажано здійснювати.

Опис змінної може виконуватися у формі оголошення або визначення. Оголошення інформує компілятор про тип змінної і класи пам'яті, а визначення містить, крім цього, вказівку компілятору про виділення пам'яті відповідно до типу змінної. У C++ більшість оголошень є одночасно і визначеннями (у наведеному вище програмному фрагменті тільки опис extern int y; є оголошенням, але не визначенням). Змінна може бути оголошена багаторазово, а визначена тільки в одному місці програми, оскільки оголошення тільки описує властивості змінної, а визначення зв'язує її з конкретною областю пам'яті.

Розглянемо далі основні типи змінних.

**Цілі змінні** (типу int, long, short) необхідні для збереження цілих значень і можуть бути знаковими і беззнаковими. Знакові змінні застосовують для подання як додатних, так і від'ємних чисел, при цьому один біт (найстарший) виділяється під знак. Для оголошення беззнакової змінної, тобто змінної, що приймає тільки додатні значення, необхідно використовувати ключове слово unsigned. За замовчуванням будь-який цілий тип вважається знаковим, і тому немає потреби у використанні ключового слова signed.

Символьний тип даних **char** застосовується у випадку, коли змінна містить інформацію про код ASCII або для побудови таких більш складних конструкцій, як рядки, символьні масиви тощо. Дані типу char також можуть бути знаковими і беззнаковими.

Змінна типу **bool** займає 1 байт і використовується, насамперед, у логічних операціях, тому що може приймати значення 0 (false – «неправда») або відмінне

від нуля (true – «істина»). У випадку перетворення до цілого типу true має значення 1.

Стандарт C++ визначає три типи даних для збереження дійсних значень змінних: float, double та long double (типи з плаваючою крапкою). Тип **float**, як правило, використовують для збереження не дуже великих дробових чисел.

Змінна типу **void** не має значення, оскільки множина значень цього типу порожня. Такі змінні необхідні для узгодження синтаксису. Тип void використовують для визначення функцій, що не повертають значення, для вказівки порожнього списку аргументів функції, а також як базовий тип для вказівників і в операції приведення типів.

Наприклад, якщо немає потреби у використанні поверненого значення функції, перед ім'ям функції ставлять тип void:

```
void minmax(int*x, int k, int*min, int&max);
```

## Константи

**Константи** являють собою фіксовані значення, що не можуть змінюватися впродовж виконання всієї програми [6].

Спосіб визначення кожної константи залежить від її типу. Константи мови C++ слід поділяти на літеральні та типізовані.

**Літеральна константа** – це лексема, що являє собою зображення фіксованого числового, рядкового або символьного значення. Такі константи бувають цілі, дійсні, символьні та рядкові (табл. 1.3).

Таблиця 1.3. Літеральні константи мови C++

| Константа | Формат | Приклади |
|---|---|---|
| Ціла | Десятковий: послідовність десяткових цифр (0, 1, 2, 3, 4, 5, 6, 7, 8, 9), що починається не з нуля, якщо це не число нуль | 9, 0, 217925 |
| | Вісімковий: нуль, за яким розташовані вісімкові цифри (0, 1, 2, 3, 4, 5, 6, 7) | 02, 050, 07245 |
| | Шістнадцятковий: 0x чи 0X, за яким розташовані шістнадцяткові цифри (0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F) | 0x1B9, 0X00FF |
| Дійсна | Десятковий: [цифри].[цифри] | 9.7, .001, 87. 0.7E6, .15e-3, 9.2, 920 e-2, |
| | Експоненціальний: [цифри][.][цифри]{E\|e}[+\|-][цифри] | 92.E-1, .92E1 |

| | | |
|---|---|---|
| Символьна | Один чи два символи, що подаються в апострофах | ‘A’, ‘ю’, ‘* ‘, ‘db’, ’\0’, ‘\n’, ’\012', ‘\x07\x07’ |
| Рядкова | Послідовність символів, що подаються в лапках | “RESULT”, “\t sum_s=\0x5\n” |

Цілі константи можуть бути десятковими, вісімковими та шістнадцятковими.

Довгі цілі константи (long) мають літеру l або L в кінці, наприклад: 32768L; 0777777l; 0XFL. Для завдання константи без знака (unsigned) застосовується літера u (U),наприклад 65535u. Довгі константи без знака записуються з використанням двох літер відразу: (ul, UL) або (lu, LU).

**Дійсні числа** у мовах програмування мають дві форми подання: десяткову (природну) та експоненціальну (показникову).

**Десяткова форма** дійсного числа – це звичайний десятковий формат запису дійсного числа, тільки частина дійсного числа відділяється від дробової крапкою, а не комою, наприклад 10.123, 1.0123, 1012.3, 0.0010123.

**Експоненціальна форма** дійсного числа використовується для запису дуже великих або дуже малих чисел, для яких задавати зайві нулі не зовсім зручно, наприклад: 1 .0123*$10^{20}$,1.0123*10-$^{10}$. У цій формі запису числа можна виділити такі основні характеристики: знак числа, мантису числа, знак порядку та порядок числа. Зазначені характеристики дійсного числа зберігаються у пам’яті комп’ютера. Число у показниковій формі може бути представлено, наприклад, так: 1.0123E-10.

Мантиса записується ліворуч від знака експоненти (E чи e), порядок – праворуч. Символ E (e) означає основу степеня 10, і компілятор розпізнає цей запис як форму представлення дійсного числа. Символ пропуску всередині числа не допускається, а для відділення цілої частини від дробової використовується не кома, а крапка. При додатних значеннях числа і мантиси знак «+» можна не вказувати.

Як десяткова, так і експоненціальна форми запису допускають відсутність або цілої частини, або дробової, але не двох одразу.

За замовчуванням всі дійсні константи мають тип double – подвійну точність, що найчастіше займає в пам’яті 64 біти, тобто 8 байтів. Але у випадку, якщо програміста не влаштовує тип за замовчуванням, його можна вказати явно за допомогою спеціальних літер. Так, додавши літеру f чи F, константі надають дійсний тип float зі звичайною точністю, наприклад, 8.5f. Якщо в представленні константи використовується літера L чи l, то вона має тип long double.

Зображення від’ємної цілої чи дійсної константи вважається константним виразом, що складається зі знака унарної операції зміни знака (-) та константи, наприклад: -273, -2730.e-1, -273L.

**Бінарні літерали і розділювач цифр в C++14**. В C++14 ми можемо використовувати бінарні (двійкові) літерали, додаючи префікс 0b:

```
#include <iostream>
int main()
{
   int bin(0);
   bin = 0b1;              // присвоюємо змінній бінарний літерал 0000 0001
   bin = 0b11;             // присвоюємо змінній бінарний літерал 0000 0011
   bin = 0b1010;           // присвоюємо змінній бінарний літерал 0000 1010
   bin = 0b11110000;       // присвоюємо змінній бінарний літерал 1111 0000
   return 0;
}
```

Оскільки довгі літерали читати важко, то в C++14 додали можливість використовувати одинарну лапку ( ' ) як розділювач цифр:

```
#include <iostream>
int main()
{
   int bin = 0b1011'0010;          // присвоюємо змінній бінарний літерал 1011 0010
   long value = 2'532'673'462;  // набагато простіше читати, ніж 2532673462
   return 0;
}
```

Якщо компілятор не підтримує C++14, то використовувати бінарні літерали і розділювач цифр ви не зможете – компілятор видасть помилку.

**Символьні константи** мають один або два символи, що подаються в апострофах. Односимвольні константи займають у пам'яті один байт і мають стандартний тип char (character-символ). Двосимвольні константи займають два байти і мають тип int. Символьні константи мають цілий тип і їх можна використовувати як цілочислові операнди у виразах.

Заслуговують уваги послідовності, що починаються зі знака «\», їх називають **керуючими** або **escape-послідовностями**. Символ зворотної косої риски «\» використовується для запису кодів, що не мають графічного зображення, для запису символів, а також для виведення символьних констант, якщо їх коди задані у вісімковому та шістнадцятковому вигляді (табл. 1.4).

Таблиця 1.4. Керуючі послідовності мови C++

| | |
|---|---|
| \a | звуковий сигнал |
| \b | повернення на крок |
| \f | переведення сторінки (формату) |
| \n | новий рядок |
| \r | повернення каретки |
| \t | горизонтальна табуляція |
| \v | вертикальна табуляція |
| \\ | символ «\» – зворотна коса риска |
| \' | символ «'» – апостроф |
| \" | символ «"» – лапки |
| \0 | нуль-символ |
| \? | знак питання |
| \0ddd | вісімковий код символу |
| \0xddd | шістнадцятковий код символу |

**Рядкова константа** (рядковий літерал) – це послідовність символів, що подається в лапках (тобто в символах «"») і зберігається у неперервній ділянці пам'яті, наприклад: "Це рядковий літерал". У кінець кожного рядкового літералу компілятором додається нуль-символ, що представляється керуючою послідовністю «\0».Тому довжина рядка завжди на одиницю більше кількості символів у його записі. Таким чином, порожній рядок (" ") має довжину 1 байт.

Слід звернути увагу на різницю між рядком з одного символу, наприклад, “C” і символьною константою ‘C’. Порожня символьна константа неприпустима.

**Керуючі послідовності** можуть також застосовуватись у рядкових константах. Так, якщо всередині рядка потрібно записати лапки, то перед ними слід розташувати зворотну косу риску («\»), за якою компілятор відрізняє їх від лапок, що обмежують рядок:

“Книга має назву \”Мова програмування C++\” “.

Рядки, що записані у програмі підряд або через символи пропуску, при компіляції конкатенуються («склеюються»). Тобто послідовність двох рядків

“Навчаючи інших, ми вчимося самі.”

“Успіх – це встигнути.”

цілком еквівалентна рядку:

“Навчаючи інших, ми вчимося самі. Успіх — це встигнути.”

Довгу рядкову константу можна розмістити також на декількох рядках. У цьому випадку ставиться зворотна коса риска і натискається клавіша Enter.

Наприклад:

“Комп`ютерна програма виконує те, \
що ви їй наказали виконати, а не те, \
що ви хотіли, щоб вона виконувала.”

Поняття та приклади оголошення **типізованої константи**, тобто константи, яку використовують як змінну, значення якої не може бути змінене після ініціювання, розглянуті вище.

Існує інша можливість задання констант – з використанням директиви препроцесора **#define**, при цьому оголошення має вигляд:

```
#define ім’я константи значення константи
```

і наприкінці такого запису символ «;» не ставлять.

Наприклад:

```
#define max 65532
#define km 1000
```

Директива #define визначає ідентифікатор (ім’я константи) і послідовність символів (значення константи), яка замінює ідентифікатор у тексті програми.

**Нульовий вказівник** (NULL-вказівник) – єдина неарифметична константа мови С++.

При застосуванні великої кількості логічно взаємозалежних констант С++ доцільно користуватися **константами перелічення**.

Тип перелічення має вигляди:

enum {список іменованих констант};

- неіменоване перелічення,

enum [ім’я] {список іменованих констант};

- іменоване перелічення.

де enum – службове слово (enumerate – перелічувати); ім’я – ім’я списку констант; список іменованих констант – розділена комами послідовність ідентифікаторів або іменованих констант вигляду:

ім’я константи = значення константи.

Наприклад:

enum {Anton, Ivan, Piter};

enum Months {January = 1, February, Marth, April, May, June, July, August, September, October, November, December};

Якщо значення константи перелічення не визначено, то воно на одиницю більше значення попередньої константи. За замовчуванням перша константа має значення 0. Тоді у першому прикладі константи одержать значення: Anton = 0, Ivan = 1, Piter = 2, а у другому – значення: January = 1, February = 2, Marth = 3 тощо. Іменовані перелічення задають унікальний цілочисловий тип і можуть використовуватися як специфікації типу для визначення змінних.

## Контрольні запитання

1. Що таке змінна в програмуванні? Поясніть її призначення та зв’язок з оперативною пам’яттю комп’ютера.

2. У чому полягає різниця між оголошенням (declaration) та ініціалізацією (initialization) змінної? Наведіть приклади коду.

3. Які існують правила іменування змінних (ідентифікаторів) у C++? Назвіть щонайменше три заборонені варіанти назв змінних.

4. Що таке ключові (зарезервовані) слова у C++? Чи можна використовувати їх як назви змінних?

5. Перелічіть базові цілочисельні типи даних у C++. У чому полягає основна різниця між типами short, int та long?

6. Як працюють модифікатори signed та unsigned? Що станеться, якщо змінній типу unsigned int присвоїти від’ємне значення?

7. Чим відрізняються типи даних з плаваючою крапкою float та double? У яких випадках краще використовувати кожен із них?

8. Для чого використовується тип char? Як цей тип даних пов’язаний з таблицею кодування (наприклад, ASCII)?

9. Які значення може приймати логічний тип даних bool? Як ці значення інтерпретуються в числовому еквіваленті (0 та 1)?

10. За допомогою якого оператора/функції можна дізнатися розмір пам’яті (у байтах), який займає певний тип даних?

11. Як правильно оголосити константу в C++? Чому в хорошому коді рекомендується використовувати іменовані константи замість так званих «магічних чисел»?

## Лекція №2. Оператори мови С++

Для здійснення маніпуляцій з даними мова С++ застосовує широкий набір операцій (див. табл. 2.1), що виконують формування і, відповідно, подальше обчислення виразів. Вирази містять одну або декілька операцій, об'єкти яких називають операндами. Операції являють собою деяку дію, що виконується над одним (унарна) або декількома (бінарна, тернарна) операндами , і мають позначення (наприклад, операція перевірки на рівність – позначення «==», операція обчислення залишку від ділення цілих чисел – позначення «%» тощо).

Операції поділяються на [7]:

- унарні або одномісні – &, *, -, +, ~, !, ++, --, sizeof;
- бінарні або двомісні – +, -, *, /, %, <>, &, :, ^, , <=, ==, >=, !=, &&, ||, =,*=, /=, %=, +=, -=, <<=, >>=, &=, |=, ^=, ., ->, ,, ( ), [ ];
- умовну тернарну або тримісну операцію – ?: .

**Унарні оператори** – це ті, які застосовуються тільки до одного операнду. Існують два унарних арифметичних оператори: *плюс* (+) і *мінус* (–).

| Оператор | Символ | Приклад | Операція |
|---|---|---|---|
| Унарний плюс | + | +x | Значення x |
| Унарний мінус | − | −x | Від'ємне значення x |

Унарний оператор **плюс** повертає значення операнда. Так, +5 = 5; +x = x. Унарний плюс здебільшого додали в якості симетрії з унарним оператором мінус. Унарний оператор **мінус** повертає операнд, помножений на (–1). Так, якщо x = 5, то −x = −5.

Обидва цих оператори потрібно розміщувати безпосередньо перед самим операндом, без пробілу (−x, а не −␣x).

Не слід плутати унарний оператор мінус з бінарним оператором віднімання, хоч вони і використовують один і той же символ. Наприклад, у виразі x = 5 − −3; перший мінус – це оператор віднімання, а другий – це унарний мінус.

Порядок застосування операції визначається пріоритетом операції (яка операція виконується раніше, а яка пізніше) та асоціативністю (виконується зліва

направо або справа наліво). У першу чергу реалізуються операції з найвищим пріоритетом. У табл. 2.1 літерою «Л» позначено величину, що стоїть ліворуч від знака операції, літерою «П» – величину, яка розташована праворуч від знака операції, а символом «→» напрямок виконання операції.

Таблиця 2.1. Основні операції мови С++

| № | Операції | Порядок виконання |
|---|---|---|
| 1 | ( ) , { } –> . | Л → П |
| 2 | ! ~ ++ -- & * (type) | П → Л |
| 3 | sizeof | П → Л |
| 4 | * / % | Л → П |
| 5 | + – | Л → П |
| 6 | << >> | Л → П |
| 7 | < <= > >= | Л → П |
| 8 | == != | Л → П |
| 9 | & | Л → П |
| 10 | ^ | Л → П |
| 11 | \| | Л → П |
| 12 | && | Л → П |
| 13 | \|\| | Л → П |
| 14 | ?: | П →Л |
| 15 | = += *= –= /= %= | П →Л |
| 16 | , | Л → П |

Розглянемо основні операції.

**Арифметичні операції**:

**+** – додає величину П до Л;

– – віднімає П із Л;

***** – множення П і Л;

**/** – ділення Л на П;

**%** – залишок від ділення величини Л на величину П (для цілих чисел), наприклад, якщо int g = 12;, то операція g = g % 9; дасть результат: g = 3;

**++** – унарна операція інкремент. Якщо змінна розташовується праворуч від знака операції (префіксна форма), то значення збільшується на 1 до використання. Якщо ж змінна знаходиться ліворуч від знака операції (постфіксна форма), то її значення збільшується на 1 після використання.

Наприклад:

int d;

++d; – префіксний інкремент;

d++; – постфіксний інкремент;

-- – унарна операція декремент аналогічно інкременту має дві форми: префіксную (змінна розташована праворуч від знака операції) – зменшення значення змінної на 1 відбувається до її використання; постфіксну (змінна знаходиться ліворуч від знака операції) – зменшення значення змінної на 1 після її використання.

**Операції присвоювання**:

**=** – присвоювання значення П змінній Л;

**+=** – додає величину П до змінної Л;

**–=** – віднімає величину П від змінної Л;

***=** – множення змінної Л на величину П;

**/=** – ділення Л на П;

**%=** – видає залишок від ділення Л на П.

Просте присвоювання здійснює операція «=». Допускається одночасне зчіплювання декількох операцій присвоювання за умови, що всі операнди мають однаковий тип, наприклад:

int i, j, c;

i = j = c = 0; .

Операції «+=», «-=», «*=», «/=» виконують складні присвоювання і дозволяють записувати вирази коротше, наприклад:

s += 7; //*s=s + 7*

i *= j + 5; //*i=i*(j +5)*

g%=9; //*g=g%9.*

**Операції відношення** порівнюють значення Л зі значенням П:

**<** – менше;

**<=** – менше або дорівнює (не перевищує);

**==** – дорівнює;

**>** – більше;

>= – більше або дорівнює (не менше);

**!=** – не дорівнює.

У мові С++ «істина» (true) – це ненульова величина, «неправда» (false) – це нуль (0). У більшості випадків одиницю – 1 – використовують як ненульове значення.

Операції відношення повертають ціле значення 1, якщо умова вірна, або 0, якщо умова хибна.

**Логічні операції** оперують з цілими розмірами або з розмірами, які можна перетворити на цілі. Обчислення зупиняється, які тільки визначиться, чи є вираз правдивим («істина») або помилковим («неправда»). При цьому, як і для операцій відношення, значенням «істина» відповідає одиниця (1), а значенням «неправда» – нуль (0).

В мові С++ існує три логічні операції:

**&&** – логічне «AND» (кон'юнкція);

|| – логічне «OR» (диз'юнкція);

**!** – логічне «NOT» (заперечення).

Результат операції «&&» є «істина» (1), якщо обидва її операнди правдиві (не рівні 0). Результат операції «||» – «істина» (1), якщо хоча б один з її операндів є «істина». Логічне заперечення «!» перетворює свій операнд на «істину» (1), якщо він дорівнює 0, і на «неправду» (0), якщо він не дорівнює 0.

З використанням логічних операцій та операцій відношення записують різні умовні вирази, наприклад, умова 3 < x < 5 запишеться як (x > 3 && x < 5).

**Операції обробки окремих бітів** застосовують для обробки даних як послідовностей бітів (розрядів), кожний з яких набуває значення 0 або 1. В мові С++ маємо наступні операції:

**&** – операція бітового множення (кон'юнкція);

| – операція бітового додавання (диз'юнкція);

^ – додавання за модулем 2;

~ – інвертування;

>> – зсув праворуч;

<< – зсув ліворуч.

**Змінна-вказівник** зберігає значення, що є адресою об’єкта в пам’яті комп’ютера. Через вказівник можна звертатися до об’єкта.

Операції з адресами та вказівниками:

**&** – одержання адреси: видає адресу змінної, ім’я якої розташоване праворуч від позначення операції;

***** – непряма адресація (розіменування): видає значення, записане за адресою, на яку посилається покажчик.

**Додаткові операції**:

**sizeof()** – знаходить розмір (у байтах) операнда, розташованого праворуч від назви операції;

**(type)** – операція приведення типу перетворює наступне за нею значення в тип, визначений ключовим словом, укладеним у круглі дужки.

Наприклад:

```
i = i+(int)*3.14;
```

**?:** – тернарна (з трьома операндами) операція, що має вигляд:

```
вираз1? вираз2 : вираз3;
```

Тут, якщо результат обчислення першого операнда (вираз1) не дорівнює 0 («істина»), то результатом операції буде значення другого операнда (вираз2), інакше – третього операнда (вираз3).

Наприклад, знаходження найбільшої з двох величин a і b, можна записати:

```
max = (b > a)? b : a;
```

Мова С++ налічує широкий спектр **математичних функцій** (табл. 2.2). Для їх використання слід включити в код програми заголовний файл math.h [2].

Таблиця 2.2. Математичні функції (заголовний файл math.h)

| **Прототип функції** | **Ім'я** | **Призначення** |
|---|---|---|
| double sin (double _x); | sin (x) | синус x (в радіанах) – sin x |
| double cos (double _x); | cos (x) | косинус x (в радіанах) – cos x |
| double tan (double _x); | tan (x) | тангенс x (в радіанах) – tg x |
| double asin (double _x); | asin (x) | арксинус x – arcsin x |
| double acos (double _x); | acos (x) | арккосинус x – arcos x |
| double atan (double _x); | atan (x) | арктангенс x – arctg x |
| double atan2 (double _y, double_x); | atan2 (y,x) | арктангенс y/x – arctg (y/x) |
| double sinh (double _x); | sinh (x) | синус гіперболічний x – sh x |
| double cosh (double _x); | cosh (x) | косинус гіперболічний x – ch x |
| double tanh (double _x); | tanh (x) | тангенс гіперболічний x – th x |
| double log (double _x); | log (x) | натуральний логарифм x – ln x |
| double log10 (double _x); | log10 (x) | десятковий логарифм x – log x |
| double exp (double _x); | exp (x) | піднесення e до степеня x – $e^x$ |
| double pow (double _x, double_y); | pow (x,y) | піднесення x до степеня y – $x^y$ |
| double pow 10 (int _p); | pow10 (p) | повертає $10^p$ |
| double sqrt (double _x); | sqrt (x) | корінь із x, x > 0 |
| double hypot (double_x, double_y); | hypot (x,y) | корінь із $(x^2+y^2)$ |
| double fabs (double _x); | fabs (x) | абсолютне значення x – \|x\| типу double |
| int abs (int _x); | abs (x) | абсолютне значення x – \|x\| типу int |
| long labs (long _x); | labs (x) | абсолютне значення x – \|x\| типу long |
| double fmod (double _x, double_y); | fmod (x,y) | залишок від ділення x на y |
| double ceil (double _x); | ceil (x) | округлення до більшого |
| double floor (double _x); | floor (x) | повертає найближче ціле, не більше за x |
| double modf (double _x, double); | modf(x,&p) | виділяє цілу й дробову частинні числа |
| double atof(const char* _s); | atof (s) | перетворює рядок символів у число з плаваючою крапкою |

Для основних математичних констант в C++ передбачено так звані **визначені константи**:

| | |
|---|---|
| π | M_PI = 3.1415… |
| e | M_E = 2.71828… |
| sqrt(2) | M_SQRT2 = 1.4142… |
| ln(2) | M_LN2 = 0.6931… |
| … | |

## Представлення чисел в двійковій системі числення

Розглянемо звичайне десяткове число, наприклад, 5623. Інтуїтивно зрозуміло, що означають всі ці цифри: (5 * 1000) + (6 * 100) + (2 * 10) + (3 * 1). Оскільки в десятковій системі числення всього 10 цифр, то кожне значення множиться на множник 10 у степені n. Вищенаведений вираз можна записати наступним чином: $(5 * 10^3) + (6 * 10^2) + (2 * 10^1) + (3 * 10^0)$ [8, 9].

Аналогічно працюють двійкові числа, за винятком того, що в системі всього 2 числа (0 і 1) і множник не 10, а 2. Так само як коми (або пробіли) використовуються для покращення читабельності великих десяткових чисел (наприклад, 1, 427, 435), двійкові числа пишуться групами – в кожній по 4 цифри (наприклад, 1101 0101).

| Десяткове значення | Двійкове значення |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |
| 8 | 1000 |
| 9 | 1001 |
| 10 | 1010 |
| 11 | 1011 |
| 12 | 1100 |
| 13 | 1101 |
| 14 | 1110 |
| 15 | 1111 |

## Конвертація чисел з двійкової системи числення в десяткову

У прикладах, наведених нижче, ми працюємо з цілочисельними значеннями unsigned. Розглянемо 8-бітне (1-байтове) двійкове число: 0101 1110. Воно означає (0 * 128) + (1 * 64) + (0 * 32) + (1 * 16) + (1 * 8) + (1 * 4 ) + (1 * 2) + (0 * 1). Якщо підсумувати, то отримаємо десяткове 64 + 16 + 8 + 4 + 2 = 94.

Ось той же процес, але в таблиці. Ми множимо кожну двійкову цифру на її значення, яке визначається її місцезнаходженням. Конвертуємо двійкове 0101 1110 в десяткову систему:

| **Двійковий символ** | 0 | 1 | 0 | 1 | 1 | 1 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| *** Значення символу** | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **= Результат (94)** | 0 | 64 | 0 | 16 | 8 | 4 | 2 | 0 |

Конвертуємо двійкове 1001 0111 в десяткову систему:

| **Двійковий символ** | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 1 |
|---|---|---|---|---|---|---|---|---|
| *** Значення символу** | 128 | 64 | 32 | 16 | 8 | 4 | 2 | 1 |
| **= Результат (151)** | 128 | 0 | 0 | 16 | 0 | 4 | 2 | 1 |

В результаті отримуємо: 1001 0111 (двійкове) = 151 (десяткове)

В такий спосіб можна легко конвертувати і 16-бітні, і 32-бітні двійкові числа, просто додаючи стовпчики. Зверніть увагу, найпростіше починати розрахунок справа наліво, помножуючи на 2 кожне наступне значення.

**Конвертація чисел з десяткової системи числення в двійкову**

Один зі способів конвертації чисел з десяткової системи числення в двійкову полягає в послідовному діленні числа на 2 і записуванні остачі. Якщо остача (**r** від англ. *"**r**emainder"*) є, то пишемо 1, якщо немає, то пишемо 0. Потім, записуючи остачі знизу вгору, отримаємо готове двійкове число.

Наприклад, конвертуємо десяткове число 148 в двійкову систему числення:

148 / 2 = 74 r0

74 / 2 = 37 r0

37 / 2 = 18 r1

18 / 2 = 9 r0

9 / 2 = 4 r1

4 / 2 = 2 r0

2 / 2 = 1 r0

1 / 2 = 0 r1

Записуємо остачі знизу вгору: 1001 0100.

В результаті отримуємо: 148 (десяткове) = 1001 0100 (двійкове)

Ви можете перевірити отриману відповідь шляхом конвертації двійкового числа назад в десяткову систему:

(1 * 128) + (0 * 64) + (0 * 32) + (1 * 16) + (0 * 8) + (1 * 4) + (0 * 2) + (0 * 1) = 148

**Бітові прапори**

Використовуючи цілий байт для зберігання значення логічного типу даних (bool), ви займаєте тільки 1 біт, а решта 7 з 8 – не використовуються. Хоча в цілому це нормально, але в особливих, ресурсоємних випадках, пов’язаних з безліччю логічних значень, може бути корисно “упакувати” 8 значень типу bool в 1 байт, заощадивши при цьому пам’ять і збільшивши, таким чином, продуктивність. Ці окремі біти і називають **бітовими прапорами** [3]. Оскільки прямого доступу до цих бітів немає, то для операцій з ними використовують спеціальні **побітові оператори**.

Чому бітові прапори корисні? Замість однієї змінної myflags, розглянемо випадок, коли у вас є дві змінні: myflags1 та myflags2, кожна з яких може зберігати 8 значень. Якщо ви визначите їх як два окремих логічних набори, то вам потрібно буде 16 логічних значень і, таким чином, 16 байт. Однак з використанням бітових прапорів вам знадобиться тільки 10 байт (8 для визначення параметрів та 1 для кожної змінної). А ось, якщо у вас буде 100 змінних myflags, то, використовуючи бітові прапори, вам знадобиться 108 байт замість 800. Чим більше ідентичних змінних вам потрібно, тим більш значною буде економія пам’яті.

Також, якщо у вас є невикористовувані бітові прапори і вам потрібно додати параметри пізніше, ви можете просто визначити бітовий прапор. Немає необхідності в зміні прототипу функції, а це – плюс до забезпечення зворотної сумісності.

## Введення в std::bitset

В Стандартній бібліотеці C++ є об’єкт **std::bitset**, який спрощує роботу з бітовими прапорами [7].

Для його використання необхідно підключити заголовний файл bitset, а потім визначити змінну типу std::bitset, вказавши необхідну кількість біт. Вона повинна бути константою часу компіляції.

Наприклад:

```
#include <bitset>
std::bitset<8> bits;        // нам потрібно 8 біт
```

За бажанням std::bitset можна ініціалізувати початковим набором значень.

Наприклад:

```
#include <bitset>
std::bitset<8> bits(option1|option2);   //почнемо з увімкнених option1 і option2
std::bitset<8> morebits(0x2);           //почнемо з бітового шаблону 0000 0010
```

Зверніть увагу! Наше початкове значення *конвертується в двійкову систему*. Оскільки ми ввели шістнадцяткове 2, то std::bitset перетворює його в двійкове 0000 0010.

У std::bitset є 4 основні **функції**:

- **test()** – дозволяє дізнатися значення біта (0 чи 1).

- **set()** – дозволяє *увімкнути* біти (якщо вони вже увімкнені, то нічого не відбудеться).

- **reset()** – дозволяє *вимкнути* біти (якщо вони вже вимкнені, то нічого не відбудеться).

- **flip()** – дозволяє змінити значення біт на протилежні (з 0 на 1 або з 1 на 0).

Кожна з цих функцій приймає в якості параметрів позиції бітів. Позиція крайнього правого біта (останнього) – 0, потім порядковий номер зростає з кожним наступним бітом вліво (1, 2, 3, 4 і т.д.). Намагайтеся давати змістовні імена бітовим індексам (або шляхом присвоювання їх константним змінним, або за допомогою перерахувань).

Зверніть увагу! Відправляючи змінну bits в std::cout – виводяться значення всіх біт в std::bitset.

Пам’ятайте! Значення, яким ініціалізується std::bitset, розглядається як двійкове, в той час як функції std::bitset використовують позиції бітів!

std::bitset також підтримує стандартні побітові оператори (|, & та ^), які також можна використовувати (вони корисні при виконанні операцій відразу з декількома бітами).

Замість виконання всіх побітових операцій вручну, рекомендується використовувати std::bitset, оскільки він зручніший і менш схильний до помилок.

**Бітові маски**

Увімкнення, вимкнення, перемикання або запит відразу декількох бітів можна здійснити в одній бітовій операції. Коли ми з’єднуємо окремі біти разом, з метою їх модифікації як групи, то це називають **бітовою маскою** [6].

Розглянемо приклад. У наступній програмі ми просимо користувача ввести число. Потім, використовуючи бітову маску, ми зберігаємо тільки останні 4 біти, значення яких і виводимо в консоль:

```
#include <iostream>
int main()
{
   const unsigned int lowMask = 0xF; //бітова маска для зберігання останніх 4 біт (шістнадцятковий літерал для 0000 0000 0000 1111)
   std::cout << "Enter an integer: ";
   int num;
   std::cin >> num;
   num &= lowMask;      //видаляємо перші біти, щоб залишити останні
   std::cout << "The 4 low bits have value: " << num << '\n';
   return 0;
}
```

Результат виконання програми:

Enter an integer: 151

The 4 low bits have value: 7

Тож, 151 в десятковій системі = 1001 0111 в двійковій. lowMask – це 0000 1111 у 8-бітній двійковій системі. 1001 0111 & 0000 1111 = 0000 0111, що дорівнює десятковому 7.

**Контрольні запитання**

1. У чому полягає принципова різниця між унарними та бінарними операторами? Наведіть по два приклади кожного з них.

2. Як працює операція отримання залишку від ділення (модуль %)? Чи можна її застосовувати до дійсних чисел (з плаваючою крапкою)?

3. Поясніть різницю між префіксною (++x) та постфіксною (x++) формами інкременту. Наведіть приклад ситуації, де ця різниця вплине на кінцевий результат виконання коду.

4. У якому порядку виконуються арифметичні операції, якщо в одному виразі є додавання, множення та унарний мінус? Що використовується для зміни пріоритету операцій?

5. Який тип даних є результатом виконання операцій відношення (==, !=, <, > тощо)?

6. Яка критична помилка часто виникає у початківців при перевірці на рівність? Поясніть різницю між = та ==.

7. Чому може бути небезпечно порівнювати два числа з плаваючою крапкою (float або double) за допомогою оператора ==?

8. Як працюють комбіновані (складені) оператори присвоювання, такі як +=, -= або *=? У чому полягає перевага їх використання?

9. Для чого використовуються побітові оператори & (AND), | (OR) та ^ (XOR)? Поясніть принципову різницю між побітовим & та логічним &&.

10. Як працюють оператори побітового зсуву вліво (<<) та вправо (>>)? Яку математичну дію над цілим числом можна швидко виконати за допомогою зсуву на 1 біт вліво або вправо?

11. Які префікси потрібно використовувати в коді програми, щоб компілятор зрозумів, що число записане у двійковій, вісімковій чи шістнадцятковій системі числення?

12. Опишіть базовий алгоритм переведення цілого числа з двійкової системи числення в десяткову.

13. Чому шістнадцяткова система числення так широко використовується в програмуванні, особливо при роботі з пам'яттю та кольорами? Як вона пов'язана з байтами?

## Лекція №3. Порядок виконання коду в програмі. Цикли і розгалуження

### Порядок виконання програм

Більшість програм, які ми розглядали до цього моменту, були лінійними з послідовним виконанням, тобто порядок виконання у них один і той же кожен раз: виконуються одні й ті ж вирази, навіть якщо значення, які вводить користувач, – змінюються.

Але на практиці це не завжди може бути вірно. В мові C++ є **оператори управління порядком виконання програми**, які дозволяють програмісту змінити потік виконання програми центральним процесором [4, 5].

### Зупинка

Найпростіший оператор управління порядком виконання програми – це зупинка (вихід з програми), який повідомляє програмі негайно припинити своє виконання. У мові C++ зупинка здійснюється за допомогою функції **exit()**, яка визначена в заголовному файлі **cstdlib**. Функція exit() приймає цілочисельний параметр, який потім повертає назад в операційну систему в якості коду виходу.

Наприклад:

```
# include <iostream>
#include <cstdlib> // для функції exit()
int main()
{
   std::cout << 5;
   exit(0);    // завершуємо виконання програми і повертаємо 0 назад в операційну систему
   // Наступні умови ніколи не виконаються
   std::cout << 3;
   return 0;
}
```

### Стрибок

Наступним оператором управління порядком виконання програми є **стрибок** (або ***“перехід”***) [3]. Він повідомляє компілятору під час виконання

програми перейти від одного виразу до іншого, тобто виконати стрибок. Ключові слова **goto**, **break** та **continue** використовують для різних типів стрибків.

**Виклики функцій** – це також, певною мірою, стрибки. При виконанні виклику функції, ЦП переходить до початку функції, яку викликали. Коли викликана функція закінчує свою роботу, виконання програми повертається до наступного виразу, який слідує за викликом цієї функції.

**Умовне розгалуження** змушує програму змінити свій порядок виконання, ґрунтуючись на значенні результату виразу. Одним з основних операторів умовного розгалуження є оператор **if/else**.

Наприклад:

```
int main()
{
   // Виконуємо A
   if (expression)
      // Виконуємо B
   else
      // Виконуємо C
   // Виконуємо D
}
```

Тут є два можливих шляхи виконання програми. Якщо результатом виразу буде true, то програма виконає A, B і D. Якщо ж результатом виразу буде false, то програма виконає A, C і D. Тобто виконуватися буде або B, або C, обидва варіанти виконуватися разом не будуть. Це вже не лінійна програма.

**Цикл** змушує програму багаторазово виконувати певну кількість виразів до тих пір, поки певна умова не стане хибною.

Наприклад:

```
int main()
{
   // Виконуємо A
   // Виконуємо B в циклі 0 або більше разів
```

```
   // Виконуємо C
}
```

Ця програма може виконуватися як ABC, ABBC, ABBBC, ABBBBC або навіть AC. Знову ж таки, вона більше не є лінійною, її порядок виконання залежить від того, скільки разів виконається цикл (якщо взагалі виконається).

У мові C++ існує 4 типи **циклів**:

- цикл while

- цикл do while

- цикл for

- цикл foreach (додали в C++11).

**Винятки** пропонують механізм обробки помилок, що виникають у функції. Якщо в функції виникає помилка, з якою вона не може впоратися, то вона може згенерувати виняток. Це змусить ЦП перейти до найближчого блоку коду, який обробляє виняток даного типу.

**Умовні розгалуження if/else**.

Найпростішими умовними розгалуженнями в мові C++ є розгалуження типу **if/else,** що реалізуються наступним чином [1]:

```
if (вираз)
   оператор1
або
if (вираз)
   оператор1
else
   оператор2
```

Вираз називають **умовою** (або ***"умовним виразом"***). Якщо результатом перевірки умови є true (будь-яке ненульове значення), то буде виконуватися оператор1. Якщо ж результатом перевірки умови є false (0), то буде виконуватися оператор2.

Наприклад:

```
#include <iostream>
```

```
int main()
{
   std::cout << "Enter a number: ";
   int a;
   std::cin >> a;
   if (a > 15)
      std::cout << a << " is greater than 15\n";
   else
      std::cout << a << " is not greater than 15\n";
   return 0;
}
```

**Використання декількох операцій в розгалуженнях if/else**

Оператор **if** виконує *лише одну* операцію, якщо вираз є true, і також *лише одну* операцію виконує **else**, якщо вираз є false. Щоб виконати декілька операцій підряд, необхідно передбачити **блок** виразів, використавши для цього фігурні дужки (**{ }**) [2].

Наприклад:

```
#include <iostream>
int main()
{
   std::cout << "Enter a number: ";
   int a;
   std::cin >> a;
   if (a > 15)
      {
      // Обидві операції будуть виконані, якщо a > 15
      std::cout << "You entered " << a << "\n";
      std::cout << a << " is greater than 15\n";
      }
   else
```

```
        {
    // Обидві операції будуть виконані, якщо a <= 15
    std::cout << "You entered " << a << "\n";
    std::cout << a << " is not greater than 15\n";
    }
  return 0;
}
```

**Неявне вказування блоків**

Якщо програміст не вказав дужки для блоку в if/else, то компілятор неявно зробить це замість нього. Так, наступний фрагмент коду:

```
if (вираз)
  оператор1
else
  оператор2
```

буде виконуватися як:

```
if (вираз)
{
  оператор1
}
else
{
  оператор2
}
```

**Поєднання розгалужень if/else**

Розгалуження if/else можна поєднувати:

```
#include <iostream>
int main()
{
  std::cout << "Enter a number: ";
  int a;
```

```
    std::cin >> a;
    if (a > 15)
        std::cout << a << " is greater than 15\n";
    else if (a < 15)
        std::cout << a << " is less than 15\n";
    else
        std::cout << a << " is exactly 15\n";
    return 0;
        }
```

**Вкладені розгалуження if/else**.

Розгалуження if можуть бути вкладені один в один:

```
#include <iostream>
int main()
{
    std::cout << "Enter a number: ";
    int a;
    std::cin >> a;
    if (a > 15)         // зовнішній оператор if
        // Це поганий спосіб написання вкладених операторів if
        if (a < 25)     // внутрішній оператор if
            std::cout << a << " is between 15 and 25\n";
        // До якого if відноситься наступний else?
        else
            std::cout << a << " is greater than 25\n";
    return 0;
        }
```

У вищенаведеній програмі ми можемо спостерігати **потенційну помилку двозначності оператора else**. До якого if відноситься оператор else: до зовнішнього чи до внутрішнього? Справа в тому, що оператор else завжди

відноситься до останнього незакритого оператору if в блоці, в якому знаходиться сам else. Тобто у вищенаведеній програмі else відноситься до внутрішнього if.

Щоб уникнути таких неоднозначностей при вкладеності операторів умовного розгалуження, рекомендується використовувати блоки операторів (вказувати дужки).

Наприклад, ось вищенаведена програма, але вже без двозначності:

```
#include <iostream>
int main()
{
   std::cout << "Enter a number: ";
   int a;
   std::cin >> a;
   if (a > 15)
   {
      if (a < 25)
         std::cout << a << " is between 15 and 25 (inclusive)\n";
      else // відноситься до внутрішнього оператора if
         std::cout << a << " is greater than 25\n";
   }
   return 0;
      }
```

Тепер зрозуміло, що оператор else відноситься до внутрішнього оператора if.

Використання дужок також дозволяє явно вказати прив’язку else до зовнішнього оператора if.

Наприклад:

```
#include <iostream>
int main()
{
   std::cout << "Enter a number: ";
   int a;
```

```
    std::cin >> a;
    if (a > 15)
    {
       if (a < 25)
          std::cout << a << " is between 15 and 25 (inclusive)\n";
    }
    else         // відноситься до зовнішнього оператора if
       std::cout << a << " is less than 15\n";
    return 0;
         }
```

Використовуючи блоки операторів, ми уточнюємо, до якого if слід прикріплювати певний else. Без блоків оператор else буде прикріплюватися до найближчого незакритого оператора if.

**Використання логічних операторів в розгалуженнях if/else**

В розгалуженнях if/else можливо перевірити відразу декілька умов, використовуючи логічні оператори (||, &&, !) [5].

Наприклад:

```
#include <iostream>
int main()
{
   std::cout << "Enter an integer: ";
   int a;
   std::cin >> a;

   std::cout << "Enter another integer: ";
   int b;
   std::cin >> b;
   if (a > 0 && b > 0) // && - це логічне І. Перевіряємо, чи є дві умови істинними
      std::cout << "Both numbers are positive\n";
```

```
else if (a > 0 || b > 0) // || - це логічне АБО. Перевіряємо, чи є істинною хоч одна з умов
    std::cout << "One of the numbers is positive\n";
else
    std::cout << "Neither number is positive\n";
return 0;
    }
```

Окремо зауважимо, що розгалуження if/else часто використовують для перевірки даних на помилки. Наприклад, значення квадратного кореня із параметра, який передається в функцію для обчислення, обов'язково повинен бути додатним. Перевірку можна реалізувати наступним чином:

```
#include <iostream>
#include <cmath> // для функції sqrt()
void printSqrt(double value)
{
  if (value >= 0.0)
    std::cout << "The square root of " << value << " is " << sqrt(value) << "\n";
  else
    std::cout << "Error: " << value << " is negative\n";
    }
```

**Оператор switch**

Ще один механізм для реалізації умовного розгалуження надає оператор **switch** [6].

Спочатку пишемо ключове слово switch, за яким слідує вираз, з яким ми хочемо працювати. Зазвичай цим виразом є окрема змінна, але це може бути й щось більш складне, наприклад, вираз: nX + 2 чи nX – nY. Єдине обмеження для цього виразу – він має бути *цілого типу даних* (тобто char, short, int, long, long long чи enum). Змінні *типу з плаваючою крапкою* або *неінтегральні типи* використовуватися не можуть.

Після виразу switch оголошуємо блок. Усередині блоку використовуємо **мітки** (англ. *“labels”*) для визначення всіх значень, які хочемо перевірити на відповідність з виразом. Існує два типи міток.

**Мітка case**. Перший вид міток – це мітка ***“кейс”***, яка оголошується з використанням ключового слова **case** і має **константний вираз**. Константний вираз – це той вираз, який генерує константне значення, іншими словами це літерал (наприклад, 5), перерахування (наприклад, COLOR_RED) чи константа (наприклад, змінна x, яка була оголошена з ключовим словом const).

Константний вираз, що знаходиться після ключового слова case, перевіряється на відповідність виразу, що знаходяться біля ключового слова switch. Якщо вони збігаються, то тоді виконується код під відповідним кейсом.

Зверніть увагу! Всі вирази case повинні генерувати унікальні значення.

Наприклад, ви не зможете зробити наступне:

```
enum Colors {
        COLOR_GRAY,
        COLOR_PINK,
        COLOR_BLUE,
        COLOR_PURPLE,
        COLOR_RED
        };
switch (z) {
   case 3:
   case 3: // заборонено, значення 3 вже використовується!
   case COLOR_PURPLE: // заборонено, COLOR_PURPLE обчислюється як 3!
              };
```

Можна використовувати відразу декілька кейсів для одного виразу.

Наприклад, наступна функція isDigit() використовує декілька кейсів для перевірки відповідності параметра p числу з ASCII-таблиці:

```
bool isDigit(char p)
{
```

```
switch (p)
{
   case '0': // якщо p = 0
   case '1': // якщо p = 1
   case '2': // якщо p = 2
   case '3': // якщо p = 3
   case '4': // якщо p = 4
   case '5': // якщо p = 5
   case '6': // якщо p = 6
   case '7': // якщо p = 7
   case '8': // якщо p = 8
   case '9': // якщо p = 9
      return true; // то повертаємо true
   default: // в протилежному випадку, повертаємо false
      return false;
}
    }
```

У випадку, якщо p є числом з ASCII-таблиці, то виконуватися буде перший оператор після кейсу – return true;.

**Мітка default**. Другий тип мітки – це так званий ***"default case"*** – **мітка за замовчуванням**, яка оголошується з використанням ключового слова **default**. Код під цією міткою виконується, якщо жоден з кейсів не відповідає виразу switch. Мітка за замовчуванням НЕ є обов'язковою. В одному switch може бути тільки один default. Зазвичай його оголошують останнім в блоці switch.

У вищенаведеному прикладі, якщо p не є числом з ASCII-таблиці, то виконується мітка за замовчуванням і повертається false.

### Оператор break

Оператор**,** оголошений з використанням ключового слова **break**, повідомляє компілятору, що ми вже зробили все, що хотіли з певним switch (або циклом while, do while чи for) і більше не маємо наміру з ним працювати [7].

Коли компілятор зустрічає оператор break, то виконання коду переходить зі switch на наступний рядок після блоку switch.

Розглянемо вищенаведений приклад, але вже з коректно вставленими операторами break:

```
switch (2)
{
  case 1: // не співпало - пропускається
     std::cout << 1 << '\n';
     break;
  case 2: // співпало! Виконання починається з наступного оператора
     std::cout << 2 << '\n'; // виконання починається тут
     break; // оператор break завершує виконання switch
  case 3:
     std::cout << 3 << '\n';
     break;
  case 4:
     std::cout << 4 << '\n';
     break;
  default:
     std::cout << 5 << '\n';
     break;
}
     // Виконання продовжується тут
```

Оскільки другий кейс відповідає виразу switch, то виводиться 2, і оператор break завершує виконання блоку switch. Решта кейсів пропускаються.

## Оператор goto

Оператор **goto** є оператором управління потоком виконання програм, який змушує центральний процесор виконати перехід з однієї точки коду в іншу (тобто здійснити стрибок). Кінцева точка стрибку ідентифікується за допомогою **мітки** [8].

Наприклад:

```
#include <iostream>
#include <cmath>          // для функції sqrt()
int main()
{
   double z;
tryAgain:                 // мітка
   std::cout << "Enter a non-negative number: ";
   std::cin >> z;
   if (z < 0.0)
      goto tryAgain;      // оператор goto
   std::cout << "The sqrt of " << z << " is " << sqrt(z) << std::endl;
   return 0;
        }
```

У цій програмі користувачеві пропонується ввести додатне число. Однак, якщо користувач введе від'ємне число, то програма, використовуючи оператор goto, виконає перехід назад до мітки tryAgain. Потім користувачеві знову потрібно буде ввести число. Таким чином, ми можемо постійно просити користувача ввести число, поки він не зробить це коректно.

Мітки використовують **область видимості функції**. Тож, важливо: оператор goto і відповідна мітка повинні знаходитися в одній і тій самій функції.

Існують деякі **обмеження на використання операторів goto**. Наприклад, ви не зможете перестрибнути вперед через змінну, яка ініціалізована в тому ж блоці, що і goto.

Наприклад:

```
int main()
{
   goto skip;         // стрибок вперед заборонений
   int z = 7;
skip: // лейбл
```

```
    z += 4;             // яке значення буде в цій змінній?
    return 0;
}
```

Загалом програмісти намагаються уникати використання оператора goto в мові C++ і в більшості інших мов програмування. Основна проблема з ним полягає в тому, що він дозволяє програмісту керувати виконанням коду так, що точка виконання може довільно переміщуватися в коді. А це, в свою чергу, створює ситуацію, яку досвідчені програмісти називають «спагеті-кодом». **Спагеті-код** – це код, порядок виконання якого нагадує тарілку зі спагеті (все заплутано і закручено), що вкрай ускладнює слідування та розуміння логіки виконання такого коду.

Оператор goto ще доволі часто використовують в деяких старих мовах програмування, таких як Basic чи Fortran, або навіть в Сі. Однак в програмування мовою C++ цей прийом заміняють використанням більш ефективних можливостей C++, таких як цикли, обробники виняткових ситуацій чи деструктори.

### Цикл while

Цикл **while** називають ***циклом з передумовою*** [9]. Він є найпростішим з 4 циклів мови C++ і за принципом дії схожий на розгалуження if/else. Синтаксис структури while наступний:

```
while (умова)
        }
  …   // тіло циклу;
        }
```

Цикл while оголошується з використанням ключового слова **while**. В якості виразу *умови* допускається використання будь-якого виразу мови C++. Якщо умова істинна, то виконується тіло циклу while. Якщо умова хибна з самого початку або стає хибною у процесі виконання тіла циклу, то виконання циклу припиняється і виконується наступний за ним оператор.

В якості *тіла циклу* допускається використання будь-якого оператору, в тому числі порожнього або складеного. Якщо тіло циклу містить більше одного оператора, то група операторів повинна бути укладена у фігурні дужки ({ }).

На відміну від розгалуженя if/else, після завершення виконання тіла циклу, управління повертається назад до while і процес перевірки умови повторюється. Якщо умова знову є true, то тоді тіло циклу виконується ще раз.

Зверніть увагу! Потрібно ініціалізувати змінну циклу while до початку виконання тіла циклу. Крім того, тіло циклу повинно містити оператор, який змінює значення змінної циклу, інакше цикл буде нескінченним.

Наприклад, наступна програма виводить всі числа від 0 до 9:

```
#include <iostream>
int main()
{
   int count = 0;
   while (count < 10)
      {
      std::cout << count << " ";
      ++count;
      }
   std::cout << "done!";
   return 0;
        }
```

Результат виконання програми:

```
0 1 2 3 4 5 6 7 8 9 done!
```

Розглянемо детально цю програму. По-перше, виконується ініціалізація змінної: int count = 0;. Умова 0 < 10 має значення true, тому виконується тіло циклу. У першому операторі ми виводимо 0, а в другому – виконуємо інкремент змінної count. Потім управління повертається до початку циклу while для повторної перевірки умови. Умова 1 < 10 має значення true, тому тіло циклу виконується ще раз. Тіло циклу буде повторно виконуватися до тих пір, поки

змінна count не дорівнюватиме 10. Тільки тоді, коли результат умови 10 < 10 буде false, цикл завершиться.

Цикл while може і взагалі не виконуватися.

Наприклад:

```
#include <iostream>
int main()
{
   int count = 15;
   while (count < 10)
      {
      std::cout << count << " ";
      ++count;
      }
   std::cout << "done!";
   return 0;
       }
```

Умова 15 < 10 відразу приймає значення false, і тіло циклу пропускається. Єдине, що виведе ця програма:

```
done!
```

**Нескінченні цикли**

Очевидно, що, якщо умова циклу завжди приймає значення true, то і сам цикл буде виконуватися нескінченно. Цю ситуацію називають **нескінченним циклом** [1 – 3].

Наприклад:

```
#include <iostream>
int main()
{
   int count = 0;
   while (count < 10)              // ця умова ніколи не буде false
      std::cout << count << " "; // тому цей рядок виконуватиметься постійно
```

```
    return 0;                    // а цей рядок ніколи не виконається
}
```

Оскільки змінна count не збільшується на одиницю в програмі, то умова count < 10 завжди буде true. Відповідно, цикл ніколи не завершиться і програма буде постійно виводити 0 0 0 0 0 ... .

Єдиний спосіб вийти з нескінченного циклу – використати один з наступних операторів: return, break, exit, goto або згенерувати виняток.

Програми, які працюють до тих пір, поки користувач не вирішить зупинити їх, іноді навмисно використовують нескінченні цикли разом з операторами return, break або exit для завершення циклу. Поширена така практика в серверних веб-додатках, які працюють безперервно і постійно обслуговують веб-запити.

**Лічильник циклу while**

Часто буває необхідно, щоб цикл while виконувався певну кількість разів. Для цього зазвичай використовують змінну у вигляді *лічильника циклу*. **Лічильник циклу** – це цілочисельна змінна, яка оголошується з єдиною метою: рахувати, скільки разів виконався цикл [4]. У вищенаведених прикладах змінна count є лічильником циклу.

Лічильникам циклу прийнято давати прості імена, такі як i, j чи k. А ще краще використовувати «реальні» імена для змінних, наприклад, count або будь-яке інше ім’я, яке надає контекст використання цієї змінної.

Також для лічильників циклу краще використовувати тип **signed int**. Використання unsigned int може призвести до несподіваних результатів.

Наприклад:

```
#include <iostream>
int main()
{
    unsigned int count = 10;
    // Рахуємо від 10 до 0
    while (count >= 0)
```

```
    {
        if (count == 0)
            std::cout << "blastoff!";
        else
            std::cout << count << " ";
        --count;
    }
    return 0;
}
```

Виявляється, вищенаведена програма є нескінченним циклом. Вона починається з виводу 10 9 8 7 6 5 4 3 2 1 blastoff!, як і передбачалося, але потім все йде шкереберть і починається відлік з 4294967295. Чому? Тому що умова циклу count >= 0 ніколи не буде хибною! Коли count = 0, то і умова 0 >= 0 має значення true, виводиться blastoff, а потім виконується декремент змінної count, відбувається переповнення і значенням змінної стає 4294967295. А оскільки умова 4294967295 >= 0 є істинною, то програма продовжує своє виконання. Лічильник циклу є типу unsigned, він ніколи не зможе бути від'ємним і цикл ніколи не завершиться.

## Ітерації

Кожне виконання циклу називають **ітерацією** (або ***"повтором"***) [5]. Оскільки тіло циклу зазвичай є блоком, і оскільки цей блок виконується з кожним повтором по-новому, то будь-які змінні, оголошені всередині тіла циклу, створюються, а потім і знищуються по-новому.

Так, у наступному прикладі змінна z створюється і знищується 6 разів:

```
#include <iostream>
int main()
{
    int count = 1;
    int result = 0; // змінна result визначена тут, оскільки вона нам знадобиться пізніше (поза тілом циклу)
```

```
    while (count <= 6) // буде 6 ітерацій
    {
        int z; // змінна z створюється тут по-новому з кожною ітерацією
        std::cout << "Enter integer #" << count << ':';
        std::cin >> z;
        result += z;
        // Збільшуємо значення лічильника циклу на одиницю
        ++count;
    } // змінна z знищується тут по-новому з кожною ітерацією!
    std::cout << "The sum of all numbers entered is: " << result;
    return 0;
}
```

Зверніть увагу, що змінна count оголошена поза тілом циклу. Це важливо і необхідно, оскільки нам потрібно, щоб значення змінної зберігалося протягом усіх ітерацій (а не знищувалося по-новому з кожною ітерацією).

**Вкладені цикли while**

Цикли while можуть бути вкладені один в один [7]. У наступному прикладі внутрішній і зовнішній цикли мають свої власні лічильники. Однак, зверніть увагу, що умова внутрішнього циклу використовує лічильник зовнішнього циклу!

```
#include <iostream>
int main()
{
    int outer = 1;
    while (outer <= 5)
    {
        int inner = 1;
        while (inner <= outer)
            std::cout << inner++ << " ";
        // Вставляємо символ нового рядка в кінці кожного рядка
```

```
        std::cout << "\n";
        ++outer;
    }
    return 0;
}
```

Результат виконання програми:

```
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

**Цикл do while**

Особливість циклу while полягає в тому, що якщо умова циклу з самого початку дорівнює false, то тіло циклу не буде виконуватися взагалі [9]. Але іноді бувають випадки, коли потрібно, щоб цикл виконався хоча б один раз, наприклад, для відображення меню. Для вирішення цієї проблеми мова C++ надає цикл do while. Цикл **do while** називають *циклом з постумовою.*

Синтаксис циклу do while наступний

```
do      {
  …     // тіло циклу;
        }
while (умова);
```

Тіло циклу do while завжди виконується хоча б один раз. Після виконання тіла циклу перевіряється умова. Якщо вона істинна, то виконання переходить до початку блока do і тіло циклу виконується знову.

Нижче наведено приклад використання циклу do while для відображення меню:

```
#include <iostream>
int main()
{
```

```
    // Змінна choice повинна бути оголошена поза тілом циклу do while
    int choice;
    do
    {
        std::cout << "Please make a selection: \n";
        std::cout << "1) Addition\n";
        std::cout << "2) Subtraction\n";
        std::cout << "3) Multiplication\n";
        std::cout << "4) Division\n";
        std::cin >> choice;
    }
    while (choice != 1 && choice != 2 &&
        choice != 3 && choice != 4);
    // Щось робимо зі змінною choice (наприклад, використовуємо оператор switch)
    std::cout << "You selected option #" << choice << "\n";
    return 0;
}
```

Цікаво, що змінна choice повинна бути оголошена поза блоками do while. Якби змінна choice була оголошена всередині блоку do, то вона була б знищена при завершенні цього блоку ще до перевірки умови while. Але нам потрібна змінна, яка буде використовуватися в умові while, отже, змінна choice повинна бути оголошена поза блоком do.

Загалом, використовувати do while замість while, коли потрібно, щоб цикл виконався хоча б один раз, є хорошою практикою.

## Цикл for

**Цикл for** в мові C++ ідеальний, коли наперед відома необхідна кількість ітерацій. Синтаксис циклу for наступний [4]:

```
for (оголошення змінних; умова; інкремент/декремент лічильника)
        {
```

```
  …   // тіло циклу;
      }
```

Цикл for можна конвертувати в еквівалентний цикл while:

```
{ // зверніть увагу, що цикл знаходиться в блоці
   оголошення змінних;
   while (умова)
   {
     тіло циклу;
     інкремент/декремент лічильника;
   }
} // змінні, оголошені всередині циклу, виходять з області видимості тут
```

Змінні, визначені всередині циклу for, мають спеціальний тип області видимості: **область видимості циклу**. Такі змінні існують тільки всередині циклу і недоступні за його межами.

Цикл for в мові C++ виконується в 3 кроки:

**1. Оголошення змінних**. Як правило, тут виконується визначення та ініціалізація лічильників циклу, а точніше – одного лічильника циклу. Ця частина виконується тільки один раз, коли цикл виконується вперше.

**2**. **Умова**. Якщо вона дорівнює false, то цикл негайно завершує своє виконання. Якщо ж умова дорівнює true, то виконується тіло циклу.

***3***. **Інкремент/декремент лічильника циклу**. Змінна збільшується або зменшується на одиницю. Після цього цикл повертається до кроку 2.

Розглянемо приклад циклу for і розберемося детально, як він працює:

```
#include <iostream>
int main()
{
  for (int count = 0; count < 10; ++count)
      std::cout << count << " ";
  return 0;
}
```

Спочатку ми оголошуємо змінну count і присвоюємо їй значення 0. Далі виконується умова count < 10, а оскільки count дорівнює 0, то умова 0 < 10 має значення true. Отже, виконається тіло циклу, в якому ми виводимо в консоль змінну count (тобто 0).

Потім виконається вираз ++count, тобто інкремент змінної. Потім цикл знову повертається до перевірки умови. Умова 1 < 10 має значення true, тому тіло циклу виконається знову. Виводиться 1, а змінна count збільшується вже до значення 2. Умова 2 < 10 є істинною, тому виводиться 2, а count збільшується до 3 і так далі. В кінці, count збільшується до 10, а умова 10 < 10 є хибною, і цикл завершується.

Отже, результат виконання програми:

0 1 2 3 4 5 6 7 8 9

### Помилка неврахованої одиниці

Однією з найчастіших проблем, з якою стикаються початківці в циклах for (а також і в інших типах циклів) є **помилка неврахованої одиниці** [3]. Вона виникає, коли цикл повторюється на 1 раз більше або на 1 раз менше від потрібної кількості ітерацій. Це зазвичай відбувається через те, що в умові використовується некоректний оператор порівняння (наприклад, > замість >= або навпаки). Як правило, ці помилки важко відстежити, тому що компілятор не скаржитиметься на них і програма працюватиме нормально, але її результати будуть неправильні.

При написанні циклів for пам’ятайте, що цикл виконуватиметься до тих пір, поки умова є істинною. Рекомендується тестувати цикли, використовуючи різні значення для перевірки працездатності циклу. Хорошою практикою є перевірка циклів за допомогою вхідних даних (чисел, символів та іншого), які змушують цикл виконатися 0, 1 і 2 рази. Якщо цикл працює справно, значить все ОК.

### Пропущені вирази в циклі

Ще однією проблемою в циклах можуть бути пропущені один або відразу всі вирази.

Наприклад:

```
#include <iostream>
int main()
{
	int count = 0;
	for (; count < 10; )
	{
		std::cout << count << " ";
		++count;
	}
	return 0;
	}
```

Результат виконання програми:

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

В наведеній програмі ініціалізацію лічильника ми прописали поза тілом циклу, а інкремент лічильника – всередині тіла циклу. У самому операторі for ми вказали лише умову. Іноді бувають випадки, коли не потрібно оголошувати лічильник циклу (тому що у нас вже є один) або збільшувати його (так як ми збільшуємо його якимось іншим способом).

**Оголошення декількох змінних-лічильників**

Хоча в циклах for зазвичай використовують тільки один лічильник, іноді можуть виникати ситуації, коли потрібно працювати відразу з декількома змінними-лічильниками. В цьому випадку використовують **оператор Кома**.

Наприклад:

```
#include <iostream>
int main()
{
	int aaa, bbb;
	for (aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)
		std::cout << aaa << " " << bbb << std::endl;
	return 0;
```

```
}
```

Цей цикл присвоює значення двом раніше оголошеним змінним: aaa = 0 та bbb = 9. Тільки з кожною ітерацією змінна aaa збільшується на одиницю, а змінна bbb – зменшується на одиницю.

Результат виконання програми:

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

**Вкладені цикли for**

Подібно до інших типів циклів, цикли for можуть бути вкладені один в один [7], що ви і можете побачити в наступному прикладі:

```
#include <iostream>
int main()
{
	for (char c = 'a'; c <= 'e'; ++c) // зовнішній цикл по буквам
	{
		std::cout << c; // спочатку виводимо букву

		for (int i = 0; i < 3; ++i) // внутрішній цикл по числам
			std::cout << i;
		std::cout << '\n';
	}
	return 0;
```

```
}
```

З однією ітерацією зовнішнього циклу виконується три ітерації внутрішнього циклу. Відповідно, результат виконання програми:

```
a012
b012
c012
d012
e012
```

## Оператори break і continue

**Оператор break і цикли**. В контексті циклів оператор **break** використовують для передчасного завершення роботи циклу [6].

Наприклад:

```
#include <iostream>
int main()
{
   int sum = 0;
   // Дозволяємо користувачеві ввести до 10 чисел
   for (int count=0; count < 10; ++count)
   {
      std::cout << "Enter a number to add, or 0 to exit: ";
      int val;
      std::cin >> val;
      // Виходимо з циклу, якщо користувач ввів 0
      if (val == 0)
         break;
      // В протилежному випадку, додаємо число до загальної суми
      sum += val;
   }
   std::cout << "The sum of all the numbers you entered is " << sum << "\n";
   return 0;
```

```
}
```

Ця програма дозволяє користувачеві ввести до 10 чисел і в кінці підраховує їх суму. Якщо користувач ввів 0, то виконається break і цикл завершиться (не важливо, скільки чисел в цей момент встиг ввести користувач).

Зверніть увагу! Оператор break може використовуватися і для виходу з нескінченного циклу.

Наприклад:

```
#include <iostream>
int main()
{
   while (true) // нескінченний цикл
   {
      std::cout << "Enter 0 to exit or anything else to continue: ";
      int val;
      std::cin >> val;
      // Виходимо з циклу, якщо користувач ввів 0
      if (val == 0)
         break;
   }
   std::cout << "We're out!\n";
   return 0;
}
```

**Оператори break і return**. Початківці часто плутають або не розуміють різниці між операторами break та return. Оператор **break** завершує роботу switch або циклу, а виконання коду продовжується з оператора, який знаходиться відразу ж після цього switch або циклу. Оператор **return** завершує виконання всієї функції, в якій знаходиться цикл, а виконання продовжується в точці після виклику функції.

Наприклад:

```
#include <iostream>
```

```
int breakOrReturn()
{
   while (true) // нескінченний цикл
   {
      std::cout << "Enter 'b' to break or 'r' to return: ";
      char sm;
      std::cin >> sm;
      if (sm == 'b')
         break; // виконання коду продовжиться з першого оператора після циклу
      if (sm == 'r')
         return 1; // виконання оператора return призведе до того, що точка виконання відразу повернеться у викликаючий об'єкт (в даному випадку, в функцію main())
   }
   // Використання оператора break призведе до того, що виконання циклу продовжиться тут
   std::cout << "We broke out of the loop\n";
   return 0;
}
int main()
{
   int returnValue = breakOrReturn();
   std::cout << "Function breakOrContinue returned " << returnValue << '\n';
   return 0;
}
```

**Оператор continue** дозволяє відразу перейти в кінець тіла циклу, пропускаючи весь код, який знаходиться під ним. Це корисно в тих випадках, коли ми передчасно хочемо завершити поточну ітерацію.

Наприклад:

```
#include <iostream>
```

```
int main()
{
	for (int count = 0; count < 20; ++count)
	{
		// Якщо число ділиться націло на 4, то пропускаємо весь код в даній ітерації після continue
		if ((count % 4) == 0)
			continue; // пропускаємо все і переходимо в кінець тіла циклу
		// Якщо число не ділиться націло на 4, то виконання коду продовжується
		std::cout << count << std::endl;
		// Точка виконання після оператора continue переміщується сюди
	}
	return 0;
}
```

Ця програма виведе всі числа від 0 до 19, які не діляться націло на 4.

У випадку з циклом for частина інкременту/декременту лічильника як і раніше виконується навіть після виконання continue (так як інкремент/декремент відбувається поза тілом циклу).

**Оператори break і continue**. Багато підручників по програмуванню попереджають не використовувати оператори break та continue, оскільки вони призводять до безпідставного переміщення точки виконання програми по всьому коду, що ускладнює розуміння і слідування логіки виконання такого коду. Проте розумне використання операторів break і continue може поліпшити читабельність циклів в програмі, зменшивши при цьому кількість вкладених блоків і необхідність наявності складної логіки виконання циклів.

Наприклад, розглянемо наступну програму:

```
#include <iostream>
int main()
{
```

```
    int count(0); // рахуємо кількість ітерацій циклу
    bool exitLoop(false); // контролюємо завершення виконання циклу
    while (!exitLoop)
    {
        std::cout << "Enter 'e' to exit this loop or any other key to continue: ";
        char sm;
        std::cin >> sm;
        if (sm == 'e')
            exitLoop = true;
        else
        {
            ++count;
            std::cout << "We've iterated " << count << " times\n";
        }
    }
    return 0;
}
```

Ця програма використовує змінну логічного типу даних для виходу з циклу, а також вкладений блок, який працюватиме тільки в тому випадку, якщо користувач не використовує символ виходу.

А ось скорочена версія, але з використанням оператора break:

```
#include <iostream>
int main()
{
    int count(0); // рахуємо кількість ітерацій циклу
    while (true) // виконання циклу продовжується, якщо його не завершить користувач
    {
        std::cout << "Enter 'e' to exit this loop or any other key to continue: ";
        char sm;
```

```
        std::cin >> sm;
        if (sm == 'e')
            break;
        ++count;
        std::cout << "We've iterated " << count << " times\n";
    }
    return 0;
}
```

Тут (з одним оператором break) ми уникнули використання як логічної змінної (а також розуміння того, навіщо вона і де використовується), так і оператора else з вкладеним блоком.

Зменшення кількості використовуваних змінних і вкладених блоків покращують читабельність і розуміння коду набагато краще, ніж оператори break або continue можуть завдати шкоди. З цієї причини вважається прийнятним їх розумне використання.

## Контрольні запитання

1. Що таке умовне розгалуження? Як воно впливає на лінійний (послідовний) хід виконання програми?

2. У чому полягає різниця між використанням кількох незалежних блоків if підряд та конструкції if... else if... else? У якому випадку програма працюватиме ефективніше?

3. Які обмеження існують щодо типів даних, які можна передавати в оператор switch? (Наприклад, чи можна використати числа з плаваючою крапкою або рядки тексту в C++?).

4. Що таке ефект «провалювання» (fall-through) в операторі switch? Яку критичну роль відіграє ключове слово break у блоках case?

5. Яке призначення блоку default в операторі switch? Чи обов'язково його писати, і що станеться, якщо його не буде, а жоден case не співпаде?

6. Як працює оператор goto і що таке «мітка» (label)? Чому в сучасному програмуванні настійно рекомендують уникати використання goto (що таке «спагеті-код»)?

7. У чому полягає принципова відмінність між циклами з передумовою (while) та з післяумовою (do while)? Який із цих циклів гарантовано виконає своє тіло щонайменше один раз?

8. Назвіть три основні блоки, з яких складається заголовок класичного циклу for. Які з цих блоків є обов'язковими, а які можна пропустити?

9. Що станеться, якщо написати цикл у вигляді for(;;) або while(true)? Як уникнути зависання програми в таких випадках?

10. Який цикл (for, while чи do while) ви б обрали для задачі: «Просити користувача вводити пароль, поки він не введе його правильно»? Обґрунтуйте свій вибір.

11. Поясніть на прикладі різницю між операторами break та continue всередині циклу. Як кожен із них впливає на поточну ітерацію та на цикл загалом?

12. Як працюють break та continue у вкладених циклах (коли один цикл знаходиться всередині іншого)? Чи зупинить break, викликаний у внутрішньому циклі, роботу зовнішнього циклу?

## Лекція №4. Масиви

На практиці часто виникає необхідність в обробці даних у вигляді довільного набору значень, тобто *масивів*. **Масив** являє собою кінцеву іменовану послідовність величин одного типу, які розрізняються за порядковим номером [3]. Опис масивів у програмі відрізняється від опису простої змінної наявністю після імені квадратних дужок **«[ ]»**, в яких задається кількість елементів масиву (розмірність). У мові С++ нумерація елементів масиву починається з 0.

Розглянемо одновимірні масиви, оголошення яких допускає одну з таких форм запису:

**<тип> <ім'я> [n];**
**<тип> <ім'я> [n] = {значення};**
**<тип> <ім'я> [ ] = {значення};**

При оголошенні одновимірного масиву, коли масив відразу ініціюється, можна не вказувати його розмір. Якщо ж ініціювання не здійснюється під час оголошення масиву, то кількість індексів слід задати обов'язково константним виразом.

Наприклад:

```
float m [6];
float m [6] = {3.4, 4.5, 5.6, 6.7, 8.9, 10.3};
float m [ ] = {3.45, 4.56, 5.67, 6.78);
```

Зрозуміло, що надалі кількість елементів змінити неможливо. Для того щоб обнулити елементи оголошеного масиву, достатньо ініціювати його перший елемент: int mas[5]={0};.

За замовчуванням, якщо в оголошеному масиві ініціюється тільки декілька перших елементів, то його інші елементи ініціюються нулями. Так, у випадку, коли float mas[10]= {2.2,34.56};, останні вісім елементів масиву одержать значення 0.

***Приклад 4.1.*** Обчислити функцію $y = ax_i^2 - \sin x_i$ , аргументи якої $x_j$ – елементи одновимірного масиву, що мають наступні значення: $x_0 = -0{,}81$; $x_1 =$ -

0,58; $x_2$ = -0,11; $x_3$ = 0,2; $x_4$ = 0,91; $x_5$ = 1,83.

Алгоритм передбачає введення значень одновимірного масиву $x_i$ (i = 0…n-l), де n = 6, та подальше застосування їх для обчислення функції.

У програмі спочатку описується масив дійсних значень: float x[n]. Цей цикл містить операцію потокового введення cin >> x[i];, перед якою знаходиться підказка cout << “x[ ” << i << “] =”; для вказівки номера елемента x[i]. Особливість виконання операції введення cin >> x[i]; полягає в такому: зустрівши її у програмі, комп’ютер призупинить виконання програми, поки не буде введене значення елемента x[i] і натиснута клавіша Enter, після чого робота програми буде продовжена. Зазначена операція введення повторюється n разів для забезпечення введення всіх елементів масиву.

Оскільки у С++ індексація елементів масиву починається з нуля, то масив float x[6] (n = 6) із шести елементів включає індексовані елементи x[0], x[1], x[2], …, x[5]. Програма використовує два цикли: один – для введення масиву, інший – для обчислення функції. Операції введення елементів масиву та обчислення значень функції можна здійснити в одному циклі.

```
#include <iostream>
#include <cmath>
using namespace std;
void main ( )
{ const int n = 7;
  float x[n], y, a(10.5);
  int i;
  for (i = 0;i< n;i++)
  {
    cin >> x[i];        //введення елемента масиву
    y = a * x[i] * x[i] - sin(x[i]);
// виведення результату
   cout << " x["<<i<<"] ="<<x[i]<<" y = " << y << endl;
 }
```

}

Результат виконання програми:

-0.81 -0.58 -0.11  0.2  0.91  1.83

x[0] = -0.81  y = 7.61334

x[1] = -0.58  y = 4.08022

x[2] = -0.11  y = 0.236828

x[3] = 0.2  y = 0.221331

x[4] = 0.91  y = 7.90555

x[5] = 1.83  y = 34.1969

У цьому випадку передбачено введення елементів масиву в рядок, і тому клавішу Enter слід натиснути лише в кінці процесу введення.

***Приклад 4.2.*** Сформувати масив $c_k$, який містить однакові елементи двох масивів $a_i$ (i = 0…n-1), n = 7 та $b_j$ (j = 0…m-1), m = 10. Масиви a і b не мають елементів, що повторюються.

Цей приклад розв'язується з використанням вкладених циклів. Процес порівняння елементів відбувається немов у "три руки". Одна рука за параметром i вибирає елемент з масиву $a_i$, друга за параметром j вибирає елемент з масиву $b_j$, а третя за параметром k розташовує вибраний елемент у масив $c_k$.

Спочатку i = 0 (відбувається порівняння з $a_0$), а j змінюється від 0 до m-1.

У циклі за параметром **j** кожний елемент $\mathbf{b_j}$ порівнюється з $\mathbf{a_i}$ доти, поки не знайдеться $a_i = b_j$ та не буде переглянутий весь масив $b_j$. Якщо $a_i = b_j$, то $b_j$ заноситься до поточного елемента масиву $c_к$.

Далі повторюються аналогічні порівняння для i (i = 1, 2,…, n-l), тобто здійснюється порівняння елементів масиву bj з наступним елементом масиву $a_i$.

***Приклад 4.3.*** За один перегляд масиву $c_i$(i = 0…n-l), де n = 15, визначити значення, а також положення максимального і мінімального його елементів та поміняти їх місцями.

```
#include <iostream>
using namespace std;
const n=15;
```

```
void main ()
{   //опис масиву c[n] та його ініціалізація
  float c[n] = {6.4, 1.5, -5.6, 3.7, 18.9, 10.3, -0.6, 44.5,-0.2, 8.9, 55.3, 6.9, 4.3, 7.7,
10.9};
  float max, min;            // максимальний і мінімальний елементи
int imax, imin;              // індекси шуканих елементів
//виведення заданого масиву c[n]
  cout << " ***** massiv c[n] ***** n= " << n << endl;
  for (int i = 0; i < n; i++)
  cout << c[i] << " ";
// визначення max і min елементів масиву та їх індексів – imax, imin
  max = min = c[0];
  imax = imin = 0;
  for (int i =1; i<n; i++)
  {
    if (c[i] >max)
 { max = c[i];
      imax = i; }
    else
    if (c[i] < min)
    { min = c[i];
      imin = i; }
  }
// перестановка max і min елементів
  c[imin] = max;
  c[imax] = min;
// виведення max, min, imax, imin
  cout << "\n\t max=  " << max << " min= " << min << endl;
  cout << "\t imax= " << imax+1 << " imin= " << imin+1 << endl;
// виведення перетвореного масиву c[n]
```

```
  cout << " **** Rezult massiv ****” << endl;
  for (int i = 0; i < n; i++)
  cout << c[i] << " " ;
}
```

Результати виконання програми:

```
***** massiv c[n] ***** n= 15
6.4 1.5 -5.6 3.7 18.9 10.3 -0.6 44.5 -0.2 8.9 55.3 6.9 4.3 7.7 2.1
max= 55.3          min= -5.6
imax= 11           imin= 3
**** Rezult massiv ****
6.5 1.5 55.3 3.7 18.9 10.3 -0.6 44.5 -0.2 8.9 -5.6 6.9 4.3 7.7 2.1
```

## Двовимірні масиви

Крім *одновимірних* масивів, тобто таких, де позиція елемента визначається за допомогою одного індексу, у практиці розв’язання задач часто застосовують *багатовимірні* масиви. У них позиція елемента визначається записом декількох індексів. Найбільш розповсюджені **двовимірні масиви** або **матриці** [5]. Матриці являють собою порядковий запис декількох одновимірних масивів. Місце розташування кожного елемента визначається за допомогою двох індексів – номера рядка і номера стовпця, тобто порядкового номера в рядку. Індекси двовимірних масивів записують в квадратних дужках, при цьому нумерація індексів починається з нуля (0).

Наприклад, двовимірний масив цілих чисел int a[3][4], що має три рядки та чотири стовпці, представлений на рис. 4.1.

| | | | |
|---|---|---|---|
| a[0][0] | a[0][1] | a[0][2] | a[0][3] |
| a[1][0] | a[1][1] | a[1][2] | a[1][3] |
| a[2][0] | a[2][1] | a[2][2] | a[2][3] |

Рис. 4.1. Вигляд двовимірного масиву (матриці) int a[3][4]

У пам’яті комп’ютера масив розташований безперервно за рядками:
a [0][0], a [0][1], a [0][2], a [0][3], a [1][0], a [1][1], a [1][2], a [1][3], … a [2][3].

Двовимірні (і багатовимірні) масиви можна оголошувати, наприклад,

таким чином:

```
int mas [2][5] ={ 1, 5, 3, 7, 4,10, 11, I3, 14, 25 };
int mas [  ][5] ={ 1, 5, 3, 7, 4, 10, 11, 13, 14, 25 };
int mas [  ][5] ={ { 1, 5, 3, 7, 4 },{10, 11, 13, 14, 25} };
```

Тобто масив задають або списком елементів у тому порядку, в якому вони розташовані у пам'яті, або подають як масив масивів, кожний з яких поміщають в свої фігурні дужки «{ }».

При оголошенні і одночасній ініціалізації багатовимірних масивів можна опускати кількість індексів тільки першого виміру. Якщо ініціалізацію не здійснювати під час оголошення масиву, то кількість індексів треба вказувати явно.

Для здійснення введення-виведення, а також для обробки елементів двовимірного масиву у програмі слід передбачати організацію двох циклів: один – для задання значень індексу рядків, другий – індексу стовпців.

***Приклад 4.4.*** Кожний елемент матриці M розмірністю 3x4 збільшити на задане число.

```
#include <iostream>
using namespace std;
void main()
{
  const int n = 3, m = 4;            // n і m – кількість рядків і стовпців матриці
  float M [n][m], z = 10;            // z – задане число
  int i, j;
//введення елементів матриці і збільшення їх значень на z
  cout << "**** Vvod matrix " << endl;
  for (i = 0; i<n; i++)
    for (j = 0; j<m; j++)
 { cout << " M [" << i << "]" << "[" << j << "]=";
     cin >> M [i][j];
     M [i][j] += z;                  // M [i][j]= M [i][j] + z;
```

```
   }
// виведення отриманої матриці в натуральному вигляді
  cout << "\n\n***** Rezult matrix: ";
 for (i = 0; i < n; i++)
  { cout << endl;
    for (j = 0; j < m; j++)
    cout << M [i][j] << " “;}
}
```

Результат виконання програми:

```
***** Vvod matrix
M [0][0]=4.5
M [0][1]=6.7
M [0][2]=4.8
M [0][3]=23.6
M [1][0]=5.7
M [1][1]=3.7
M [1][2]=2.9
M [1][3]=6.1
M [2][0]=1.2
M [2][1]=4.5
M [2][2]=4.6
M [2][3]=2.7
***** Rezult matrix:
14.5   16.7   14.8   33.6
15.7   13.7   12.9   16.1
11.2   14.5   14.6   12.7
```

У програмі при описі матриці float M[n][m] вказується діапазон зміни двох індексів, перший з яких призначений для індексування рядків – (i), другий – для індексування стовпців – (j). Введення, обробка і виведення елементів матриці здійснюються за допомогою двох циклів, один з яких є вкладеним в іншій. Це

дозволяє при кожному значенні змінної і перебирати всі значення змінної j.

Розглянута програма може бути скорочена шляхом об’єднання всіх трьох блоків циклу в один, але в такому випадку вона буде менш наочною.

***Приклад 4.5.*** Елементи головної та побічної діагоналей квадратної матриці С розмірністю 4х4 поміняти місцями. Визначити максимальний елемент перетвореної матриці, а також номери рядка та стовпця, на перетині яких він знаходиться.

```
// визначення максимального елемента матриці та його індексів
#include <iostream>
using namespace std;
void main ()
{ const int n = 4;
  int i, j, imax, jmax;
 float max, C[ ][n] = {  {3.6, 8.9, 1.9, 5.8},
                         {8.8, 4.1, 1.2, 6.3},
                         {2.5, 6.4, 0.1, 5.5},
                         {8.8, 4.1, 1.2, 6.3} };   //ініціалізація C[n][n]
// виведення початкової матриці в натуральному вигляді
 cout << " ***** massiv C[n][n] *****";
  for (i = 0; i < n; i++)
  { cout << endl;
    for (j = 0; j < n; j++)
    cout << C[i][j] << " "; }
// перестановка елементів головної і бічної діагоналей
  float rab;                  // допоміжна змінна для перестановки
  for (i = 0; i < n; i++)     // for (i = 0, j = n-1; i < n; i++, j--)
 {
  j = n-1-i;                  //  rab = C[i][i];
  rab = C[i][i];              // C[i][i]=C[i][j];
  C[i]|i] =C[i][j];           // C[i][j] = rab; }
```

```
  C[i][j] = rab;
 }
// виведення перетвореної матриці
  cout<<"\n\n ***** REZULT massiv ***** ";
  for (i = 0; i < n; i++)
  {
   cout << endl;
    for (j = 0; j < n; j++)
    cout << C[i|j] << "   ";
  }
// визначення max елементу матриці та його індексів – imax, jmax
  max = C[0][0];
  imax = jmax = 0;
  for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
      if (C[i][j] > max)
      {
        max = C[i][j];
        imax = i; jmax = j;
      }
  cout << "\n\n max= " << max << "  index stroki = " << imax+1 << " index stolbca =
"<< jmax+1;
}
```

Результат виконання програми:

```
***** massiv C[n][n] *****
3.6  8.9  1.9  5.8
8.8  4.1  1.2  6.3
2.5  6.4  0.1  5.5
8.8  4.1  1.2  6.3
***** REZULT massiv *****
```

5.8 8.9 1.9 3.6

8.8 1.2 4.1 6.3

2.5 0.1 6.4 5.5

6.3 4.1 1.2 8.8

max = 8.9 index stroki = 1 index stolbca = 2 .

## Контрольні запитання

1. Що таке масив у мові C++? Як його елементи фізично розташовуються в оперативній пам’яті комп’ютера (хаотично чи послідовно)?

2. З якого числа починається індексація елементів масиву? Якщо масив має розмір N, яким буде індекс його останнього елемента?

3. Що означає термін «фіксований масив» (статичний)? Чи можна змінити кількість його елементів під час виконання програми (в runtime)?

4. Як правильно оголосити та відразу ініціалізувати масив з 5 цілих чисел? Наведіть приклад відповідного синтаксису.

5. Що відбудеться, якщо при ініціалізації фіксованого масиву передати менше значень, ніж його розмір (наприклад, int arr[5] = {1, 2};)? Якими значеннями будуть заповнені решта елементів?

6. Що станеться, якщо спробувати звернутися до елемента масиву за індексом, який виходить за його межі (out of bounds)? Чому компілятор C++ часто не попереджає про цю помилку і чим це небезпечно?

7. Як програмно обчислити кількість елементів у фіксованому масиві, використовуючи оператор sizeof? Напишіть формулу.

8. Чи можна присвоїти один масив іншому безпосередньо через оператор присвоювання (наприклад, arr1 = arr2;)? Обґрунтуйте свою відповідь.

9. Чому цикли (зокрема for) є невід'ємним інструментом при роботі з масивами?

10. Які параметри потрібно задати в класичному циклі for, щоб безпечно пройтись по всіх елементах масиву від першого до останнього?

11. Що таке цикл for на основі діапазону (range-based for loop), доданий у

стандарті C++11? У чому його синтаксична перевага та зручність при переборі елементів масиву?

12. Що являє собою двовимірний масив у C++? Поясніть концепцію «масив масивів» через аналогію з таблицею (рядки та стовпці).

13. Як правильно оголосити двовимірний масив на 3 рядки та 4 стовпці? Який індекс відповідає за рядок, а який – за стовпець?

14. У якому порядку елементи двовимірного масиву зберігаються в лінійній пам'яті комп'ютера: по рядках (row-major) чи по стовпцях (column-major)? Чому важливо враховувати це при написанні вкладених циклів для їх обробки?

## Лекція №5. Функції

**Функція** – це іменована логічно завершена сукупність оголошень і операторів, призначених для виконання певної задачі [2].

Програма мовою С++ містить одну або декілька функцій, кожна з яких повинна бути оголошена та визначена до її першого використання. **Оголошення функції** (прототип, заголовок) задає ім'я функції, тип значення, що повертає функція (якщо воно є), а також імена та типи аргументів, які можуть передаватися як у функцію, так і з неї. **Визначення функції** – це задання способу виконання операцій.

Серед функцій програми повинна бути одна з ім'ям **main(), – головна функція**, – яка може знаходитися в будь-якому місці програми. Ця функція виконується завжди першою і закінчується останньою.

Усі функції мають однакову структуру визначення у вигляді:

```
[тип результату] ім'я функції ([список формальних аргументів])
{
// тіло функції
опис даних;
оператори;
[return] [вираз];
};
```

де **тип результату** – будь-який базовий або раніше описаний тип значення (за винятком масиву і функції), що повертається функцією (необов'язковий параметр). За відсутності цього параметра тип результату за замовчуванням буде цілий (int). Він також може бути описаний ключовим словом (void), тоді функція не повертає ніякого значення. Якщо результат повертається функцією, то в тілі функції є необхідним оператор **return** та вираз, де **вираз** формує значення, що співпадає з типом результату;

**ім'я функції** – ідентифікатор функції, за яким завжди знаходиться пара круглих дужок «**( )**», де записуються формальні аргументи. Фактично ім'я

функції – це особливий вид покажчика на функцію, його значенням є адреса початку входу у функцію;

**список формальних аргументів** – визначає кількість, тип і порядок проходження переданих у функцію вхідних аргументів, які розділяють комою («,»). У випадку, коли параметри відсутні, дужки залишаються порожніми або містять ключове слово (void). Формальні параметри функції локалізовані в ній і недоступні для будь-яких інших функцій.

Список формальних аргументів має такий вигляд:

([const] тип 1 [параметр 1], [const] тип 2 [параметр 2], . . .)

Зверніть увагу! У списку формальних аргументів для кожного параметра треба вказати його тип (не можна групувати параметри одного типу, вказавши їх тип один раз).

Тіло функції може складатися з описів змінних та операторів. Змінні, що використовують при виконанні функції, можуть бути *глобальні* та *локальні*. Змінні, що описані (визначені) за межами функції, називають **глобальними**. За допомогою глобальних параметрів можна передавати дані у функцію, не включаючи ці змінні до складу формальних параметрів. У тілі функції їх можна змінювати і потім отримані значення передавати в інші функції. Змінні, що описані у тілі функції, називають **локальними** (або **автоматичними)**. Вони існують тільки під час роботи функції, а після реалізації функції система видаляє локальні змінні та звільняє пам’ять. Тобто між викликами функції вміст локальних змінних знищується, тому ініціювання локальних змінних треба робити щоразу під час виклику функції. За необхідності збереження цих значень, їх треба описати як статичні за допомогою службового слова static.

Наприклад:

```
static int
x, y;
```

або

```
static float p = 3.25;
```

**Статична** змінна схожа на глобальну, *але діє тільки у тій функції, в якій вона оголошена.*

На початку програми можна не описувати всю функцію, а записати тільки її **прототип**. Запис прототипу може містити тільки перелік типів формальних параметрів без імен, а наприкінці прототипу завжди ставиться символ «;», тоді як у описі (визначенні) функції цей символ після заголовка не присутній.

Механізм передачі параметрів є основним засобом обміну інформацією між функцією, що викликається, та функцією, яка викликає. Параметри, котрі зазначають у заголовку опису функції, як відомо, називають **формальними**, а параметри, які записані у операторах виклику функції – **фактичними**.

Наведемо приклад фрагмента програми з використанням функцій:

```
double sqr (double);        // прототип функції sqr()
main( )                     // головна функція
{
  // виклик функції sqr()
  cout << "Квадрат числа=" << sqr (10) << endl;
}
double sqr (double p)       //  функція sqr()
{
  return p*p;
}                   // повернення за значенням
```

Результат виконання програми:

Квадрат числа = 100

Функція завжди має бути визначена або оголошена до її виклику.

Зверніть увагу! При оголошенні, визначенні та виклику тієї самої функції типи та послідовність параметрів повинні співпадати. На імена параметрів обмежень на відповідність не існує, оскільки функцію можна викликати з різними аргументами, а в прототипах імена ігноруються компілятором (вони необхідні тільки для покращення читання програми). Тип значення, що повертає функція, та типи параметрів спільно визначають тип функції.

У найпростішому випадку при виклику функції слід вказати її ім’я, за яким у круглих дужках через кому перелічити імена аргументів, що передаються. Виклик функції може здійснюватися у будь-якому місці програми, де за синтаксисом дозволений вираз того типу, що формує функція. Якщо тип значення, що повертає функція не void, вона може входити до складу виразів або, у поодинокому випадку, розташовуватись у правій частині оператора присвоювання.

За замовчуванням, аргументи в мові C++ передаються **за значенням**, тобто їх значення копіюється в параметр функції.

Наприклад:

```
#include <iostream>
void boo(int y)
{
   std::cout << "y = " << y << std::endl;
}
int main()
{
   boo(7);          // 1-й виклик
   int x = 8;
   boo(x);          // 2-й виклик
   boo(x + 2);      // 3-й виклик
   return 0;
}
```

У першому виклику функції boo() аргументом є літерал 7. При виклику boo() створюється змінна y, в яку копіюється значення 7. Потім, коли boo() завершує своє виконання, змінна y знищується.

У другому виклику функції boo() аргументом вже є змінна x = 8. Коли boo() викликається вдруге, змінна y створюється знову і значення 8 копіюється в y. Потім, коли boo() завершує своє виконання, змінна y знову знищується.

У третьому виклику функції boo() аргументом є вираз x + 2, який обчислюється в значення 10. Потім це значення передається в змінну y. При завершенні виконання функції boo() змінна y знову знищується.

Таким чином, результат виконання програми:

```
y = 7
y = 8
y = 10
```

Оскільки в функцію передається копія аргументу, то початкове значення не може бути змінено функцією. Це добре проілюстровано в наступному прикладі:

```
#include <iostream>
void boo(int y)
{
    std::cout << "y = " << y << '\n';
    y = 8;
    std::cout << "y = " << y << '\n';
} // змінна y знищується тут
int main()
{
    int x = 7;
    std::cout << "x = " << x << '\n';
    boo(x);
    std::cout << "x = " << x << '\n';
    return 0;
}
```

Результат виконання програми:

```
x = 7
y = 7
y = 8
x = 7
```

На початку функції main() змінна x дорівнює 7. При виклику boo() значення x (7) передається в параметр у функції boo(). Усередині boo() змінній y спочатку присвоюється значення 8, а потім y знищується. Значення x не змінюється, навіть якщо змінити y.

Параметри функції, передані за значенням, також можуть бути const. Тоді вже буде 100% гарантія того, що функція не змінить значення параметру.

**Плюси передачі за значенням:**

- Аргументи, передані за значенням, можуть бути змінними (наприклад, x), літералами (наприклад, 8), виразами (наприклад, x + 2), структурами, класами або перелічуваннями (тобто майже будь-чим).

- Аргументи ніколи не змінюються функцією, в яку передаються, що запобігає виникненню побічних ефектів.

**Мінуси передачі за значенням**:

- Копіювання структур і класів може призвести до значного зниження продуктивності (особливо, коли функція викликається багато разів).

**Коли використовувати передачу за значенням**:

- У випадку передачі фундаментальних типів даних і енумераторів, коли припускається, що функція не повинна змінювати аргумент.

**Коли не використовувати передачу за значенням**:

- У випадку передачі масивів, структур і класів.

У більшості випадків, передача за значенням – це найкращий спосіб передачі аргументів фундаментальних типів даних, коли функція не повинна змінювати вихідні значення. Передача за значенням є гнучкою і безпечною, а у випадку фундаментальних типів даних ще й ефективною.

У випадку передачі змінної **за посиланням** потрібно просто оголосити параметри функції як **посилання**, а не як звичайні змінні.

Наприклад:

```
void func(int &x)  // x - це змінна-посилання
{
  x = x + 1;
```

```
}
```

В разі виклику функції змінна x стане посиланням на аргумент. Оскільки посилання на змінну обробляється точно так же, як і сама змінна, то будь-які зміни, внесені в посилання, призведуть до змін вихідного значення аргументу!

У наступному прикладі це добре проілюстровано:

```
#include <iostream>
void boo(int &value)
{
   value = 7;
}
int main()
{
   int value = 6;
   std::cout << "value = " << value << '\n';
   boo(value);
   std::cout << "value = " << value << '\n';
   return 0;
}
```

Ця програма точно така ж, як і програма з попереднього прикладу, за винятком того, що параметром функції boo() тепер є посилання замість звичайної змінної.

Результат виконання програми:

```
value = 6
value = 7
```

Як ви можете бачити, функція змінила значення аргументу з 6 на 7!

### Повернення відразу декількох значень

Іноді може знадобитися, щоб функція повертала відразу декілька значень [7]. Однак оператор return дозволяє функції мати тільки одне значення, що повертається. Одним із способів повернення відразу декількох значень є використання **посилань** в якості параметрів.

Наприклад:

```
#include <iostream>
#include <math.h> // для sin() і cos()
void getSinCos(double degrees, double &sinOut, double &cosOut)
{
   // sin() і cos() приймають радіани, а не градуси, тому потрібна конвертація
   const double pi = 3.14159265358979323846; // значення Пі
   double radians = degrees * pi / 180.0;
   sinOut = sin(radians);
   cosOut = cos(radians);
}
int main()
{
   double sin(0.0);
   double cos(0.0);
   // Функція getSinCos() повертає sin і cos в змінні sin і cos
   getSinCos(30.0, sin, cos);
   std::cout << "The sin is " << sin << '\n';
   std::cout << "The cos is " << cos << '\n';
   return 0;
}
```

Ця функція приймає один параметр (передача за значенням) в якості вхідних даних і «повертає» два параметри (передача по посиланню) в якості вихідних даних. Параметри, які використовують тільки для повернення значень назад в caller, називають **параметрами виведення**. Вони дають

зрозуміти caller-у, що значення вихідних змінних, переданих у функцію, не настільки значні, так як ми очікуємо, що ці змінні будуть перезаписані.

Якби sin() і cos() були передані за значенням, а не за посиланням, то функція getSinCos() змінила б копії sin() та cos(), а не вихідні значення і ці зміни знищились би в кінці функції – змінні вийшли б з **локальної області видимості**. Але, оскільки sin() та cos() передавалися за посиланням, то будь-які зміни, внесені в sin() або cos() (через посилання), зберігаються і за межами функції getSinCos(). Таким чином, ми можемо використовувати цей механізм для повернення відразу декількох значень назад в caller.

**Передача за константним посиланням**

Одним з найголовніших недоліків передачі за значенням є те, що всі аргументи, передані за значенням, *копіюються* в параметри функції. Коли аргументами є великі структури або класи, то цей процес може зайняти багато часу. У випадку з передачею за посиланням ця проблема легко вирішується. Коли аргумент передається за посиланням, то створюється *посилання* на фактичний аргумент (що займає мінімальну кількість часу для виконання), і ніякого копіювання значень не відбувається. Це дозволяє передавати великі структури або класи з мінімальною затратою ресурсів [3].

Однак тут також можуть виникнути потенційні проблеми. Посилання дозволяють функції змінювати значення аргументів напряму, що небажано, якщо ми хочемо, щоб аргумент був доступний тільки для читання. Коли ми знаємо, що функція не повинна змінювати значення аргументу, але не хочемо використовувати передачу за значенням, кращим рішенням буде використовувати **передачу за константним посиланням**.

**Константне (const) посилання** – це посилання на змінну, значення якої змінити через це ж посилання ніяк не вийде. Отже, якщо ми використовуємо константне посилання в якості параметру, то отримуємо 100% гарантію того, що функція не змінить аргумент!

Запустивши наступний фрагмент коду, ми отримаємо помилку компіляції:

```
void boo(const int &y) // y - це константне посилання
{
  y = 8; /* помилка компіляції: константне посилання не може змінити своє ж значення!*/
}
```

Константне (**const**) посилання корисне з наступних причин:

- Ми отримуємо гарантію від компілятора, що значення, які не повинні бути змінені – не зміняться (компілятор видасть помилку, якщо ми спробуємо зробити щось подібне тому, що було у вищенаведеному прикладі).

- Програміст, коли бачить const, розуміє, що функція не змінить значення аргументу. Це може допомогти в разі **відлагодження програми**.

- Ми не можемо передати константний аргумент в неконстантне посилання-параметр. Використання константного параметру гарантує, що ми зможемо передавати як неконстантні, так і константні аргументи в функцію.

- Константні посилання можуть приймати будь-які типи аргументів, включаючи l-values, константні l-values та r-values.

Зверніть увагу! В разі передачі аргументів за посиланням завжди використовуйте константні посилання, якщо вам не потрібно, щоб функція змінювала значення аргументів.

**Плюси передачі за посиланням:**

- Посилання дозволяють функції змінювати значення аргументу, що іноді корисно. В іншому випадку, для гарантії того, що функція не змінить значення аргументу, потрібно використовувати константні посилання.

- Оскільки при передачі за посиланнямкопіювання аргументів не відбувається, то цей спосіб набагато ефективніший і швидший за передачу за значенням, особливо при роботі з великими структурами або класами.

- Посилання можуть використовуватися для повернення відразу декількох значень з функції (через параметри виводу).

**Мінуси передачі за посиланням:**

- Важко визначити, чи є параметр, переданий по неконстантному посиланню, параметром вводу, параметром виводу чи тим і іншим одночасно. Розумне використання const та суфікса Out для зовнішніх змінних вирішує цю проблему.

- За викликом функції неможливо визначити, чи буде аргумент змінений функцією чи ні. Аргумент, переданий за значенням або за посиланням, виглядає однаково. Ми можемо визначити спосіб передачі аргументу тільки переглянувши оголошення функції. Це може призвести до ситуації, коли програміст не відразу зрозуміє, що функція змінює значення аргументу.

**Коли використовувати передачу за посиланням:**

- у випадку передачі структур або класів (використовуйте const, якщо потрібно тільки для читання);

- коли потрібно, щоб функція змінювала значення аргументу.

**Коли не використовувати передачу по посиланню:**

- у випадку передачі фундаментальних типів даних (використовуйте передачу за значенням);

- у випадку передачі звичайних масивів (використовуйте передачу за адресою).

## Передача аргументів за адресою

**Передача аргументів за адресою** – це передача адреси змінної-аргументу (а не вихідної змінної) [4]. Оскільки аргумент є адресою, то параметром функції повинен бути **вказівник**. Потім функція зможе розіменувати цей вказівник для доступу або зміни вихідного значення. Ось приклад функції, яка приймає параметр, який передається за адресою:

```
#include <iostream>
void boo(int *ptr)
{
   *ptr = 7;
}
```

```
int main()
{
   int value = 4;
   std::cout << "value = " << value << '\n';
   boo(&value);
   std::cout << "value = " << value << '\n';
   return 0;
}
```

Результат виконання програми:

```
value = 4
value = 7
```

Функція boo() змінила значення аргументу (змінну value) через параметр-вказівник ptr.

Передачу за адресою зазвичай використовують **з вказівниками** на звичайні **масиви**.

Наприклад, наступна функція виведе всі значення масиву:

```
void printArray(int *array, int length)
{
  for (int index=0; index < length; ++index)
       std::cout << array[index] << ' ';
}
```

Ось приклад програми, яка викликає цю функцію:

```
int main()
{
   int array[7] = { 9, 8, 6, 4, 3, 2, 1 }; /* пам'ятаєте, що масиви конвертуються у вказівники при передачі?*/
   printArray(array, 7); /* тому що тут array – це вказівник на перший елемент масиву (у використанні оператора & тут немає необхідності)*/
}
```

Результат виконання програми:

9 8 6 4 3 2 1

Фіксовані **масиви** конвертуються **у вказівники** в разі передачі у функцію, тому їх довжину потрібно передавати в якості окремого параметру. Перед розіменуванням параметрів, переданих за адресою, не зайвим буде перевірити – чи не є вони **нульовими вказівниками**. Розіменування нульового вказівника призведе до збою в програмі.

В наступному прикладі функція printArray() з перевіркою (виявленням) нульових вказівників:

```
#include <iostream>
void printArray(int *array, int length)
{
   // Якщо користувач передав нульовий вказівник в якості array
   if (!array)
      return;
   for (int index=0; index < length; ++index)
      std::cout << array[index] << ' ';
}
int main()
{
   int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
   printArray(array, 7);
}
```

**Передача за константною адресою**

Оскільки printArray() все одно не змінює значення отриманих аргументів, то параметр array можна зробити константою [8]:

```
#include <iostream>
void printArray(const int *array, int length)
{
   // Якщо користувач передав нульовий вказівник в якості array
```

```
    if (!array)
        return;
    for (int index=0; index < length; ++index)
        std::cout << array[index] << ' ';
}
int main()
{
    int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
    printArray(array, 7);
        }
```

Так ми бачимо відразу, що функція printArray() не змінить переданий аргумент array. Коли ви передаєте вказівник у функцію за адресою, то значення цього вказівника (адреса, на яку він вказує) *копіюється* з аргументу в параметр функції. Іншими словами, він *передається за значенням*. Якщо змінити значення параметру функції, то зміниться тільки копія, вихідний вказівник-аргумент не буде змінено.

Наприклад:

```
#include <iostream>
void setToNull(int *tempPtr)
{
    /* Ми присвоюємо tempPtr інше значення (ми не змінюємо значення, на яке вказує tempPtr)*/
    tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}
int main()
{
    // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
    int six = 6;
    int *ptr = &six;
```

```
    // Тут виведеться 6
    std::cout << *ptr << "\n";
    // tempPtr отримує копію ptr
    setToNull(ptr);
    // ptr до сих пір вказує на змінну six!
    // Тут виведеться 6
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";
    return 0;
}
```

У tempPtr копіюється адреса вказівника ptr. Незважаючи на те, що ми змінили tempPtr на нульовий вказівник (присвоїли йому nullptr), це ніяк не вплинуло на значення, на яке вказує ptr.

Отже, результат виконання програми:

```
6
6
```

Зверніть увагу! Хоча сама адреса передається за значенням, ви все одно можете розіменувати її для зміни значення вихідного аргументу.

В разі передачі аргументу за адресою в змінну-параметр функції копіюється адреса з аргументу. У цей момент параметр функції і аргумент вказують на одне і те ж значення.

Якщо параметр функції потім розіменувати для зміни початкового значення, то це призведе до зміни значення, на яке вказує аргумент, оскільки параметр функції і аргумент вказують на одне і те ж значення!

Якщо параметру функції присвоїти іншу адресу, то це ніяк не вплине на аргумент, оскільки параметр функції є копією, а зміна копії не призводить до зміни оригіналу. Після зміни адреси параметра функції, параметр функції і

аргумент вказуватимуть на різні значення, тому розіменування параметру і подальша його зміна ніяк не вплинуть на значення, на яке вказує аргумент.

У наступній програмі це все добре проілюстровано:

```
#include <iostream>
void setToSeven(int *tempPtr)
{
   *tempPtr = 7; // ми змінюємо значення, на яке вказує tempPtr (і ptr також)
}
int main()
{
   // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
   int six = 6;
   int *ptr = &six;
   // Тут виведеться 6
   std::cout << *ptr << "\n";
   // tempPtr отримує копію ptr
   setToSeven(ptr);
   // tempPtr змінив значення, на яке вказував, на 7
   // Тут виведеться 7
   if (ptr)
      std::cout << *ptr << "\n";
   else
      std::cout << " ptr is null";
   return 0;
}
```

Результат виконання програми:

```
6
7
```

### Передача адрес за посиланням

Виникає питання: «А що, якщо ми хочемо змінити адресу, на яку вказує аргумент, всередині функції?». Виявляється, це можна зробити дуже легко. Ви можете просто передати адресу за посиланням [9].

Наприклад:

```
#include <iostream>
// tempPtr тепер є посиланням на вказівник, тому будь-які зміни tempPtr призведуть і до зміни вихідного аргументу!
void setToNull(int *&tempPtr)
{
   tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}
int main()
{
   // Спочатку ми присвоюємо ptr адресу six, тобто *ptr = 6
   int six = 6;
   int *ptr = &six;
   // Тут виведеться 6
   std::cout << *ptr;
   // tempPtr є посиланням на ptr
   setToNull(ptr);
   // ptr було присвоєно значення nullptr!
   if (ptr)
      std::cout << *ptr;
   else
      std::cout << " ptr is null";
   return 0;
}
```

Результат виконання програми:

```
6 ptr is null
```

Нарешті, наша функція setToNull() дійсно змінила значення ptr з &six на nullptr.

**Плюси передачі за адресою:**

- Передача за адресою дозволяє функції змінити значення аргументу, що іноді корисно. В іншому випадку, використовуємо const для гарантії того, що функція не змінить аргумент.

- Оскільки копіювання аргументів не відбувається, то швидкість передачі за адресою досить висока, навіть якщо передавати великі структури або класи.

- Ми можемо повернути відразу декілька значень з функції, використовуючи параметри виведення.

**Мінуси передачі за адресою**:

- Всі вказівники потрібно перевіряти, чи не є вони нульовими. Спроба розіменувати нульовий вказівник призведе до збою в програмі.

- Оскільки розіменування вказівника виконується повільніше, ніж доступ до значення напряму, то доступ до аргументів, переданих за адресою, виконується також повільніше, ніж доступ до аргументів, переданих за значенням.

**Коли використовувати передачу за адресою**:

- у випадку передачі звичайних масивів (якщо немає ніяких проблем з тим, що масиви конвертуються у вказівники при передачі).

**Коли не використовувати передачу за адресою**:

- у разі передачі структур або класів (використовуйте передачу по посиланню);

- у разі передачі фундаментальних типів даних (використовуйте передачу за значенням).

Передача за адресою та за посиланням мають майже однакові переваги і недоліки. Оскільки передача за посиланням зазвичай безпечніша, ніж передача за адресою, то в більшості випадків краще використовувати передачу за посиланням.

## Повернення за значенням

**Повернення за значенням** – це найпростіший та найбезпечніший тип повернення [4]. У випадку поверненні за значенням, копія значення, що повертається, передається назад в caller. Як і у випадку з передачею за значенням, ви можете повертати літерали (наприклад, 7), змінні (наприклад, x) або вирази (наприклад, x + 2), що робить цей спосіб дуже гнучким.

Ще однією перевагою є те, що ви можете повертати змінні (або вирази), в обчисленні яких задіяні і локальні змінні, оголошені в тілі самої функції. При цьому, можна не турбуватися про проблеми, які можуть виникнути з областю видимості. Оскільки змінні обчислюються до того, як функція здійснює повернення значення, то тут не повинно бути ніяких проблем з областю видимості цих змінних, коли закінчується блок, в якому вони оголошені.

Наприклад:

```
int doubleValue(int a)
{
   int value = a * 3;
   return value; // копія value повертається тут
} // value виходить з області видимості тут
```

Повернення за значенням ідеально підходить для повернення змінних, які були оголошені всередині функції, або для повернення аргументів функції, які були передані за значенням. Однак, подібно до *передачі* за значенням, *повернення* за значенням повільне у випадку роботи зі структурами та класами.

**Коли використовувати повернення за значенням:**

- у випадку повернення змінних, які були оголошені всередині функції;
- у випадку повернення аргументів функції, які були передані в функцію за значенням.

**Коли не використовувати повернення за значенням:**

- в разі повернення **стандартних масивів** або **вказівників** (використовуйте повернення за адресою);

- в разі повернення великих структур або класів (використовуйте повернення за посиланням).

**Повернення за адресою**

**Повернення за адресою** – це повернення адреси змінної назад в caller [4]. Подібно передачі за адресою, повернення за адресою може повертати тільки адресу змінної. Літерали і вирази повертати не можна, так як вони не мають адрес. Оскільки при поверненні за адресою просто копіюється адреса з функції в caller, то цей процес також дуже швидкий.

Проте цей спосіб має один недолік, який відсутній при поверненні за значенням: якщо ви спробуєте повернути адресу локальної змінної, то отримаєте несподівані результати.

Наприклад:

```
int* doubleValue(int a)
{
   int value = a * 3;
   return &value; // value повертається за адресою тут
} // value знищується тут
```

Змінна value знищується відразу після того, як її адреса повертається в caller. Кінцевим результатом буде те, що caller отримає адресу звільненої пам’яті (**висячий вказівник**), що, безсумнівно, викличе проблеми. Це одна з найпоширеніших помилок, яку роблять початківці. Більшість сучасних компіляторів видадуть попередження (а не помилку), якщо програміст спробує повернути локальну змінну за адресою. Однак є кілька способів обдурити компілятор, щоб зробити щось “погане”, не генеруючи при цьому попередження, тому вся відповідальність лежить на програмісті, який повинен гарантувати, що адреса, що повертається, буде коректною.

Повернення за адресою часто використовують для повернення динамічно виділеної пам’яті назад в caller.

Наприклад:

```
int* allocateArray(int size)
{
   return new int[size];
}
int main()
{
   int *array = allocateArray(20);
   // Робимо що-небудь з array
   delete[] array;
   return 0;
}
```

Тут не виникне ніяких проблем, тому що динамічно виділена пам’ять не виходить з області видимості в кінці блоку, в якому вона оголошена, і все ще існуватиме, коли адреса повертатиметься в caller.

**Коли використовувати повернення за адресою:**

- у випадку повернення динамічно виділеної пам’яті;

- у випадку повернення аргументів функції, які були передані за адресою.

**Коли не використовувати повернення за адресою:**

- в разі повернення змінних, які були оголошені всередині функції (використовуйте повернення за значенням);

- в разі повернення великої структури або класу, які були передані за посиланням(використовуйте повернення по посиланню).

**Повернення за посиланням**

Подібно передачі за посиланням, значення, які **повертаються за посиланням**, повинні бути змінними (ви не зможете повернути посилання на літерал або вираз) [6]. У випадку повернення за посиланням в caller повертається посилання на змінну. Потім caller може її використовувати для

продовження модифікації змінної, що може бути іноді корисно. Цей спосіб також дуже швидкий й у випадку повернення великих структур або класів.

Однак, як й в разі повернення за адресою, ви не повинні повертати локальні змінні за посиланням. Розглянемо наступний фрагмент коду:

```
int& doubleValue(int a)
{
   int value = a * 3;
   return value; // value повертається за посиланням тут
} // value знищується тут
```

Тут повертається посилання на змінну value, яка буде знищена, коли функція завершить своє виконання. Це означає, що caller отримає посилання на сміття. На щастя, ваш компілятор, найімовірніше, видасть попередження або помилку, якщо ви спробуєте зробити подібне.

Повернення за посиланням зазвичай використовують для повернення аргументів, переданих у функцію за посиланням.

У наступному прикладі ми повертаємо (за посиланням) елемент масиву, який був переданий у функцію за посиланням:

```
#include <iostream>
#include <array>
// Повертаємо посилання на елемент масиву під індексом index
int& getElement(std::array<int, 20> &array, int index)
{
   /* Ми знаємо, що array[index] не знищиться, коли ми будемо повертати дані в caller (так як caller сам передав цей array у функцію!)*/
   // Так що тут не повинно бути ніяких проблем з поверненням по посиланню
   return array[index];
}
int main()
{
   std::array<int, 20> array;
```

```
    // Присвоюємо елементу масиву під індексом 15 значення 7
    getElement(array, 15) = 7;
    std::cout << array[15] << '\n';
    return 0;
}
```

Результат виконання програми:

```
7
```

Коли ми викликаємо getElement(array, 15), то getElement() повертає посилання на елемент масиву під індексом 15, а потім main() використовує це посилання для присвоювання значення 7 цьому елементу.

**Коли використовувати повернення за посиланням:**

- у випадку повернення посилання-параметру;

- у випадку повернення елементу масиву, який був переданий у функцію;

- у випадку повернення великої структури або класу, який не знищується в кінці функції (наприклад, той, який був переданий у функцію).

**Коли не використовувати повернення за посиланням:**

- в разі повернення змінних, які були оголошені всередині функції (використовуйте повернення за значенням);

- в разі повернення стандартного масиву або значення вказівника (використовуйте повернення за адресою).

## Перевантаження функцій

**Перевантаження функцій** – це можливість визначати декілька функцій з одним і тим самим ім’ям, але з різними параметрами [7].

Наприклад:

```
int subtract(int a, int b)
{
    return a - b;
}
```

Тут ми виконуємо операцію віднімання з цілими числами. Однак, що, якщо нам потрібно використати числа типу з плаваючою крапкою? Ця функція зовсім не підходить, так як будь-які параметри типу double будуть конвертуватися в тип int, в результаті чого дрібна частина значень губитиметься.

Одним із способів вирішення цієї проблеми є визначення двох функцій з різними іменами і параметрами.

Наприклад:

```
int subtractInteger(int a, int b)
{
   return a - b;
}
double subtractDouble(double a, double b)
{
   return a - b;
}
```

Але є краще рішення – *перевантаження* функції. Ми можемо просто визначити ще одну функцію subtract(), яка приймає параметри типу double.

Наприклад:

```
double subtract(double a, double b)
{
   return a - b;
}
```

Тепер у нас є дві версії функції subtract():

```
int subtract(int a, int b); // цілочисельна версія
double subtract(double a, double b); // версія типу з плаваючою крапкою
```

Компілятор може визначити сам, яку версію subtract() слід викликати на основі аргументів, які використовуються у виклику функції. Якщо параметрами будуть змінні типу int, то мова C++ розуміє, що ми хочемо викликати subtract(int, int). Якщо ж ми надамо два значення типу з плаваючою

крапкою, то мова C++ зрозуміє, що ми хочемо викликати subtract(double, double). Фактично, ми можемо визначити стільки перевантажених функцій subtract(), скільки хочемо, до тих пір, поки кожна з них матиме свої (унікальні) параметри.

Очевидно, функцію subtract() можна визначити і з більшою кількістю параметрів.

Наприклад:

```
int subtract(int a, int b, int c)
{
   return a - b - c;
}
```

Хоча тут subtract() має 3 параметри замість 2, це не є помилкою, оскільки ці параметри відрізняються від параметрів інших версій subtract().

**Виклики перевантаженої функції.**

Виконання виклику перевантаженої функції призводить до одного з трьох можливих результатів:

- **Збіг знайдено.** Виклик дозволений для відповідної перевантаженої функції.

- **Збіг не знайдено.** Аргументи не відповідають будь-якій з перевантажених функцій.

- **Знайдено декілька збігів.** Аргументи відповідають більше, ніж одній перевантаженій функції.

В процесі компіляції перевантаженої функції, C++ виконує наступні кроки для визначення того, яку версію функції слід викликати:

1. Намагається знайти точний збіг. Це той випадок, коли фактичний аргумент точно відповідає типу параметра однієї з перевантажених функцій.

Наприклад:

```
void print(char *value);
void print(int value);
print(0); // точний збіг з print(int)
```

Хоча 0 може технічно відповідати і print(char *) (як нульовий вказівник), але він точно відповідає print(int). Таким чином, print(int) є кращим (точним) збігом.

2. Якщо точного збігу не знайдено, то C++ намагається знайти збіг шляхом подальшого неявного перетворення типів:

- char, unsigned char і short конвертуються в int;

- unsigned short може конвертуватися в int або unsigned int (в залежності від розміру int);

- float конвертується в double;

- enum конвертується в int.

Наприклад:

```
void print(char *value);
void print(int value);
print('b'); // збіг з print(int) після неявної конвертації
```

В цьому випадку, оскільки немає print(char), символ b конвертується в тип int, який потім вже відповідає print(int).

3. Якщо неявне перетворення неможливе, то C++ намагається знайти відповідність за допомогою стандартного перетворення. У стандартному перетворенні:

- будь-який числовий тип відповідатиме будь-якому іншому числовому типу, включаючи unsigned (наприклад, int дорівнює float);

- enum відповідає формальному типу числового типу даних (наприклад, enum дорівнює float);

- нуль відповідає типу вказівника і числовому типу (наприклад, 0 як char * або 0 як float);

- вказівник відповідає вказівнику типу void.

4. C++ намагається знайти відповідність шляхом користувацької конвертації.

Наприклад, можна створити клас W і в ньому визначити для користувача конвертацію в тип int:

```
class W;            // з користувацькою конвертацією в тип int
void print(float value);
void print(int value);
W value;            // оголошуємо змінну value типу класу W
print(value);       // value конвертується в тип int і відповідає print(int)
```

Хоча value відноситься до типу класу W, але, оскільки той має користувацьку конвертацію в тип int, виклик print(value) відповідає версії print(int).

## Рекурсія

**Рекурсивна функція** (або просто ***"рекурсія"***) в мові C++ – це функція, яка викликає саму себе [2 – 5].

Наприклад:

```
#include <iostream>
void countOut(int count)
{
   std::cout << "push " << count << '\n';
   countOut(count-1); // функція countOut() рекурсивно викликає саму себе
}
int main()
{
   countOut(4);
   return 0;
}
```

В разі виклику функції countOut(4) на екран виведеться push 4, а потім буде виклик countOut(3). Далі countOut(3) виведе push 3 і викличе countOut(2). Послідовність виклику countOut(n) інших функцій countOut(n-1) повторюється нескінченну кількість разів (аналог нескінченного циклу).

## Умова завершення рекурсії

Рекурсивні виклики функцій працюють так само, як і звичайні виклики функцій. Однак, програма, наведена вище, ілюструє найбільш важливу

відмінність використання простих функцій від рекурсивних: ви повинні вказати умову завершення рекурсії, в протилежному випадку – функція виконуватиметься нескінченну кількість разів (фактично до тих пір, поки не закінчиться пам’ять у стеку викликів).

**Умова завершення рекурсії** – це умова, за якої рекурсивна функція перестане викликати саму себе. В цій умові зазвичай використовують оператор if**.**

Ось приклад функції, наведеної вище, але вже з умовою завершення рекурсії (і ще з одним додатковим виводом тексту на екран):

```
#include <iostream>
void countOut(int count)
{
   std::cout << "push " << count << '\n';
   if (count > 1) // умова завершення
      countOut(count-1);
   std::cout << "pop " << count << '\n';
}
int main()
{
   countOut(4);
   return 0;
}
```

Коли ми запустимо цю програму, то countOut() почне виводити:

```
push 4
push 3
push 2
push 1
```

Якщо зараз подивитися на стек викликів, то побачимо наступне:

```
countOut(1)
countOut(2)
```

countOut(3)
countOut(4)
main()

Через умову завершення, countOut(1) не викличе countOut(0): умова if не виконається, виведеться pop 1 та countOut(1) завершить своє виконання. На цьому етапі countOut(1) витягується зі стеку, і керування повертається до countOut(2). countOut(2) відновлює виконання в точці після виклику countOut(1), і тому виведеться pop 2, а потім countOut(2) завершиться. Рекурсивні виклики функцій countOut() поступово витягуються зі стеку до тих пір, поки не будуть видалені всі екземпляри countOut().

Тож, результат виконання програми, наведеної вище, наступний:

push 4
push 3
push 2
push 1
pop 1
pop 2
pop 3
pop 4

Варто відзначити, що push виводиться в порядку спадання, а pop – в порядку зростання. Справа у тому, що push виводиться до виклику рекурсивної функції, а pop виконується (виводиться) після виклику рекурсивної функції, коли всі екземпляри countOut() витягуються зі стеку (це відбувається в порядку, зворотному тому, в якому ці екземпляри були додані у стек).

## Рекурсивні алгоритми

Рекурсивні функції зазвичай вирішують проблему, спочатку знайшовши рішення для підмножин проблеми (рекурсивно), а потім модифікуючи це «підрішення», щоб дістатися вже до вірного рішення. У вищенаведеному прикладі, алгоритм sumCount(value) спочатку вирішує

sumCount(value-1), а потім додає значення value, щоб знайти рішення для sumCount(value).

У багатьох рекурсивних алгоритмах деякі дані введення видають передбачувані дані виведення. Наприклад, sumCount(1) має передбачуване виведення 1 (ви можете легко це обчислити і перевірити самостійно). Випадок, коли алгоритм за певних даних введення видає передбачувані дані виведення, називають **базовим випадком**. Базові випадки виконуються як умови для завершення виконання алгоритму. Їх часто можна ідентифікувати, розглядаючи результати виведення для таких значень введення як 0, 1, «» або null.

Найбільш популярне питання, яке задають про рекурсивні функції: «Навіщо використовувати рекурсивну функцію, якщо завдання можна виконати і за допомогою ітерацій (використовуючи цикл for чи цикл while)?». Виявляється, ви завжди можете вирішити рекурсивну проблему ітеративно. Однак, для нетривіальних випадків, рекурсивна версія часто буває набагато простіша як для написання, так і для читання. Наприклад, функцію обчислення n-го числа Фібоначчі можна написати і за допомогою ітерацій, але це складніше.

## Ітеративні функції

**Ітеративні функції** – ті, що використовують цикли for або while, – майже завжди більш ефективні, ніж їх рекурсивні аналоги. Це пов’язано з тим, що кожен раз, при виконанні функції, витрачається певна кількість ресурсів на додання і витягування фреймів зі стеку. Ітеративні функції витрачають набагато менше цих ресурсів.

Це не означає, що ітеративні функції завжди є кращим варіантом. Іноді рекурсивна реалізація може бути чистішою і простішою, а деякі додаткові витрати можуть бути більш ніж виправдані, звівши до мінімуму труднощі майбутньої підтримки коду, особливо, якщо алгоритм не вимагає занадто багато часу для пошуку рішення.

Рекурсія є хорошим вибором, якщо виконується більшість з наступних тверджень:

- рекурсивний код набагато простіше реалізувати;
- глибина рекурсії може бути обмежена;
- ітеративна версія алгоритму вимагає управління стеком даних;
- це не критична частина коду, яка напряму впливає на продуктивність програми.

Якщо рекурсивний алгоритм простіше реалізувати, то є сенс почати з рекурсії, а потім вже оптимізувати код в ітеративний алгоритм.

Рекомендується використовувати ітерацію замість рекурсії в тих випадках, коли це дійсно практичніше.

## Контрольні запитання

1. У чому полягає різниця між оголошенням (прототипом) та визначенням (реалізацією) функції? Чому виникає помилка компіляції, якщо викликати функцію до її визначення, не вказавши прототип?

2. Що таке «сигнатура функції»? Які елементи функції входять до її сигнатури, а які – ні (наприклад, чи входить тип поверненого значення)?

3. Для чого використовується ключове слово void у контексті функцій? Чи можна використовувати оператор return у функції типу void?

4. Як працює передача аргументів «за значенням» (by value)? Що відбудеться з оригінальною змінною в функції main, якщо змінити її копію всередині викликаної функції?

5. У чому суть передачі аргументів «за посиланням» (by reference), використовуючи оператор &? Наведіть класичний приклад задачі, де без посилань (або вказівників) не обійтися (наприклад, функція swap для обміну значень двох змінних).

6. Чому для передачі великих обсягів даних (наприклад, великих рядків тексту або структур) часто використовують константні посилання (const Type&)? Які дві проблеми це вирішує одночасно?

7. Яку роль відіграє оператор return? Чи може функція повернути кілька значень одночасно класичним способом (без використання структур, масивів чи посилань)?

8. Що станеться, якщо функція має тип повернення (наприклад, int), але програміст забув написати return або написав його не у всіх гілках if/else? Чому це вважається критичною помилкою (Undefined Behavior)?

9. Чому категорично заборонено повертати посилання на локальну змінну, створену всередині функції? Що таке «висяче посилання» (dangling reference) і як воно пов'язане зі стеком пам'яті?

10. Що таке рекурсивна функція? Чим вона відрізняється від звичайного циклічного виклику?

11. Що таке «базовий випадок» (умова виходу) в рекурсії? Що станеться з програмою, якщо забути прописати умову виходу (що таке переповнення стека – Stack Overflow)?

12. Порівняйте рекурсію та ітерацію (цикли). У яких випадках рекурсія робить код більш читабельним, і в чому її головний недолік з точки зору продуктивності та використання пам'яті?

## Лекція №6. Вказівники та адреси

**Вказівник** (або ***"покажчик"***) – це змінна, значенням якої є адреса комірки в пам'яті [2, 3]. Вказівники оголошуються так само, як і звичайні змінні, тільки із зірочкою між типом даних та ідентифікатором.

Наприклад:

```
int *iPtr;          // вказівник на значення типу int
double *dPtr;       // вказівник на значення типу double
```

Синтаксично мова C++ приймає оголошення вказівника, коли зірочка знаходиться поруч із типом даних, із ідентифікатором або навіть посередині.

Наприклад:

```
int* iPtr3;         // коректний синтаксис (дозволено, але не бажано)
int * iPtr4;        // коректний синтаксис (не робіть так)
```

Зверніть увагу! Ця зірочка НЕ є оператором розіменування. Це всього лише частина синтаксису оголошення вказівника.

Однак, при оголошенні кількох вказівників, зірочка повинна знаходитися біля кожного ідентифікатора. Це легко забути, якщо ви звикли вказувати зірочку біля типу даних, а не біля імені змінної.

Наприклад:

```
int *iPtr5, *iPtr6;  // оголошуємо два вказівники для змінних типу int
int* iPtr3, iPtr4; /* iPtr3 – це вказівник на значення типу int, а iPtr4 – це звичайна змінна типу int*/
```

З цієї причини, в оголошенні вказівника рекомендовано вказувати зірочку біля імені змінної.

Як і звичайні змінні, вказівники не ініціалізуються при оголошенні. Вмістом неініціалізованого вказівника є звичайне «сміття».

### Присвоювання значень вказівнику

Оскільки вказівники містять тільки адреси, то в разі присвоювання значення вказівнику – це значення повинно бути адресою [6]. Для отримання адреси змінної використовують оператор адреси (&).

Наприклад:

```
int value = 5;
int *ptr = &value;          // ініціалізуємо ptr адресою значення змінної
```

Ось чому вказівники мають таку назву: ptr містить адресу значення змінної value, і, можна сказати, ptr *вказує* на це значення.

Ще дуже часто можна побачити наступне:

```
#include <iostream>
int main()
{
  int value = 5;
  int *ptr = &value;              // ініціалізуємо ptr адресою значення змінної
  std::cout << &value << '\n'; // виводимо адресу значення змінної value
  std::cout << ptr << '\n';       // виводимо адресу, яку містить ptr
  return 0;
}
```

Результат виконання програми:

```
003AFCD4
003AFCD4
```

Тип вказівника повинен відповідати типу змінної, на яку він вказує.

Наприклад:

```
int iValue = 7;
double dValue = 9.0;
int *iPtr = &iValue;        // ок
double *dPtr = &dValue; // ок
iPtr = &dValue; /*неправильно: вказівник типу int не може вказувати на адресу змінної типу double*/
dPtr = &iValue; /*неправильно: вказівник типу double не може вказувати на адресу змінної типу int*/
```

Наступне не є допустимим:

```
int *ptr = 7;
```

Це пов’язано з тим, що вказівники можуть містити тільки адреси, а цілочисельний літерал 7 не має адреси в пам’яті. Якщо ви все ж зробите це, то компілятор повідомить вам, що він не може перетворити цілочисельне значення в цілочисельний вказівник.

**Оператор адреси повертає вказівник**. Варто зазначити, що оператор адреси & не повертає адресу свого операнда в якості літералу. Замість цього він повертає вказівник, що містить адресу операнда, тип якого отримано з аргументу (наприклад, адреса змінної типу int передається як адреса вказівника на значення типу int).

Наприклад:

```
#include <iostream>
#include <typeinfo>
int main()
{
	int x(4);
	std::cout << typeid(&x).name();
	return 0;
}
```

Результат виконання програми:

```
int *
```

**Розіменування вказівників**

Як тільки у нас є вказівник, який вказує на що-небудь, ми можемо його розіменувати, щоб отримати значення, на яке він вказує. **Розіменований вказівник** – це вміст комірки в пам’яті, на яку він вказує [5].

Наприклад:

```
#include <iostream>
int main()
{
	int value = 5;
	std::cout << &value << std::endl; // виводимо адресу value
```

```
    std::cout << value << std::endl; // виводимо вміст value
    int *ptr = &value; // ptr вказує на value
    std::cout << ptr << std::endl; /* виводимо адресу, яка зберігається в ptr (тобто &value) */
    std::cout << *ptr << std::endl; /* розіменовуємо ptr (отримуємо значення, на яке вказує ptr) */
    return 0;
}
```

Результат виконання програми:

```
0034FD90
5
0034FD90
5
```

Ось чому вказівники повинні мати тип даних. Без типу вказівник не знав би, як інтерпретувати вміст, на який він вказує (в разі розіменування). Разом із тим, повинні збігатися тип вказівника і типо змінної. Якщо вони не збігатимуться, то вказівник в разі розіменування може неправильно інтерпретувати біти (наприклад, замість типу double використати тип int).

Одному вказівнику можна присвоювати різні значення.

Наприклад:

```
int value1 = 5;
int value2 = 7;
int *ptr;
ptr = &value1;      // ptr вказує на value1
std::cout << *ptr;  // виведеться 5
ptr = &value2;      // ptr тепер вказує на value2
std::cout << *ptr;  // виведеться 7
```

Коли адреса значення змінної присвоєна вказівнику, то виконується наступне:

- ptr – це те ж саме, що і &value;

- *ptr обробляється так само, як і value.

Оскільки *ptr обробляється так само, як і value, то ми можемо присвоювати йому значення так, наче це звичайна змінна.

Наприклад:

```
int value = 5;
int *ptr = &value; // ptr вказує на value
*ptr = 7; // *ptr - це те ж саме, що і value, якому ми присвоїли значення 7
std::cout << value; // виведеться 7
```

**Розіменування некоректних вказівників**

Вказівники в мові C++ по своїй суті є небезпечними, а їх неправильне використання – один з найкращих способів отримати збій в програмі.

В разі розіменування вказівника, програма намагається перейти в комірку в пам’яті, яка зберігається у вказівнику і “витягнути” вміст цієї комірки [5, 7]. З міркувань безпеки сучасні операційні системи (ОС) запускають програми безпечно та ізольовано для запобігання їх неправильної взаємодії з іншими програмами і для захисту стабільності самої операційної системи. Якщо програма спробує отримати доступ до комірки в пам’яті, не виділеної для неї операційною системою, то ОС відразу завершить виконання цієї програми.

Наступна програма добре ілюструє вищесказане:

```
#include <iostream>
void foo(int *&p)
{
}
int main()
{
  int *p; // створюємо неініціалізований вказівник (вмістом якого є “сміття”)
  foo(p); /* вводимо компілятор в оману, ніби ми збираємося присвоїти вказівнику коректне значення */
```

```
    std::cout << *p; // розіменовуємо вказівник зі “сміттям”
    return 0;
}
```

При запуску цієї програми буде отримано збій.

## Розмір вказівників

**Розмір вказівника** залежить від архітектури, на якій скомпільовано виконуваний файл: 32-бітний виконуваний файл використовує 32-бітні адреси в пам’яті. Відповідно, вказівник на 32-бітному пристрої займає 32 біти (4 байти). З 64-бітним виконуваним файлом вказівник займатиме 64 біти (8 байтів). І це незалежно від того, на що вказує вказівник/

Наприклад:

```
char *chPtr;          // тип char займає 1 байт
int *iPtr;            // тип int займає 4 байти
struct Something
{
    int nX, nY, nZ;
};
Something *somethingPtr;
std::cout << sizeof(chPtr) << '\n';          // виведеться 4
std::cout << sizeof(iPtr) << '\n';           // виведеться 4
std::cout << sizeof(somethingPtr) << '\n'; // виведеться 4
```

Розмір вказівника завжди один і той самий. Це пов’язано з тим, що вказівник – це всього лише адреса в пам’яті, а кількість біт, необхідна для доступу до адреси в пам’яті на певному пристрої, – завжди постійна.

**Вказівники корисні в наступних випадках**:

1. Масиви реалізовані за допомогою вказівників. Вказівники можуть використовуватися для ітерації по масиву.

2*:* Вказівники є єдиним способом динамічного виділення пам’яті в C++. Це, безумовно, найбільш поширений варіант використання вказівників.

3: Вказівники можуть використовуватися для передачі великої кількості даних в функцію без копіювання цих даних.

4: Вказівники можуть використовуватися для передачі однієї функції в якості параметра іншій функції.

5: Вказівники використовують для досягнення поліморфізму при роботі з наслідуванням.

6: Вказівники можуть використовуватися для представлення однієї структури/класу в іншій структурі/класі, формуючи, таким чином, “ланцюжки”.

**Нульове значення і нульові вказівники**

Крім адрес в пам’яті, є ще одне значення, яке вказівник може зберігати: значення null. **Нульове значення** (або ***“null-значення”***) – це спеціальне значення, яке означає, що вказівник ні на що не вказує. Вказівник, що містить значення null, називають **нульовим вказівником**.

У мові C++ ми можемо присвоїти вказівнику нульове значення, ініціалізувавши його/присвоюючи йому літерал 0.

Наприклад:

```
int *ptr(0);   // ptr тепер є нульовим вказівником
int *ptr1;     // ptr1 не ініціалізований
ptr1 = 0;      // ptr1 тепер є нульовим вказівником
```

Оскільки значенням нульового вказівника є нуль, то це можна використовувати всередині умовного розгалуження для перевірки того, чи є вказівник нульовим чи ні.

Наприклад:

```
#include <iostream>
int main()
{
    double *ptr(0);
    if (ptr)
        std::cout << "ptr is pointing to a double value.";
```

```
    else
        std::cout << "ptr is a null pointer.";
    return 0;
}
```

**Розіменування нульових вказівників**

Розіменування вказівників зі “сміттям” призведе до несподіваних результатів. З розіменуванням нульового вказівника все аналогічно: у більшості випадків ви отримаєте збій в програмі. У цьому є сенс, адже розіменування вказівника означає, що потрібно «перейти до адреси, на яку вказує вказівник, і дістати з цієї адреси значення». Нульовий вказівник не має адреси, тому й такий результат.

**Схожість між вказівниками і масивами**

Фіксований масив визначають наступним чином:

```
int array[4] = { 5, 8, 6, 4 }; /*визначаємо фіксований масив, який містить 4 цілих числа*/
```

Для нас це масив з 4 цілих чисел, але для компілятора array є змінною типу int[4]. Ми знаємо, що array[0] = 5, array[1] = 8, array[2] = 6 і array[3] = 4. Але яке значення має сам array? Змінна array містить адресу першого елемента масиву так, наче це вказівник.

Наприклад:

```
#include <iostream>
int main()
{
   int array[4] = { 5, 8, 6, 4 };
   // Виводимо на екран значення масиву (змінну array)
   std::cout << "The array has address: " << array << '\n';
   // Виводимо на екран адресу елемента масиву
   std::cout << "Element 0 has address: " << &array[0] << '\n';
   return 0;
}
```

Результат виконання програми:

```
The array has address: 004BF968
Element 0 has address: 004BF968
```

Адреса, що зберігається в змінній array, є адресою першого елемента масиву.

Поширена помилка думати, що змінна array і вказівник на array є одним і тим же об’єктом. Це не так. Хоча обидва вказують на перший елемент масиву, інформація про тип даних у них різна. У вищенаведеному прикладі типом змінної array є int[4], тоді як типом вказівника на масив є int *.

Плутанина викликана тим, що в багатьох випадках, в процесі обчислення, фіксований масив **розпадається** (неявно перетворюється) у вказівник на перший елемент масиву. Доступ до елементів як і раніше здійснюється через вказівник, але інформація, отримана з типу масиву (наприклад, його розмір), не може бути доступна з типу вказівника.

Однак і це не є настільки вагомим аргументом, щоб розглядати фіксовані масиви і вказівники як різні значення.

Наприклад, ми можемо розіменувати масив, щоб отримати значення першого елементу:

```
int array[4] = { 5, 8, 6, 4 };
/* Розіменування масиву (змінної array) призведе до повернення першого елемента масиву (елементу під номером 0) */
std::cout << *array;        // виведеться 5
char name[] = "John";       // рядок C-style (також масив)
std::cout << *name;         // виведеться 'J'
```

Ми не розіменовуємо фактичний масив. Масив (типу int[4]) неявно конвертується у вказівник (типу int *), і ми розіменовуємо вказівник, який вказує на значення першого елемента масиву.

Також ми можемо створити вказівник і присвоїти йому array.

Наприклад:

```
#include <iostream>
```

```
int main()
{
	int array[4] = { 5, 8, 6, 4 };
	std::cout << *array;		// виведеться 5
	int *ptr = array;
	std::cout << *ptr;		// виведеться 5
	return 0;
}
```

Це працює через те, що змінна array розпадається у вказівник типу int *, а тип нашого вказівника такий самий (int *).

**Відмінності між вказівниками і масивами**

Однак є випадки, коли різниця між фіксованими масивами і вказівниками має значення. Основна відмінність виникає в разі використання оператора sizeof. У випадку використання в фіксованому масиві, оператор sizeof повертає **розмір всього масиву** (довжина_масиву * розмір_елементу). У випадку використання з вказівником, оператор sizeof повертає **розмір адреси в пам'яті (в байтах)**.

Наприклад:

```
#include <iostream>
int main()
{
	int array[4] = { 5, 8, 6, 4 };
	std::cout << sizeof(array) << '\n'; // виведеться довжина array
	int *ptr = array;
	std::cout << sizeof(ptr) << '\n'; // виведеться розмір вказівника
	return 0;
}
```

Результат виконання програми:

```
16
4
```

Фіксований масив знає свою довжину, а вказівник на масив – ні.

Інша відмінність виникає в разі використання оператора адреси (&). Використовуючи адресу вказівника, ми отримуємо адресу пам’яті змінної вказівника. Використовуючи адресу масива, повертається вказівник на весь масив. Цей вказівник також вказує на перший елемент масиву, але інформація про тип відрізняється. Навряд чи вам коли-небудь доведеться це використовувати.

**Передача масивів у функції**

В мові C++ масив не передається у функцію цілком як єдина копія. Замість цього C++ використовує оптимізацію. У випадку передачі масиву в функцію в якості аргумента масив автоматично «перетворюється» на вказівник, який вказує на перший елемент цього масиву. І цей вказівник передається у функцію, в результаті чого функція отримує не копію всього масиву, а лише адресу його початку.

Наприклда:

```
#include <iostream>
void printSize(int *array)
{
   // Тут масив розглядається як вказівник
   std::cout << sizeof(array) << '\n'; /* виведеться розмір вказівника, а не довжина масиву */
}
int main()
{
   int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
   std::cout << sizeof(array) << '\n'; // виведеться довжина масиву
   printSize(array); // тут аргумент array розглядається як вказівник
    return 0;
}
```

Результат виконання програми:

```
32
4
```

Результат буде таким самим, навіть якщо параметром буде фіксований масив.

Наприклад:

```
#include <iostream>
// C++ неявно конвертує array[] в *array
void printSize(int array[])
{
   // Тут масив розглядається як вказівник, а не як фіксований масив
   std::cout << sizeof(array) << '\n'; /* виведеться розмір вказівника, а не розмір масиву */
}
int main()
{
   int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
   std::cout << sizeof(array) << '\n'; // виведеться sizeof(int) * довжина масиву array
   printSize(array); // тут аргумент array розглядається як вказівник
    return 0;
}
```

Результат виконання програми:

```
32
4
```

У вищенаведеному прикладі C++ неявно конвертує параметр з синтаксису масиву ([ ]) в синтаксис вказівника (*). Це означає, що наступні два оголошення функції ідентичні:

```
void printSize(int array[]);
void printSize(int *array);
```

В більшості випадків, оскільки вказівник не знає, наскільки великим є масив, вам доведеться передавати розмір масиву в якості окремого параметру (рядки є винятком, тому що вони нуль-терміновані).

Рекомендується використовувати синтаксис вказівника, оскільки він дозволяє зрозуміти, що параметр оброблятиметься як вказівник, а не як фіксований масив, і певні операції, такі як у випадку з оператором sizeof, виконуватимуться з параметром-вказівником (а не з параметром-масивом).

**Передача за адресою**

Той факт, що масиви перетворюються на вказівники в разі передачі у функції, пояснює основну причину, за якої зміна масиву в функції призведе до зміни фактичного масиву [5].

Розглянемо наступний приклад:

```
#include <iostream>
// Параметр ptr містить копію адреси масиву
void changeArray(int *ptr)
{
   *ptr = 5; // тому зміна елемента масиву призведе до зміни фактичного масиву
}
int main()
{
   int array[] = { 1, 2, 3, 4, 4, 9, 15, 25 };
   std::cout << "Element 0 has value: " << array[0] << '\n';
   changeArray(array);
   std::cout << "Element 0 has value: " << array[0] << '\n';
    return 0;
}
```

Результат виконання програми:

```
Element 0 has value: 1
Element 0 has value: 5
```

В разі виклику функції changeArray(), масив перетворюється на вказівник, а значення цього вказівника (адреса пам’яті першого елемента масиву) копіюється в параметр ptr функції changeArray(). Хоча значення ptr в функції є копією адреси масиву, ptr все рівно вказує на фактичний масив (а не на копію!). Тому при розіменуванні ptr, розіменується і фактичний масив.

Цей феномен працює так само і з вказівниками на значення не з масиву.

**Масиви в структурах і класах**

Масиви, які є частиною структур або класів, не розпадаються, коли вся структура або клас передається в функцію.

**Адресна арифметика**

Зауважимо, що ptr + 1 не повертає наступну будь-яку адресу в пам’яті, яка знаходиться відразу після ptr, але повертає адресу наступного об’єкта в пам’яті, тип якого збігається з типом значення, на яке вказує ptr. Якщо ptr вказує на адресу цілочисельного значення в пам’яті (розмір якого складає 4 байти), то ptr + 3 повертатиме адресу третього цілочисельного значення після ptr в пам’яті. Якщо ptr вказує на адресу значення типу char в пам’яті, то ptr + 3 повертатиме адресу третього значення типу char в пам’яті після ptr.

При обчисленні результату виразу **адресної арифметики** (або ***“арифметики з вказівниками”***) компілятор завжди множить цілочисельний операнд на розмір об’єкта, на який вказує вказівник [3 – 6].

Наприклад:

```
#include <iostream>
int main()
{
   int value = 8;
   int *ptr = &value;
   std::cout << ptr << '\n';
   std::cout << ptr+1 << '\n';
   std::cout << ptr+2 << '\n';
   std::cout << ptr+3 << '\n';
```

```
    return 0;
}
```

Результат виконання програми:

```
002CF9A4
002CF9A8
002CF9AC
002CF9B0
```

Кожна наступна адреса збільшується на 4. Це пов’язано з тим, що розмір типу int на моєму комп’ютері становить 4 байти.

Ось та сама програма, але з використанням типу short замість типу int:

```
#include <iostream>
int main()
{
    short value = 8;
    short *ptr = &value;
    std::cout << ptr << '\n';
    std::cout << ptr+1 << '\n';
    std::cout << ptr+2 << '\n';
    std::cout << ptr+3 << '\n';
    return 0;
}
```

Результат виконання програми:

```
002BFA20
002BFA22
002BFA24
002BFA26
```

Оскільки тип short займає 2 байти, то кожна наступна адреса збільшується на 2.

### Розташування елементів масиву в пам’яті

Використовуючи оператор адреси (&), ми можемо легко визначити, що елементи масиву розташовані в пам’яті послідовно.

Наприклад:

```
#include <iostream>
int main()
{
   int array[] = { 7, 8, 2, 4, 5 };
   std::cout << "Element 0 is at address: " << &array[0] << '\n';
   std::cout << "Element 1 is at address: " << &array[1] << '\n';
   std::cout << "Element 2 is at address: " << &array[2] << '\n';
   std::cout << "Element 3 is at address: " << &array[3] << '\n';
   return 0;
}
```

Результат виконання програми:

```
Element 0 is at address: 002CF6F4
Element 1 is at address: 002CF6F8
Element 2 is at address: 002CF6FC
Element 3 is at address: 002CF700
```

Кожна з цих адрес окремо займає 4 байти, як і розмір типу int на моєму комп’ютері.

### Індексація масивів

Елементи масиву розташовані в пам’яті послідовно. Фіксований масив може конвертуватися на вказівник, який вказує на перший елемент (елемент під індексом 0) масиву.

Додавання одиниці до вказівника повертає адресу в пам’яті наступного об’єкта цього ж типу даних. Додавання одиниці до ідентифікатора масиву призведе до повернення адреси другого елемента (елементу під індексом 1) масиву в пам’яті.

Наприклад:

```
#include <iostream>
int main()
{
    int array [5] = { 7, 8, 2, 4, 5 };
    std::cout << &array[1] << '\n'; // виведеться адреса в пам'яті елементу під індексом 1
    std::cout << array+1 << '\n'; // виведеться адреса в пам'яті вказівника на масив + 1
    std::cout << array[1] << '\n'; // виведеться 8
    std::cout << *(array+1) << '\n'; // виведеться 8 (зверніть увагу на дужки, вони тут обов'язкові)
   return 0;
}
```

При розіменуванні результату виразу адресної арифметики дужки необхідні для дотримання пріоритету операцій, оскільки оператор (*) має більший пріоритет, ніж оператор (+).

Результат виконання програми:

```
001AFE74
001AFE74
8
8
```

Виявляється, коли компілятор бачить оператор індексу [], він, насправді, конвертує його у вказівник з операцією додавання і розіменування! Тобто, array[n] – це те ж саме, що *(array + n), де n є цілочисельним значенням. Оператор індексу [] використовується в цілях зручності, щоб не потрібно було завжди пам’ятати про дужки.

**Використання вказівника для ітерації по масиву**

Ми можемо використовувати вказівники і адресну арифметику для виконання ітерацій по масиву. Хоча зазвичай це не робиться (використання

оператора індексу, як правило, більш читабельне і менш вразливе до помилок).

Наступний приклад показує, що це можливо:

```
#include <iostream>
int main()
{
	const int arrayLength = 9;
	char name[arrayLength] = "Jonathan";
	int numVowels(0);
	for (char *ptr = name; ptr < name + arrayLength; ++ptr)
	{
		switch (*ptr)
		{
		case 'A':
		case 'a':
		case 'E':
		case 'e':
		case 'I':
		case 'i':
		case 'O':
		case 'o':
		case 'U':
		case 'u':
			++numVowels;
		}
	}
	std::cout << name << " has " << numVowels << " vowels.\n";
	return 0;
}
```

Програма використовує вказівник для прогону кожного елемента масиву по черзі. Пам’ятаєте, що масив конвертується у вказівник на перший елемент масиву? Відповідно, присвоївши name для ptr, сам ptr став вказувати на перший елемент масиву. Кожен елемент розіменовується за допомогою кейсу switch, та, якщо поточний елемент масиву є голосною буквою, numVowels збільшується. Для переміщення вказівника до наступного символу (елементу) масиву в циклі for використовується оператор ++. Цикл завершиться, коли всі символи будуть перевірені.

Результат виконання програми:

Jonathan has 3 vowels.

## Контрольні запитання

1. Що таке вказівник (pointer) у C++? Яку саме інформацію він зберігає та чим концептуально відрізняється від звичайної змінної?

2. Поясніть призначення оператора отримання адреси (&) та оператора розіменування (*). Наведіть приклад коду, де змінній присвоюється значення через вказівник на неї.

3. Що таке «нульовий вказівник»? Чому в сучасному C++ (починаючи зі стандарту C++11) настійно рекомендується використовувати ключове слово nullptr замість макроса NULL або числа 0?

4. Що станеться, якщо спробувати розіменувати неініціалізований вказівник (який вказує на випадкову ділянку пам’яті) або нульовий вказівник?

5. Як працює додавання цілого числа до вказівника (наприклад, ptr + 1)? Якщо ptr вказує на змінну типу int (яка займає 4 байти), на скільки байтів фактично зміститься адреса в пам'яті після виконання цієї операції?

6. Як компілятор сприймає ім’я статичного масиву у більшості виразів? На який саме елемент масиву вказує це ім’я за замовчуванням?

7. Продемонструйте еквівалентність синтаксису: як звернутися до i-го елемента масиву arr, використовуючи вказівну арифметику та оператор розіменування замість звичних квадратних дужок arr[i]?

8. У чому полягає фундаментальна різниця між вказівником та іменем масиву? Чи можна змінити адресу, на яку вказує ім’я фіксованого масиву (наприклад, написати arr = ptr;)?

9. Що покаже оператор sizeof, якщо застосувати його до масиву int arr[10], і якщо застосувати його до вказівника int* ptr = arr? Поясніть, чому результати будуть різними.

## Лекція №7. Динамічна пам’ять

Як статичне, так і автоматичне виділення пам’яті має дві загальні властивості [8, 9]:

- Розмір змінної/масиву повинен бути відомий під час компіляції.

- Виділення і звільнення пам’яті відбувається автоматично (коли змінна створюється/знищується).

Коли справа доходить до роботи з користувацьким вводом, то ці обмеження можуть призвести до проблем.

Наприклад, при використанні рядка для зберігання імені користувача, ми не знаємо наперед наскільки довгим воно буде, поки користувач його не введе. Або нам потрібно створити гру з непостійною кількістю монстрів (під час гри одні монстри вмирають, інші з’являються, намагаючись, таким чином, вбити гравця).

Якщо нам потрібно оголосити розмір всіх змінних під час компіляції, то найкраще, що ми можемо зробити – це спробувати вгадати їх максимальний розмір, сподіваючись, що цього буде достатньо.

Наприклад:

```
char name[30]; /* будемо сподіватися, що користувач введе ім'я довжиною не більше 30 символів*/
Monster monster[30]; // 30 монстрів максимум
Polygon rendering[40000]; /* цьому рендерингу краще містити не більше 40000 полігонів*/
```

Це погане рішення, принаймні, з трьох причин:

По-перше, втрачається пам’ять, якщо змінні фактично не використовуються або використовуються, але не всі. Наприклад, якщо ми виділимо 30 символів для кожного імені, але імена в середньому займатимуть по 15 символів, то споживання пам’яті буде в два рази більше, ніж нам насправді потрібно. Або розглянемо масив rendering: якщо він використовує тільки 20 000 полігонів, то пам’ять для інших 20 000 полігонів фактично витрачається даремно (тобто не використовується).

По-друге, пам’ять для більшості звичайних змінних (включаючи фіксовані масиви) виділяється зі спеціального резервуара пам’яті – **стеку**. Обсяг пам’яті стеку в програмі, як правило, невеликий: в Visual Studio він за замовчуванням становить 1 МБ. Якщо ви перевищите це значення, то відбудеться *переповнення стеку*, і операційна система автоматично завершить виконання програми.

Наприклад:

```
int main()
{
   int array[1000000000]; // виділяємо 1 мільярд цілочисельних значень
}
```

Ліміт в 1 МБ пам’яті може бути проблематичним для багатьох програм, особливо для тих, де використовується графіка.

По-третє, і найголовніше, – це може призвести до штучних обмежень і/або переповнення масиву. Що станеться, якщо користувач спробує прочитати 500 записів з диску, але ми виділили пам’ять максимум для 400? Або ми виведемо користувачеві помилку, що максимальна кількість записів становить 400, або (в гіршому випадку) виконається переповнення масиву і потім щось дуже нехороше.

На щастя, ці проблеми легко вирішуються за допомогою динамічного виділення пам’яті. **Динамічне виділення пам’яті** – це спосіб запиту пам’яті з операційної системи запущеними програмами в міру необхідності [7]. Ця пам’ять не виділяється з обмеженої пам’яті стеку програми, а виділяється з набагато більшого сховища, керованого операційною системою – **купи**. На сучасних комп’ютерах розмір купи може становити гігабайти пам’яті.

Для динамічного виділення пам’яті для змінної використовують **оператор new**.

Наприклад:

```
new int; /* динамічно виділяємо цілочисельну змінну і відразу ж відкидаємо результат (так як ніде його не зберігаємо)*/
```

У прикладі, наведеному вище, ми робимо запит на виділення пам’яті з операційної системи для цілочисельної змінної. Оператор new повертає **вказівник**, що містить **адресу** виділеної пам’яті.

Тож, для доступу до виділеної пам’яті створюємо вказівник:

int *ptr = new int; /* *динамічно виділяємо цілочисельну змінну і присвоюємо її адресу для ptr, щоб потім мати доступ до неї*/

Потім ми можемо розіменувати вказівник для отримання значення:

*ptr = 8; // *присвоюємо значення 8 тільки що виділеній пам’яті*

Без вказівника з адресою на тільки що виділену пам’ять у нас не було б способу отримати доступ до неї.

**Як працює динамічне виділення пам’яті?**

На комп’ютері є пам’ять (можливо, більша її частина), яка доступна для використання програмами. При запуску програми операційна система завантажує її в деяку частину цієї пам’яті. І ця пам’ять, яка використовується вашою програмою, розділена на кілька частин, кожна з яких виконує певне завдання. Одна частина містить код, інша використовується для виконання звичайних операцій (відстеження функцій, створення і знищення глобальних і локальних змінних тощо). Велика частина доступної пам’яті комп’ютера просто знаходиться в очікуванні запитів на виділення від програм.

Коли пам’ять виділяється динамічно, то операційна система резервує частину цієї пам’яті для використання програмою. Якщо ОС може виконати цей запит, то повертається адреса цієї пам’яті назад в вашу програму. З цього моменту і надалі ваша програма зможе використовувати цю пам’ять, як тільки побажає. Коли ви вже виконали з цією пам’яттю все, що було необхідно, то її потрібно повернути назад в операційну систему для розподілу між іншими запитами на виділення.

На відміну від статичного або автоматичного виділення пам’яті, програма самостійно відповідає за запит і зворотне повернення динамічно виділеної пам’яті.

### Звільнення пам’яті

Коли ви динамічно виділяєте змінну, то ви також можете її ініціалізувати за допомогою **прямої ініціалізації** або **uniform-ініціалізації** [4].

Наприклад:

```
int *ptr1 = new int (7); // використовуємо пряму ініціалізацію
int *ptr2 = new int { 8 }; // використовуємо uniform-ініціалізацію
```

Коли вже все, що було потрібно, виконано з динамічно виділеною змінної — потрібно явно вказати для C++ звільнити цю пам’ять. Для змінних це виконується за допомогою **оператора delete**.

Наприклад:

```
// Припустимо, що ptr раніше уже був виділений за допомогою оператора new
delete ptr; // повертаємо пам'ять, на яку вказував ptr, назад в операційну систему
ptr = 0; /* робимо ptr нульовим вказівником (використовуйте nullptr замість 0 в C++11 і вище)*/
```

Оператор delete насправді нічого не видаляє. Він просто повертає пам’ять, яка була виділена раніше, назад в операційну систему. Потім операційна система може перепризначити цю пам’ять іншому додатку/програмі (або цій же програмі знову).

Хоча може здатися, що ми видаляємо *змінну*, але це не так! Змінна-вказівник має ту ж область видимості, що і раніше, і їй можна присвоїти нове значення, як і будь-якій іншій змінній.

Зверніть увагу, видалення вказівника, який не вказує на динамічно виділену пам’ять, може призвести до проблем.

### Висячі вказівники

Мова C++ не надає жодних гарантій щодо того, що станеться з вмістом звільненої пам’яті або зі значенням вказівника, який ви видалили. У більшості випадків, пам’ять, яка повертається назад в операційну систему, міститиме ті ж значення, які були у неї до звільнення, а вказівник так і залишиться вказувати на тільки що звільнену (видалену) пам’ять.

Вказівник, який вказує на звільнену пам’ять, називають **висячим вказівником**. Розіменування або видалення висячого вказівника призведе до несподіваних результатів.

Розглянемо наступну програму:

```
#include <iostream>
int main()
{
   int *ptr = new int; // динамічно виділяємо цілочисельну змінну
   *ptr = 8; // поміщаємо значення у виділену комірку пам'яті
   delete ptr; /* повертаємо пам'ять назад в операційну систему, ptr тепер є висячим вказівником*/
   std::cout << *ptr; /* розіменування висячого вказівника призведе до несподіваних результатів*/
   delete ptr; /* спроба звільнити пам'ять знову призведе до несподіваних результатів*/
   return 0;
}
```

У вищенаведеній програмі значення 8, яке раніше було присвоєно динамічній змінній, після звільнення може і далі перебувати там, а може і ні. Також можливо, що звільнена пам’ять вже могла бути виділена іншому додатку/програмі (або залишена для власного використання операційної системи), і спроба доступу до неї призведе до того, що операційна система автоматично припинить виконання вашої програми.

Процес звільнення пам’яті може також призвести і до створення *декількох* висячих вказівників.

Розглянемо наступний приклад:

```
#include <iostream>
int main()
{
   int *ptr = new int; // динамічно виділяємо цілочисельну змінну
```

```
    int *otherPtr = ptr; /* otherPtr тепер вказує на ту ж саму виділену пам'ять, що й ptr*/
    delete ptr; /* повертаємо пам'ять назад в операційну систему. ptr і otherPtr тепер є висячими вказівниками*/
    ptr = 0; // ptr тепер уже nullptr
    // Проте, otherPtr як і раніше є висячим вказівником!
    return 0;
}
```

Є кілька рекомендацій:

По-перше, намагайтеся уникати ситуацій, коли кілька вказівників вказують на одну і ту ж частину виділеної пам'яті. Якщо це неможливо, то з'ясуйте, який з усіх вказівників «володіє» пам'яттю (і відповідає за її видалення), а які вказівники просто отримують до неї доступ.

По-друге, коли ви видаляєте вказівник, і, якщо він не виходить з області видимості відразу ж після видалення, його потрібно зробити нульовим (тобто присвоїти йому значення 0 або nullptr). Під "виходом з області видимості відразу ж після видалення" мається на увазі, що ви видаляєте вказівник в самому кінці блоку, в якому він оголошений.

Зверніть увагу! Присвоюйте видаленим вказівникам значення 0 (або nullptr), якщо вони не виходять з області видимості відразу ж після видалення.

**Оператор new**

При запиті пам'яті з операційної системи в рідкісних випадках вона може бути не виділена (тобто її може і не бути в наявності).

За замовчуванням, якщо оператор new не спрацював, пам'ять не виділилася, то генерується виняток bad_alloc. Якщо цей виняток буде неправильно опрацьовано (а саме так і буде, оскільки ми ще не розглядали винятки і їх обробку), то програма просто припинить своє виконання (станеться збій) з помилкою необробленого винятку.

У багатьох випадках процес генерації винятку оператором new (як і збій програми) небажаний, тому є альтернативна форма оператора new, яка

повертає нульовий вказівник, якщо пам’ять не може бути виділена. Потрібно просто додати **константу std::nothrow** між ключовим словом new і типом даних.

Наприклад:

```
int *value = new (std::nothrow) int; /* вказівник value стане нульовим, якщо динамічне виділення цілочисельної змінної не виконається*/
```

У вищенаведеному прикладі, якщо оператор new не поверне вказівник з динамічно виділеною пам’яттю, то повернеться нульовий вказівник.

Розіменовувати його також не рекомендується, тому що це призведе до несподіваних результатів (швидше за все, до збою в програмі). Тому найкращою практикою є перевірка всіх запитів на виділення пам’яті для забезпечення того, що ці запити будуть виконані успішно та пам’ять виділиться.

Наприклад:

```
int *value = new (std::nothrow) int; /* запит на виділення динамічної пам'яті для цілочисельного значення*/
if (!value) /* обробляємо випадок, коли оператор new повертає null (тобто пам'ять не виділяється)*/
{
   // Обробка цього випадку
   std::cout << "Could not allocate memory";
}
```

Оскільки не виділення пам’яті оператором new відбувається вкрай рідко, то зазвичай програмісти забувають виконувати цю перевірку!

## Нульові вказівники і динамічне виділення пам’яті

**Нульові вказівники** – вказівники зі значенням 0 або nullptr – особливо корисні в процесі динамічного виділення пам’яті [3, 4]. Їх наявність наче повідомляє нам: «Цьому вказівнику не виділено ніякої пам’яті». А це, в свою чергу, можна використати для виконання умовного виділення пам’яті.

Наприклад:

```
// Якщо для ptr досі не виділено пам'яті, то виділяємо її
if (!ptr)
   ptr = new int;
```

Видалення нульового вказівника ні на що не впливає. Таким чином, в наступному немає необхідності:

```
if (ptr)
   delete ptr;
```

Замість цього можна просто написати:

```
delete ptr;
```

Якщо ptr не є нульовим, то динамічно виділена змінна буде видалена. Якщо значенням вказівника є нуль, то нічого не станеться.

## Витік пам’яті

Динамічно виділена пам’ять не має області видимості, тобто вона залишається виділеною до тих пір, поки не буде явно звільнена або поки ваша програма не завершить своє виконання (і операційна система очистить всі буфери пам’яті самостійно). Однак вказівники, які використовуються для зберігання динамічно виділених адрес в пам’яті, дотримуються правил області видимості звичайних змінних. Ця невідповідність може викликати цікаву поведінку.

Наприклад:

```
void doSomething()
{
   int *ptr = new int;
}
```

Тут ми динамічно виділяємо цілочисельну змінну, але ніколи не звільняємо пам’ять через використання оператора delete. Оскільки вказівники слідують всім тим же правилам, що і звичайні змінні, то коли функція завершить своє виконання, ptr вийде з області видимості. Оскільки ptr – це єдина змінна, яка зберігає адресу динамічно виділеної цілочисельної змінної, то коли ptr знищиться, більше не залишиться вказівників на динамічно виділену пам’ять. Це означає, що програма «втратить» адресу динамічно виділеної пам’яті. І в результаті цю динамічно виділену цілочисельну змінну не можна буде видалити. Цю ситуацію називають *витоком пам’яті*.

**Витік пам’яті** відбувається, коли програма втрачає адресу певної динамічно виділеної частини пам’яті (наприклад, змінної або масиву), перш ніж повернути її назад в операційну систему. Коли це відбувається, то програма вже не може видалити цю динамічно виділену пам’ять, оскільки вона більше не знає, де та знаходиться. Операційна система також не може використовувати цю пам’ять, оскільки вважається, що вона як і раніше використовується вашою програмою.

Витоки пам’яті “з’їдають” вільну пам’ять під час виконання програми, зменшуючи кількість доступної пам’яті не тільки для цієї програми, але і для інших програм також. Програми з серйозними проблемами з витоком пам’яті можуть “з’їсти” всю доступну пам’ять, в результаті чого ваш комп’ютер повільніше працюватиме або навіть станеться збій. Тільки після того, як виконання вашої програми завершиться, операційна система зможе очистити і повернути всю пам’ять, яка “втекла”.

Хоча витік пам’яті може виникнути і через те, що вказівник виходить з області видимості, можливі й інші способи, які можуть призвести до витоку пам’яті.

Наприклад, якщо вказівнику, який зберігає адресу динамічно виділеної пам’яті, присвоїти інше значення:

```
int value = 7;
int *ptr = new int;  // виділяємо пам’ять
```

```
ptr = &value;        // стара адреса втрачена, тому відбудеться витік пам'яті
```

Це легко вирішується видаленням вказівника перед операцією переприсвоювання:

```
int value = 7;
int *ptr = new int;  // виділяємо пам'ять
delete ptr;          // повертаємо пам'ять назад в операційну систему
ptr = &value;        // переприсвоюємо вказівнику адресу value
```

Крім того, витік пам'яті також може статися і через подвійне виділення пам'яті:

```
int *ptr = new int;
ptr = new int;       // стара адреса втрачена, тому відбудеться витік пам'яті
```

Адреса, що повертається з другого виділення пам'яті, перезаписує адресу з першого виділення. Відповідно, перше динамічне виділення стає витоком пам'яті!

Точно так же цього можна уникнути видаленням вказівника перед операцією переприсвоювання.

**Динамичні масиви**

Для виділення динамічного масиву і роботи з ним використовуються окремі форми операторів new і delete: **new[]** та **delete[]** [7, 8].

Наприклад:

```
#include <iostream>
int main()
{
   std::cout << "Enter a positive integer: ";
   int length;
   std::cin >> length;
   int *array = new int[length]; /* використовуємо оператор new[] для виділення масиву. Зверніть увагу, змінна length не обов'язково повинна бути константою*/
   std::cout << "I just allocated an array of integers of length " << length << '\n';
   array[0] = 7; // присвоюємо елементу під індексом 0 значення 7
```

```
    delete[] array; /* використовуємо оператор delete[] для звільнення виділеної масиву пам'яті*/
    array = 0; // використовуйте nullptr замість 0 в C++11
    return 0;
}
```

Оскільки ми виділяємо масив, то C++ розуміє, що він повинен використовувати іншу форму оператора new – форму для масиву, а не для змінної. По факту, викликається оператор new[], навіть якщо ми і не вказуємо [] відразу після ключового слова new.

Оскільки пам'ять для динамічних і фіксованих масивів виділяється з різних "резервуарів", то розмір динамічного масиву може бути досить великим. Ви можете запустити вищенаведену програму, але вже виділити масив довжиною 1 000 000 (або, можливо, навіть 100 000 000) елементів без проблем.

Видалення динамічного масиву. При видаленні динамічних масивів також використовується форма оператора delete для масивів – delete[]. Таким чином, ми повідомляємо процесору, що йому потрібно очистити пам'ять від декількох змінних замість однієї. Поширена помилка, яку роблять початківці при роботі з динамічним виділенням пам'яті, є використання delete замість delete[] для видалення динамічних масивів. Використання форми оператора delete для змінних при видаленні масиву призведе до таких несподіваних результатів, як пошкодження даних,витік пам'яті, збій або інші проблеми.

**Ініціалізація динамічного масиву**

Якщо ви хочете ініціалізувати динамічний масив значенням 0, то все досить просто:

```
int *array = new int[length]();
```

До C++11 не було простого способу ініціалізувати динамічний масив ненульовими значеннями (список ініціалізаторів працював тільки з фіксованими масивами). А це означає, що потрібно перебрати кожен елемент масиву і явно присвоїти йому значення.

Наприклад:

```
int *array = new int[5];
array[0] = 9;
array[1] = 7;
array[2] = 5;
array[3] = 3;
array[4] = 1;
```

Починаючи з C++11, з'явилася можливість ініціалізації динамічних масивів через списки ініціалізаторів.

Наприклад:

```
int fixedArray[5] = { 9, 7, 5, 3, 1 }; // ініціалізуємо фіксований масив
int *array = new int[5] { 9, 7, 5, 3, 1 }; // ініціалізуємо динамічний масив
```

Зверніть увагу, в синтаксисі динамічного масиву між довжиною масиву і списком ініціалізаторів оператора присвоювання (=) немає.

У C++11 фіксовані масиви також можуть бути ініціалізовані з використанням **uniform-ініціалізації**.

Наприклад:

```
int fixedArray[5] { 9, 7, 5, 3, 1 }; // ініціалізуємо фіксований масив в C++11
char fixedArray[14] { "Hello, world!" }; // ініціалізуємо фіксований масив в C++11
```

В C++11 не можна ініціалізувати динамічний масив символів **рядком C-style**.

Наприклад:

```
char *array = new char[14] { "Hello, world!" }; // не працює в C++11
```

Можна динамічно виділити **std::string** (або виділити динамічний масив символів, а потім за допомогою функції strcpy_s() скопіювати вміст потрібного рядка в цей масив).

Динамічні масиви повинні бути оголошені з явним вказуванням їх довжини.

Наприклад:

```
int fixedArray[] {1, 2, 3}; // ок: неявне вказування довжини фіксованого масиву
```

```
int *dynamicArray1 = new int[] {1, 2, 3}; /* не ок: неявне вказування довжини динамічного масиву*/
int *dynamicArray2 = new int[3] {1, 2, 3}; /* ок: явне вказування довжини динамічного масиву*/
```

### Зміна довжини масиву

Динамічне виділення масивів дозволяє задавати довжину масивів під час операції їх виділення. Однак мова C++ не надає вбудований спосіб зміни довжини масиву, який вже був виділений. Але і це обмеження можна обійти, динамічно виділивши новий масив, скопіювавши всі елементи зі старого масиву, а потім видаливши старий масив. Однак цей спосіб вразливий до помилок.

В мові C++ є масиви, розмір яких можна змінювати. Такі масиви називають **векторами** (std::vector).

### Контрольні запитання

1. У чому полягає фундаментальна різниця між статичним виділенням пам’яті (в стеку) та динамічним (у купі/heap)? Ким і коли контролюється життєвий цикл змінних у кожному з цих випадків?

2. У яких практичних ситуаціях програміст змушений використовувати саме динамічне виділення пам’яті? Наведіть приклад задачі, яку неможливо вирішити лише статичними змінними.

3. Що таке «витік пам’яті» (memory leak)? Які наслідки він може мати для програми, що працює безперервно протягом тривалого часу (наприклад, сервера)?

4. Який саме тип даних повертає оператор new при успішному виділенні пам’яті? Напишіть приклад коду для створення однієї змінної типу int у динамічній пам’яті.

5. Що відбудеться, якщо оператору new не вдасться знайти достатньо вільної пам’яті (наприклад, запитано надто великий обсяг)? Яким чином програма реагує на цю ситуацію за замовчуванням?

6. Яка головна функція оператора delete? Чому в C++ немає автоматичного «збирача сміття» (garbage collector), як у Java чи C#, і звільнення пам’яті лягає на плечі розробника?

7. Що таке «висячий вказівник» (dangling pointer)? Чому після виклику delete ptr; вважається правилом хорошого тону одразу ж написати ptr = nullptr;? Що станеться, якщо двічі застосувати delete до одного й того ж вказівника?

8. Як правильно виділити пам'ять під динамічний масив із N елементів типу double, де N вводиться користувачем з клавіатури? Напишіть відповідний рядок коду.

9. У чому полягає критична різниця між операторами delete та delete[]? Що відбудеться (яка помилка або невизначена поведінка виникне), якщо спробувати звільнити пам’ять з-під динамічного масиву за допомогою звичайного delete?

10. Чи існує в C++ вбудований оператор для прямої зміни розміру вже існуючого динамічного масиву (щоб просто додати туди ще 5 елементів)? Опишіть покроковий алгоритм того, як «розширити» динамічний масив зі збереженням старих даних.

11. Як дізнатися розмір динамічного масиву (кількість його елементів), маючи лише вказівник на його перший елемент? Чи спрацює тут трюк із sizeof, як це було з фіксованими масивами?

## Лекція №8. Рядки та структури

### Тип даних string

Щоб мати можливість використовувати рядки в мові C++, спочатку необхідно підключити **заголовок** string [5]. Як тільки це буде зроблено, можна визначити змінні типу string.

Наприклад:

```
#include <string>
// ...
std::string name;
// ...
```

Змінні типу string можна ініціалізовувати або присвоювати їм значення.

Наприклад:

```
std::string name("Sasha"); // ініціалізуємо змінну name рядковим літералом "Sasha"
name = "Masha"; // присвоюємо змінній name рядковий літерал "Masha"
```

Рядки також можуть містити числа.

Наприклад:

```
std::string myID("34"); // "34" тут - це не ціле число 34, а текст
```

Числа, які присвоюються, тип string обробляє як текст, а не як числа. А це означає, що ними не можна маніпулювати як звичайними числами (наприклад, ви не зможете виконати з ними арифметичні операції). Мова C++ автоматично НЕ конвертує їх в значення цілочисельних типів данів або типів з плаваючою крапкою.

### Введення/виведення рядків

Рядки можна виводити за допомогою **std::cout** [6].

Наприклад:

```
#include <iostream>
#include <string>
int main()
{
   std::string name("Sasha");
```

```
    std::cout << "My name is " << name;
    return 0;
}
```

Результат виконання програми:

```
My name is Sasha
```

Розглянемо наступний приклад:

```
#include <iostream>
#include <string>
int main()
{
    std::cout << "Enter your full name: ";
    std::string myName;
    std::cin >> myName; /* це працюватиме не так, як очікується, оскільки вилучення даних з потоку std::cin зупиниться на першому пробілі*/
    std::cout << "Enter your age: ";
    std::string myAge;
    std::cin >> myAge;
    std::cout << "Your name is " << myName << " and your age is " << myAge;
}
```

Результат виконання програми:

```
Enter your full name: Sasha Mak
Enter your age: Your name is Sasha and your age is Mak
```

**Оператор виведення (>>)** повертає символи з вхідного потоку даних тільки до першого пробілу. Всі інші символи залишаються всередині std::cin, очікуючи наступного вилучення.

Тому, коли ми використали оператор >> для вилучення даних в змінну myName, тільки Sasha вдалося витягнути, Mak залишився всередині std::cin, очікуючи наступного вилучення. Коли ми використали оператор >> знову,

щоб витягнути дані в змінну myAge, ми отримали Mak замість 25. Якби ми виконали третє вилучення, то отримали б 25.

### Використання std::getline()

Щоб витягнути цілий рядок з вхідного потоку даних (разом з пробілами), використовується **функція std::getline()** [3]. Вона приймає два параметри: перший – std::cin, другий – змінна типу string.

Перепишемо вищенаведену програму з використанням std::getline():

```
#include <iostream>
#include <string>
int main()
{
   std::cout << "Enter your full name: ";
   std::string myName;
   std::getline(std::cin, myName); // повністю вилучаємо рядок в змінну myName
   std::cout << "Enter your age: ";
   std::string myAge;
   std::getline(std::cin, myAge); // повністю вилучаємо рядок в змінну myAge
   std::cout << "Your name is " << myName << " and your age is " << myAge;
}
```

Тепер програма працює правильно:

```
Enter your full name: Sasha Mak
Enter your age: 25
Your name is Sasha Mak and your age is 25
```

### Використання std::getline() з std::cin

Вилучення даних з std::cin за допомогою std::getline() іноді може призвести до несподіваних результатів.

Наприклад, розглянемо наступну програму:

```
#include <iostream>
#include <string>
int main()
```

```
{
   std::cout << "Pick 1 or 2: ";
   int choice;
   std::cin >> choice;
   std::cout << "Now enter your name: ";
   std::string myName;
   std::getline(std::cin, myName);
   std::cout << "Hello, " << myName << ", you picked " << choice << '\n';
   return 0;
}
```

Коли ви запустите цю програму, і вона попросить вас ввести ваше ім’я, вона не чекатиме вашого вводу, а відразу виведе результат (просто пробіл замість вашого імені)!

Результат пробного запуску програми:

```
Pick 1 or 2: 2
Now enter your name: Hello, , you picked 2
```

Коли ви вводите числове значення, потік cin захоплює разом з вашим числом і символ нового рядка. Тому, коли ми ввели 2, cin фактично отримав 2\n. Потім він витягнув значення 2 в змінну, залишаючи \n (символ нового рядка) у вхідному потоці. Потім, коли std::getline() отримує дані для myName, він бачить в потоці \n і думає, що ми, мабуть, ввели просто порожній рядок! А це, безумовно, не те, що ми хочемо.

Хорошою практикою вважається видаляти з вхідного потоку даних символ нового рядка. Це можна зробити наступним чином:

```
std::cin.ignore(32767, '\n'); /* ігноруємо символи нового рядка "\n" у вхідному потоці довжиною 32767 символів*/
```

Якщо ми вставимо цей рядок безпосередньо після отримання вхідних даних, то символ нового рядка буде видалено з вхідного потоку, і програма працюватиме належним чином:

```
#include <iostream>
```

```
#include <string>
int main()
{
	std::cout << "Pick 1 or 2: ";
	int choice;
	std::cin >> choice;
	std::cin.ignore(32767, '\n'); /* видаляємо символ нового рядка з вхідного потоку даних*/
	std::cout << "Now enter your name: ";
	std::string myName;
	std::getline(std::cin, myName);
	std::cout << "Hello, " << myName << ", you picked " << choice << '\n';
	return 0;
}
```

Зверніть увагу! У випадку введення числових значень не забувайте видаляти символ нового рядка з вхідного потоку даних за допомогою std::cin.ignore().

**Додавання рядків**

Можна використовувати оператор + для об’єднання двох рядків або оператор += для додавання двох рядків [4, 5].

У наступній програмі ми протестуємо ці два оператори, а також покажемо, що відбудеться, якщо ви спробуєте використати оператор + для додавання двох числових рядків:

```
#include <iostream>
#include <string>
int main()
{
   std::string x("44");
   std::string y("12");
```

```
    std::cout << x + y << "\n"; // об’єднуємо рядки x і y (а не додаємо числа)
    x += " cats";
    std::cout << x;
    return 0;
}
```

Результат виконання програми:

```
4412
44 cats
```

Оператор + об’єднав два числових рядки в один (44 + 12 = 4412). Він не додавав ці рядки як числа.

### Довжина рядків

Щоб дізнатися довжину рядка, ми можемо зробити наступне:

```
#include <iostream>
#include <string>
int main()
{
    std::string myName("Sasha");
    std::cout << myName << " has " << myName.length() << " characters\n";
    return 0;
}
```

Результат виконання програми:

```
Sasha has 5 characters
```

Зверніть увагу, замість запиту довжини рядка як length(myName), ми пишемо myName.length().

Функція запиту довжини рядка не є звичайною функцією, як ті, що ми використовували раніше. Це особливий тип функції класу std::string, який називають **методом**.

### Рядки C-style

**Рядок C-style** – це простий масив символів, який використовує нуль-термінатор. **Нуль-термінатор** – це спеціальний символ (**ASCII-код** якого

дорівнює 0), який використовується для позначення кінця рядку.

Для визначення рядку C-style необхідно просто оголосити масив типу char і ініціалізувати його **літералом**.

Наприклад:

```
char mystring[] = "string";
```

Хоча string має лише 6 літер, мова C++ автоматично додає **нуль-термінатор** в кінець рядку (нам не потрібно додавати його вручну). Відповідно, довжина масиву mystring дорівнює 7!

В якості прикладу розглянемо наступну програму, яка виводить довжину рядка, а потім ASCII-коди всіх символів літералу string:

```
#include <iostream>
int main()
{
   char mystring[] = "string";
   std::cout << mystring << " has " << sizeof(mystring) << " characters.\n";
   for (int index = 0; index < sizeof(mystring); ++index)
      std::cout << static_cast<int>(mystring[index]) << " ";
   return 0;
}
```

Результат виконання програми:

```
string has 7 characters.
115 116 114 105 110 103 0
```

Нуль в кінці є ASCII-кодом нуль-термінатора, який був доданий в кінець рядка.

При такому оголошенні рядків рекомендується використовувати квадратні дужки [], щоб дозволити компілятору визначати довжину масиву самостійно. Таким чином, якщо ви зміните рядок пізніше, вам не доведеться вручну змінювати значення довжини масиву.

Рядки C-style слідують всім тим же правилам, що і масиви. Це означає, що ви можете ініціалізувати рядок при створенні, але після цього не зможете присвоювати йому значення за допомогою оператора присвоювання.

Наприклад:

```
char mystring[] = "string"; // ок
mystring = "cat"; // не ок!
```

Це те ж саме, якби ми зробили наступне:

```
int array[] = { 4, 6, 8, 2 }; // ок
array = 7; // що це означає?
```

Оскільки рядки C-style є масивами, то ви можете використовувати оператор [] для зміни окремих символів в рядку.

Наприклад:

```
#include <iostream>
int main()
{
   char mystring[] = "string";
   mystring[1] = 'p';
   std::cout << mystring;
   return 0;
}
```

Результат виконання програми:

```
spring
```

При виведенні рядка C-style, std::cout виводить символи до тих пір, поки не зустріне нуль-термінатор. Якби ви випадково перезаписали нуль-термінатор в кінці рядка (наприклад, присвоївши що-небудь для mystring[6]), то ви б не тільки отримали всі символи рядка, але std::cout також вивів би все, що знаходиться в сусідніх комірках пам’яті до тих пір, поки він не дійшов би до 0!

Це нормально, якщо довжина масиву більше рядка, якого він (масив) зберігає.

Наприклад:

```
#include <iostream>
int main()
{
   char name[15] = "Max"; /* використовується тільки 4 символи (3 букви + нуль-термінатор)*/
   std::cout << "My name is: " << name << '\n';
   return 0;
}
```

В цьому випадку рядок Max виведеться на екран, а std::cout зупиниться на нуль-термінаторі. Решта символів в масиві будуть проігноровані.

**Рядки C-style і std::cin**

Є багато випадків, коли ми не знаємо заздалегідь, наскільки довгим буде наш рядок. Наприклад, розглянемо проблему написання програми, де ми просимо користувача ввести своє ім’я. У такому випадку ми можемо оголосити масив розміром більше, ніж нам потрібно.

Наприклад:

```
#include <iostream>
int main()
{
   char name[255]; // оголошуємо достатньо великий масив (для зберігання 255 символів)
   std::cout << "Enter your name: ";
   std::cin >> name;
   std::cout << "You entered: " << name << '\n';
   return 0;
}
```

У вищенаведеній програмі ми оголосили масив з 255 символів, припускаючи, що користувач не введе ім’я більше 255 символів. Хоча це і поширена практика, але вона не дуже ефективна, оскільки користувачу нічого

не перешкоджає ввести ім’я, яке містить більше 255 символів (випадково або навмисно).

Набагато краще зробити наступне:

```
#include <iostream>
int main()
{
   char name[255]; /* оголошуємо достатньо великий масив (для зберігання 255 символів)*/
   std::cout << "Enter your name: ";
   std::cin.getline(name, 255);
   std::cout << "You entered: " << name << '\n';
   return 0;
}
```

Виклик cin.getline() прийматиме до 254 символів в масив name (залишаючи місце для нуль-термінатора). Будь-які зайві символи будуть проігноровані. Таким чином, гарантовано масив не буде переповнено.

## Керування рядками C-style

Мова C++ надає безліч функцій для керування рядками C-style, які підключають за допомогою заголовного файлу cstring**.**

**Функція strcpy_s()** копіює вміст одного рядка в інший. Найчастіше це використовують для присвоювання значень рядку.

Наприклад:

```
#include <iostream>
#include <cstring>
int main()
{
   char text[] = "Print this!";
   char dest[50];
   strcpy_s(dest, text);
   std::cout << dest; // виводимо "Print this!"
```

```
    return 0;
}
```

Однак, використання функції strcpy_s() може легко викликати **переповнення** масиву.

У наступній програмі довжина масиву dest менше ніж довжина рядка, який ми копіюємо, тому в результаті ми отримаємо переповнення масиву:

```
#include <iostream>
    #include <cstring>
    int main()
    {
        char text[] = "Print this!";
        char dest[5]; // зверніть увагу, що довжина масиву dest всього 5 символів
        strcpy_s(dest, text); // переповнення!
        std::cout << dest;
        return 0;
    }
```

**Функція strlen()** повертає довжину рядка C-style (не враховуючи нуль-термінатор).

Наприклад:

```
#include <iostream>
#include <cstring>
int main()
{
    char name[15] = "Max"; /* використовується тільки 4 символи (3 букви + нуль-термінатор)*/
    std::cout << "My name is " << name << '\n';
    std::cout << name << " has " << strlen(name) << " letters.\n";
    std::cout << name << " has " << sizeof(name) << " characters in the array.\n";
    return 0;
}
```

Результат виконання програми:

```
My name is Max
Max has 3 letters.
Max has 15 characters in the array.
```

Зверніть увагу на різницю між функцією strlen() і оператором sizeof. Функція strlen() виводить кількість символів ДО нуль-термінатора, тоді як **оператор sizeof** повертає розмір цілого масиву, незалежно від того, що в ньому знаходиться.

**Ось ще декілька корисних функцій керування рядками C-style:**

- **функція strcat()** – додає один рядок до іншого (небезпечно);

- **функція strncat()** – додає один рядок до іншого (з перевіркою розміру місця призначення);

- **функція strcmp()** – порівнює два рядки (повертає 0, якщо вони рівні);

- **функція strncmp()** – порівнює два рядки до певної кількості символів (повертає 0, якщо до вказаного символу не було відмінностей).

Наприклад:

```
#include <iostream>
#include <cstring>
int main()
{
   // Просимо користувача ввести рядок
   char buffer[255];
   std::cout << "Enter a string: ";
   std::cin.getline(buffer, 255);
   int spacesFound = 0;
   // Перебираємо кожний символ, який ввів користувач
   for (int index = 0; index < strlen(buffer); ++index)
   {
      // Підраховуємо кількість пробілів
      if (buffer[index] == ' ')
```

```
        spacesFound++;
    }
    std::cout << "You typed " << spacesFound << " spaces!\n";
    return 0;
}
```

Мова C++ підтримує ще один спосіб створення символьних констант рядків C-style – **через вказівники**.

Наприклад:

```
#include <iostream>
int main()
{
    const char *myName = "John";
    std::cout << myName;
    return 0;
}
```

Хоча обидві ці програми працюють та показують однакові результати, виділення пам’яті в них виконується по-різному.

У першому випадку в програмі виділяється пам’ять для **фіксованого масиву** довжиною 5 і ініціалізується ця пам’ять рядком John\0. Оскільки пам’ять була спеціально виділена для масиву, то ми можемо змінювати її вміст. Сам масив розглядається як звичайна локальна змінна, тому, коли він виходить з області видимості, пам’ять, яку він використовував, звільняється для інших об’єктів.

Що відбувається у випадку з **символьною константою**? Компілятор поміщає рядок John\0 в пам’ять типу read-only (тільки читання), а потім створює вказівник, який вказує на цей рядок. Декілька рядкових літералів з одним і тим же вмістом можуть вказувати на одну і ту ж адресу. Оскільки ця пам’ять доступна тільки для читання, а також тому, що внесення змін до рядкового літералу може вплинути на подальше його використання, найкраще перестрахуватися, оголосивши рядок константним (типу const). Також,

оскільки рядки, оголошені таким чином, існують протягом усього “життя” програми (вони мають статичну тривалість життя, а не автоматичну, як більшість інших “локально визначених” літералів), нам не потрібно турбуватися про проблеми, пов’язані з областю видимості.

Наприклад:

```
const char* getName()
{
   return "John";
}
```

У фрагменті, наведеному вище, функція getName() повертає вказівник на рядок C-style John. Все добре, так як John не виходить з області видимості, коли getName() завершує своє виконання, тому caller все одно має доступ до рядка.

**std::cout та вказівники типу char**

Розглянемо наступний приклад:

```
#include <iostream>
int main()
{
   int nArray[5] = { 9, 7, 5, 3, 1 };
   char cArray[] = "Hello!";
   const char *name = "John";
   std::cout << nArray << '\n'; // nArray конвертується у вказівник типу int
   std::cout << cArray << '\n'; // cArray конвертується у вказівник типу char
   std::cout << name << '\n'; // name уже і так є вказівником типу char
   return 0;
}
```

Результат виконання програми на моєму комп’ютері:

```
0046FAE8
Hello!
John
```

При передачі вказівника НЕ типу char, в результаті виводиться просто вміст цього вказівника (адреса в пам’яті). Однак, якщо ви передасте об’єкт типу char* або const char*, то std::cout припустить, що ви маєте намір вивести рядок на екран. Отже, замість виведення значення вказівника – виведеться рядок, на який він вказує.

Наприклад:

```
#include <iostream>
int main()
{
   char a = 'R';
   std::cout << &a;
   return 0;
}
```

Тут ми маємо намір вивести адресу змінної a. Проте, &a має тип char*, тому std::cout виведе це як рядок.

Результат виконання програми:

R╠╠╠╠╣ 4;¿■A

std::cout припустив, що &a (типу char*) є рядком. Тому спочатку вивелося R, а потім операція виведення продовжилася. Наступним в пам’яті було сміття. Зрештою, std::cout зіткнувся з коміркою в пам’яті, яка має значення 0, яке він інтерпретував як нуль-термінатор, і, відповідно, припинив операцію виводу. (Те, що ви побачите в результаті, залежить від того, що знаходиться в пам’яті після змінної a).

## Структури

Мова C++ дозволяє програмістам створювати свої власні **користувацькі типи даних** – типи, які групують декілька окремих змінних разом. Одним з найпростіших користувацьких типів даних є **структура** [3 – 6]. Структура дозволяє згрупувати змінні різних типів даних в єдине ціле.

## Оголошення і визначення структур

Оскільки структури визначають програмісти, то спочатку ми повинні повідомити компілятору, як структура взагалі виглядатиме. Для цього використовують ключове слово **struct**.

Наприклад:

```
struct Employee
{
    short id;
    int age;
    double salary;
};
```

Ми визначили структуру з ім’ям Employee. Вона містить 3 змінні:

id типу short;

age типу int;

salary типу double.

Ці змінні, які є частиною структури, називають **членами структури** (або **“полями структури”**). Employee – це просте оголошення структури. Хоч ми і вказали компілятору, що вона має змінні-члени, пам’ять під неї зараз не виділяється. Імена структур прийнято писати з великої літери, щоб відрізняти їх від імен змінних.

Щоб використовувати структуру Employee, нам потрібно просто оголосити змінну типу Employee.

Наприклад:

```
Employee john; /* ім'я структури Employee починається з великої літери, а змінна john – з маленької*/
```

Тут ми визначили змінну типу Employee з ім’ям john. Як і у випадку зі звичайними змінними, визначення змінної структури призведе до виділення для неї пам’яті.

Оголосити можна і декілька змінних однієї структури.

Наприклад:

```
Employee john; // створюємо окрему структуру Employee для John
Employee james; // створюємо окрему структуру Employee для James
```

**Доступ до членів структур**

Коли ми оголошуємо змінну структури, наприклад, Employee john, то john посилається на всю структуру. Для того, щоб отримати доступ до окремих її членів, використовується оператор вибору члена (.).

Наприклад, в коді, наведеному нижче, ми використовуємо оператор вибору членів для ініціалізації кожного члена структури:

```
Employee john;        // створюємо окрему структуру Employee для John
john.id = 8;          // присвоюємо значення члену id структури john
john.age = 27;        // присвоюємо значення члену age структури john
john.salary = 32.17;  // присвоюємо значення члену salary структури john
Employee james;       // створюємо окрему структуру Employee для James
james.id = 9;         // присвоюємо значення члену id структури james
james.age = 30;       // присвоюємо значення члену age структури james
james.salary = 28.35; // присвоюємо значення члену salary структури james
```

Як і у випадку зі звичайними змінними, змінні-члени структури не ініціалізуються автоматично і зазвичай містять сміття. Ініціалізувати їх потрібно вручну.

У вищенаведеному прикладі легко визначити, яка змінна відноситься до структури John, а яка – до структури James. Це забезпечує набагато більш високий рівень організації, ніж у випадку зі звичайними окремими змінними.

Змінні-члени структури працюють так само, як і прості змінні, тому з ними можна виконувати звичайні **арифметичні операції** та **операції порівняння**.

Наприклад:

```
int totalAge = john.age + james.age;
if (john.salary > james.salary)
   cout << "John makes more than James\n";
```

```
else if (john.salary < james.salary)
   cout << "John makes less than James\n";
else
   cout << "John and James make the same amount\n";
// James отримав підвищення на роботі
james.salary += 3.75;
// Сьогодні день народження у John
++john.age; // використовуємо пре-інкремент для збільшення віку John на 1 рік
```

**Ініціалізація структур**

Ініціалізація структур шляхом присвоювання значень кожному члену в індивідуальному порядку – заняття досить нудне (особливо, якщо цих членів багато), тому в мові C++ є більш швидкий спосіб ініціалізації структур – за допомогою **списку ініціалізаторів**, який дозволяє ініціалізувати деякі або всі члени структури під час оголошення змінної типу struct.

Наприклад:

```
struct Employee
{
   short id;
   int age;
   double salary;
};
Employee john = { 5, 27, 45000.0 }; /* john.id=5, john.age=27, john.salary=45000.0
Employee james = { 6, 29 }; /* james.id = 6, james.age = 29, james.salary = 0.0 (ініціалізація за замовчуванням)*/
```

В C++11 ініціалізації структур також можна використовувати **uniform-ініціалізацію**.

Наприклад:

```
Employee john { 5, 27, 45000.0 }; // john.id = 5, john.age = 27, john.salary = 45000.0
Employee james { 6, 29 }; /* james.id = 6, james.age = 29, james.salary = 0.0 (ініціалізація за замовчуванням)*/
```

Якщо в списку ініціалізаторів не буде одного або декількох елементів, то цим елементам присвоїться значення за замовчуванням (зазвичай, 0). У вищенаведеному прикладі члену james.salary присвоюється значення за замовчуванням 0.0, тому що самі ми не надали ніякого значення під час ініціалізації.

**Присвоювання значень членам структур**

До C++11, якби ми захотіли присвоювати значення членам структури, то нам би довелося це робити вручну для кожного члену окремо.

Наприклад:

```
struct Employee
{
   short id;
   int age;
   double salary;
};
Employee john;
john.id = 5;
john.age = 27;
john.salary = 45000.0;
```

У C++11 є можливість присвоювати значення членам структур, використовуючи список ініціалізаторів.

Наприклад:

```
struct Employee
{
   short id;
   int age;
   double salary;
};
Employee john;
john = { 5, 27, 45000.0 }; // починаючи з C++11
```

## Структури і функції

Великою перевагою використання структур замість окремих змінних є можливість передати всю структуру в функцію, яка повинна працювати з її членами.

Наприклад:

```
#include <iostream>
struct Employee
{
   short id;
   int age;
   double salary;
};
void printInformation(Employee employee)
{
   std::cout << "ID: " << employee.id << "\n";
   std::cout << "Age: " << employee.age << "\n";
   std::cout << "Salary: " << employee.salary << "\n";
}
int main()
{
   Employee john = { 21, 27, 28.45 };
   Employee james = { 22, 29, 19.29 };
   // Виводимо інформацію про John
   printInformation(john);
   std::cout << "\n";
   // Виводимо інформацію про James
   printInformation(james);
   return 0;
}
```

У вищенаведеному прикладі ми передали структуру Employee в функцію printInformation(). Це дозволило нам не передавати кожну змінну окремо. Більше того, якщо ми коли-небудь захочемо додати нових членів в структуру Employee, то нам не доведеться змінювати оголошення або виклик функції.

Результат виконання програми:

```
ID: 21
Age: 27
Salary: 28.45
ID: 22
Age: 29
Salary: 19.29
```

Функція також може повертати структуру (це один з тих небагатьох випадків, коли функція може повертати декілька змінних).

Наприклад:

```
#include <iostream>
struct Point3d
{
   double x;
   double y;
   double z;
};
Point3d getZeroPoint()
{
   Point3d temp = { 0.0, 0.0, 0.0 };
   return temp;
}
int main()
{
   Point3d zero = getZeroPoint();
```

```
    if (zero.x == 0.0 && zero.y == 0.0 && zero.z == 0.0)
        std::cout << "The point is zero\n";
    else
        std::cout << "The point is not zero\n";
    return 0;
}
```

Результат виконання програми:

```
The point is zero
```

**Вкладені структури**

Одні структури можуть містити інші структури, тобто можуть бути вкладені одна в одну [2].

Наприклад:

```
struct Employee
{
    short id;
    int age;
    double salary;
};
struct Company
{
    Employee CEO; // Employee - це структура всередині структури Company
    int numberOfEmployees;
};
Company myCompany;
```

В цьому випадку, якщо б ми захотіли дізнатися, яка зарплата в CEO (виконавчого директора), то нам довелося б використати оператор вибору членів двічі:

```
myCompany.CEO.salary
```

Спочатку ми вибираємо поле CEO з структури myCompany, а потім вибираємо поле salary з структури Employee.

Можна також використовувати вкладені списки ініціалізаторів з вкладеними структурами.

Наприклад:

```
struct Employee
{
   short id;
   int age;
   float salary;
};
struct Company
{
   Employee CEO; // Employee є структурою всередині структури Company
   int numberOfEmployees;
};
Company myCompany = {{ 3, 35, 55000.0f }, 7 };
```

**Розмір структур**

Як правило, **розмір структури** – це сума розмірів всіх її членів, але не завжди.

Наприклад, розглянемо структуру Employee. На більшості платформ тип short займає 2 байти, тип int – 4 байти, а тип double – 8 байт. Отже, очікується, що Employee займатиме 2 + 4 + 8 = 14 байт. Щоб дізнатися точний розмір Employee, ми можемо скористатися **оператором sizeof**:

```
#include <iostream>
struct Employee
{
   short id;
   int age;
   double salary;
};
int main()
```

```
{
   std::cout << "The size of Employee is " << sizeof(Employee) << "\n";
   return 0;
}
```

Результат виконання програми:

```
The size of Employee is 16
```

Виявляється, ми можемо сказати тільки, що розмір структури буде, принаймні, не менше суми розмірів всіх її членів. Але він може бути і більше. З міркувань продуктивності компілятор іноді може додавати “пробіли” в структури.

У структурі Employee компілятор неявно додав 2 байти після члену id, збільшуючи розмір структури до 16 байтів (замість 14).

### Доступ до структур з декількох файлів

Оскільки оголошення структури не спричинює виділення пам’яті, то використовувати **попереднє оголошення** для неї ви не зможете. Але є обхідний шлях: якщо ви хочете використовувати оголошення структури в декількох файлах (щоб мати можливість створювати змінні цієї структури в декількох файлах), то **помістіть оголошення структури у заголовний файл** та підключайте цей файл всюди, де необхідно використовувати структуру.

Змінні типу struct слідують тим же правилам, що і звичайні змінні. Отже, якщо ви хочете зробити змінну структури доступною в декількох файлах, то ви можете використати **ключове слово extern**.

### Контрольні запитання

1. Що таке С-рядок (C-string) з точки зору організації в пам’яті? Який спеціальний невидимий символ обов’язково має стояти в кінці такого рядка і чому він критично важливий?

2. У чому полягають головні переваги використання класу std::string зі стандартної бібліотеки C++ порівняно з класичними масивами символів (С-рядками)?

3. Як обчислити довжину рядка (кількість символів) для С-рядка і для об’єкта std::string? Назвіть відповідну С-функцію та метод С++ класу. Чи враховують вони нуль-термінатор у загальну довжину?

4. Як відбувається об’єднання (конкатенація) двох рядків? Поясніть, чому для std::string можна просто використати оператор +, а для С-рядків потрібна спеціальна функція (наприклад, strcat).

5. Як порівняти два рядки на ідентичність? Чому оператор == працює коректно для порівняння двох std::string, але при застосуванні до двох С-рядків він порівнює лише їхні адреси в пам’яті, а не текст?

6. Як отримати сумісність: яким методом можна витягнути «сирий» С-рядок (вказівник const char*) із об'єкта std::string? Для чого це найчастіше буває потрібно на практиці?

7. Що таке структура (struct) у мові С++? Чим структура принципово відрізняється від масиву?

8. За допомогою якого синтаксису (яких операторів) здійснюється звернення до окремих полів (членів) структури?

9. Як правильно ініціалізувати об’єкт структури одразу під час його створення? Наведіть приклад використання списку.

10. Чи завжди розмір структури в пам’яті (результат роботи оператора sizeof) дорівнює простій математичній сумі розмірів усіх її полів? Що таке «вирівнювання пам’яті» (memory alignment / padding) і для чого компілятор додає порожні байти між полями?

## Лекція №9. Основи об’єктно-орієнтованого програмування

До сих пір ми писали програми, в яких не використовувались які б то С++-засоби об’єктно-орієнтованого програмування. Таким чином, попередні ваші програми відображали структурне, а не об’єктно-орієнтоване програмування. Для написання ж об’єктно-орієнтованих програм необхідне використання класів.

### Нові концепції програмування. Історія

Об’єктно-орієнтоване програмування – це технологія, що виникла як реакція на кризу програмного забезпечення, коли методи структурного програмування вже не дозволяли справлятися із зростаючою складністю промислового програмного продукту [1 – 4].

40 й більше років тому програмісти реалізовували свої проекти лише шляхом безпосереднього написання коду. Проте, із зростанням розміру та складності проектів ставало все очевиднішим, що такий підхід є невдалий. Проблема полягала у непропорційному зростанні складності процесу створення самих програм.

В результаті пошуку нових більш ефективних технологій програмування було розроблено три нові концепції програмування:

- об’єктно-орієнтоване програмування (ООП);
- уніфікована мова моделювання (UML);
- спеціалізовані засоби розробки програмного забезпечення.

В першу чергу, розвиток об’єктно-орієнтованого методу програмування обумовлений обмеженістю інших методів програмування, розроблених раніше. Для того, щоб зрозуміти й оцінити значення ООП, необхідно розібратися, в чому полягає ця обмеженість та яким чином вона проявляється в традиційних мовах програмування.

### Процедурні мови. Недоліки структурного програмування

C, Pascal, FORTRAN та інші схожі з ними мови програмування відносяться до категорії процедурних мов. Кожен оператор такої мови є вказівкою комп’ютеру виконати певну дію. Програміст створює перелік інструкцій, а

комп’ютер виконує дії, що відповідають цим інструкціям.

Як би ефективно не використовувався структурний підхід, він не дозволяє в достатній мірі спростити великі складні проекти. Коли розмір програми великий, список команд стає доволі громіздким і стає неможливим утримувати в пам’яті всі деталі (небагато програмістів здатні утримувати в голові більш ніж 500 рядків програмного коду).

Крім того, у процедурній програмі існує ще й проблема неконтрольованого доступу до даних. В чому суть цієї проблеми? В процедурній програмі, що написана, наприклад, мовою С, існує два типи даних. *Локальні* дані знаходяться всередині функції й призначені для використання виключно цією функцією. Якщо існує необхідність спільного використання одних і тих самих даних кількома функціями, то дані повинні бути оголошені як *глобальні* (рис. 9.1). А це, як правило, стосується тих даних програми, які є найбільш важливими. Великі програми зазвичай містять велику кількість функцій та глобальних змінних. Проблема процедурного підходу полягає в тому, що число можливих зв’язків між глобальними змінними та функціями може бути дуже великим.

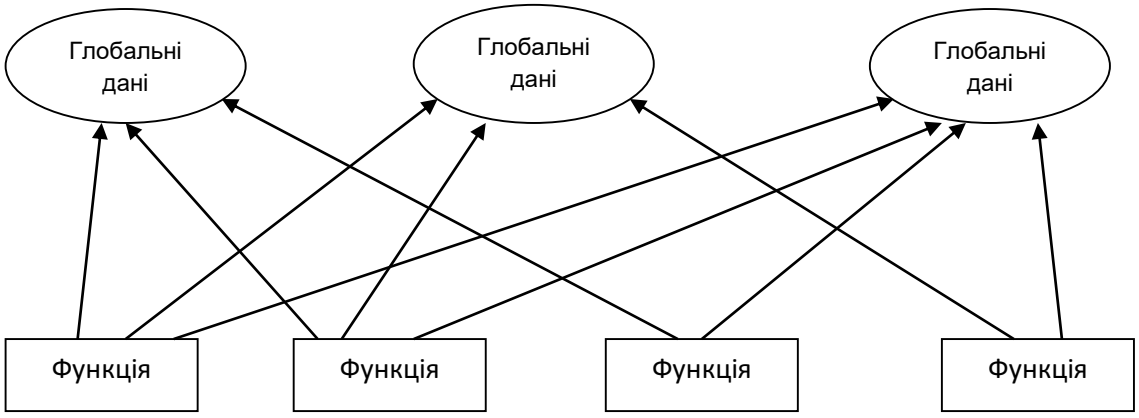



Рис. 9.1. Глобальні дані програм

Велика кількість зв’язків між функціями та даними, в свою чергу, також породжує кілька проблем. По-перше, ускладнюється структура програми. По-друге, в програму стає важко вносити зміни. Зміна структури глобальних даних може потребувати переписування всіх функцій, працюючих з цими даними.

Коли зміни вносяться в глобальні дані великих програм, буває складно швидко визначитися, які саме функції необхідно скорегувати. Навіть у тому випадку, коли це вдається зробити, через велику кількість зв'язків між функціями та даними виправлені функції починають некоректно працювати з іншими глобальними даними.

І, нарешті, третя, більш важлива, проблема процедурного програмування полягає в тому, що відокремлення даних від функцій виявилось малопридатним для відображення картини реального світу.

Подивіться на світ навколо нас. В реальному світі нам доводиться мати справу з фізичними об'єктами, такими, наприклад, як люди, машини чи комп'ютери. Кожен об'єкт має специфічні властивості та виконує специфічні задачі (включаючи сидіння й байдикування). Та ці об'єкти не можна віднести а ні до даних, а ні до функцій, оскільки реальні речі являють собою сукупність *властивостей* й *дій* (*поведінки)* [5 – 9].

**Властивості**

Прикладами властивостей для людей можуть бути колір очей, вік чи місце роботи; для машин – потужність двигуна, марка машини чи рік виробництва. Таким чином, властивості об'єктів рівносильні даним в програмі, бо вони мають певне значення: наприклад, 20 для віку людини чи 2005 для року випуску автомобіля.

**Поведінка**

Поведінка – це реакція об'єкта у відповідь на зовнішній вплив. Наприклад, ваш викладач у відповідь на прохання поставити вам залік може дати відповідь "так" чи "ні". Якщо ви натиснете на гальмо автомобіля, це призведе до його зупинки. Відповідь викладача та зупинка є прикладами поведінки (дії) об'єкта.

**Класи**

Однією з найбільш корисних особливостей мови C++ є можливість визначати власні типи даних, які краще підходитимуть для вирішення конкретних проблем. Ви вже бачили, як перерахування і структури можуть використовуватися для створення власних користувацьких типів даних.

Перерахування та структури — це традиційний (НЕ об'єктно-орієнтований) світ програмування, в якому ми можемо тільки зберігати дані.

В об'єктно-орієнтованому програмуванні типи даних можуть містити не тільки дані, але і функції, які працюватимуть з цими даними. Для визначення такого типу даних в мові C++ використовується ключове слово **class**. Використання ключового слова class визначає новий користувацький тип даних — **клас**.

У мові C++ класи дуже схожі на структури, за винятком того, що вони забезпечують набагато більшу потужність і гнучкість.

Так само, як і оголошення структури, оголошення класу не призводить до виділення будь-якої пам'яті. Для використання класу потрібно оголосити змінну цього типу класу:

У мові C++ змінну класу називають **екземпляром** (або "**об'єктом**") класу. Точно так же, як визначення змінної фундаментального типу даних (наприклад, int x) призводить до виділення пам'яті для цієї змінної, так само і створення об'єкта класу (наприклад, DateClass today) призводить до виділення пам'яті для цього об'єкта.

### Методи класів

Крім зберігання даних, класи можуть містити і функції! Функції, визначені всередині класу, називають **методами**. Методи можуть бути визначені, як всередині, так і поза класом. Поки що ми будемо визначати їх всередині класу (для простоти), як визначити їх поза класом — розглянемо трохи пізніше.

Точно так же, як до членів структури, так і до членів (змінних і функцій) класу доступ здійснюється через оператор вибору членів (.).

### Наслідування

Поняття класу приводить нас до поняття *наслідування*. В повсякденному житті ми часто стикаємося з розбиттям класів на підкласи: наприклад, клас «хордові» можна розбити на підкласи риби, земноводні, плазуни, птахи, ссавці. Клас «міський транспорт» поділяється на підкласи трамваї, тролейбуси, автобуси, метро, таксі тощо. Принцип, покладений в основу такого ділення,

полягає в тому, що кожен підклас володіє властивостями, притаманними тому класу, з якого виділений даний підклас. Крім тих властивостей, які є загальними для класу та підкласу, підклас може володіти й власними властивостями.

В програмуванні клас також може породити безліч підкласів. Цю можливість називають наслідуванням. Наслідування – це можливість, яка дозволяє одному класу наслідувати властивості іншого.

В мові C++ клас, що породжує всі інші класи (який наслідується), називають **базовим класом**. Класи, які наслідують базовий клас, називають **похідними**. Базовий клас містить елементи, спільні для групи похідних класів. А похідні класи наслідують всі його властивості, одночасно володіючи власними властивостями. Тобто кожен похідний клас представляє собою спеціалізовану версію базового класу.

Роль наслідування в ООП – це можливість створення ієрархії класів. Оскільки властивості при наслідуванні повторно не описуються, це скорочує обсяг програми та дозволяє спростити зв’язки між її елементами.

**Повторне використання коду**

В ООП концепція наслідування відкриває нові можливості програмування. А саме можливості повторного використання коду. Мова йде про те, що розроблений клас може бути використаний в інших програмах. Цю властивість називають можливістю **повторного використання коду**. Аналогічною властивістю в процедурному програмуванні володіють бібліотеки функцій, які можна включати в різноманітні програмні проекти.

Програміст може взяти існуючий клас, та, нічого не змінюючи, додати до нього свої елементи. Всі похідні класи унаслідують ці зміни, й в той же час кожен з похідних класів можна модифікувати окремо.

Наприклад, ви (чи хтось інший) розробили клас, що представляє систему меню, аналогічну графічному інтерфейсу Windows. Ви не хочете змінювати цей клас, але вам необхідно додати ще один пункт меню. В цьому випадку ви створюєте новий клас, що наслідує всі властивості вихідного класу, й додаєте в нього необхідний код.

Легкість повторного використання коду вже написаних програм є важливим достоїнством ООП.

### Користувацькі типи даних

Ще одним нововведенням став опис власних – **користувацьких** – **типів даних**, що дозволяло представляти інформацію в більш природному виді.

Наприклад, вам необхідно працювати з об’єктами, що мають дві координати, наприклад x та y. Вам хотілося б виконувати звичайні арифметичні дії над такими об’єктами, наприклад:

pos3 = pos1 + pos2

де змінні pos1, pos2 та pos3 являють собою набори з двох координат. Якщо ми опишемо клас Position, що включає в себе пару координат, то цим ми фактично створимо новий тип даних. А для роботи з подібними даними достатньо оголосити об’єкти цього класу з іменами pos1, pos2 та pos3.

### Поліморфізм та перевантаження

Зверніть увагу на те, що операції присвоювання (=) та додавання (+) для типу даних Position повинні виконувати дії, що відрізняються від тих, які ці операції виконують для об’єктів стандартних типів. Чи, наприклад, вам необхідно використати операцію додавання (+) для додавання інтервалів чи об’єднання рядків. Як же оператори = та + розпізнають, які саме дії необхідно здійснити над операндами? Відповідь полягає в тому, що ми самі можемо задавати ці дії, зробивши потрібні оператори методами відповідного класу (наприклад, класу Position).

Можливість використання операцій та функцій різним чином в залежності від того, з якими типами даних вони працюють, називають **поліморфізмом**.

Коли існуюча операція, наприклад = чи +, наділяється можливістю здійснювати дії над операндами нового користувацького типу, кажуть, що така операція є **перевантаженою**.

Перевантаженими можуть бути будь-які функції та операції (унарні чи бінарні). Перевантаження являє собою частковий випадок поліморфізму і є важливим інструментом ООП.

Таким чином, ООП – це не просто набір нових засобів, доданих у мову (на C++ можна успішно писати і без використання ООП). ООП часто називають новою парадигмою програмування. Красивий термін «парадигма» означає набір теорій, стандартів і методів, які разом представляють собою спосіб організації знань – іншими словами, спосіб бачення світу. У програмуванні цей термін використовують для визначення моделі обчислень, тобто способу структурування інформації, організації обчислень і даних. Об’єктно-орієнтована програма будується в термінах об’єктів і їх взаємозв’язків.

## Контрольні запитання

1. Які основні етапи еволюції парадигм програмування ви можете назвати? Чому постійно виникала потреба у створенні нових підходів?

2. Яку головну мету переслідував Б’ярн Страуструп при розробці мови C++? Які концепції він запозичив із мови C, а які – з інших мов?

3. У чому полягає базова концепція процедурного програмування? Як у цьому підході взаємодіють між собою функції (підпрограми) та дані?

4. Які три базові керуючі конструкції лежать в основі структурного програмування, дозволяючи відмовитися від оператора безумовного переходу goto?

5. З якими головними проблемами стикаються розробники при використанні виключно процедурного/структурного підходу у великих проєктах?

6. Чому відокремлення даних від функцій, які їх обробляють, стає критичним недоліком? Поясніть проблему використання глобальних змінних та ризики неконтрольованої зміни даних у різних частинах програми.

7. У чому полягає фундаментальна зміна мислення при переході від процедурного до об'єктно-орієнтованого підходу?

8. Що таке «об’єкт» у контексті ООП? Як він концептуально поєднує в собі стан (характеристики/дані) та поведінку (дії/методи)?

9. Як саме об’єктно-орієнтований підхід вирішує проблему вразливості даних, притаманну процедурним мовам?

## Лекція №10. Об’єкти і класи

### Об’єкти

Основною ідеєю об’єктно-орієнтованого підходу є об’єднання даних і дій, виконуваних над цими даними, в єдине ціле, яке називають **об’єктом** [2 – 5]. Тобто, головним компонентом об’єктно-орієнтованої програми є об’єкт. І замість того, щоб розглядати програму як набір послідовно виконуваних інструкцій, в ООП програма представляється у вигляді сукупності об’єктів.

Дані об’єкта, які в термінології C++ називають *членами-даними*, визначають його властивості, параметри, вигляд тощо.

Функції об’єкта, які в термінології C++ називають **методами** чи **функціями-членами**, зазвичай призначені для доступу до даних об’єкта.

Адже прямий доступ до даних неможливий. Дані скриті від зовнішнього впливу, що захищає їх від випадкової зміни. Кажуть, що дані та методи **інкапсульовані**.

Якщо необхідно змінити дані об’єкта, то, очевидно, ця дія також буде покладена на методи об’єкта. Ніякі інші «сторонні» функції не можуть змінити дані. Такий підхід полегшує написання, налагодження й використання програми.

Представити собі об’єкти можна, наприклад, чимось на зразок компанії – з відділами бухгалтерії, продажу, кадрів тощо. Поділ на відділи є важливою частиною структурної організації фірми. У більшості компаній в обов’язки співробітника не входить вирішення одночасно кадрових, торгівельних та обліково-бухгалтерських питань. Обов’язки чітко розподілено між підрозділами, і у кожного підрозділу є дані, з якими він працює: в бухгалтерії – заробітна плата, у відділу продажу – відомості щодо торгівлі, у відділу кадрів – персональна інформація про співробітників. Співробітники кожного відділу виконують операції лише з тими даними, які відносяться до даного відділу. Якщо у менеджера з продажу виникне необхідність дізнатися про сумарний оклад співробітників за серпень, то йому не потрібно буде самому йти до бухгалтерії та ритися в паперах. Йому буде достатньо зробити запит компетентній особі.

Така схема дозволяє забезпечити правильну обробку даних, а також її захист від можливого впливу сторонніх осіб.

## Класи

Та, хоча кожен об’єкт є унікальним, він підпадає під певну категорію, яку називають **класом** [1 – 4]. І коли ми говоримо про об’єкти, ми кажемо, що вони є **екземплярами класів**. Що це означає?

Клас є свого роду формою (зразком), що визначає, які дані й функції будуть включені в об’єкт цього класу. Тобто, клас можна представити як деякий шаблон, що визначає формат об’єкта.

Клас є описом сукупності схожих між собою об’єктів. А об’єкти є втіленням властивостей, атрибутів, притаманних класу, до якого вони, – об’єкти, – належать.

Хоча мій PC є унікальним, він відноситься до класу персональних комп’ютерів, так само як і ваші комп’ютери. Проте, кожна машина працює в різних умовах (різні версії Windows, файли, папки тощо) та виконує різні задачі (бух. облік, набір текстів, проектування техніки, діагностика хворого тощо). Але при цьому вони всі виконують такі задачі, які визначаються класом. Іншими словами, я не управляю своїм PC як автомобілем. Аналогічно, жодна з наших машин не має властивостей поза класом персональних комп’ютерів – наприклад, не використовує в якості джерела енергії ядерну енергію. Кожен PC – екземпляр класу персональних комп’ютерів.

А Prince, Sting, Madonna відносяться до класу рок-музикантів.

Використання класів є ключовою відмінністю ООП від структурного програмування. Класи – це серце C++ та головна складова об’єктно-орієнтованого програмування. Клас – це базова одиниця інкапсуляції C++. Клас являє собою модель реального об’єкта у вигляді даних і функцій для роботи з ними (рис. 10.1).


Рис.10.1. Клас об’єднує дані та функції

**Клас – це абстрактний тип даних, що визначається програмістом**. Класи використовують для створення (побудови) об’єктів. Таким чином, об’єкт знаходиться у такому ж відношенні до свого класу, в якому змінна знаходиться по відношенню до свого типу.

При цьому важливо розуміти, що **клас – це логічна абстракція**, котра реально не існує до тих пір, поки не буде створений об’єкт цього класу.

Ще раз зазначимо, що конкретні змінні типу даних «клас» називають *екземплярами класу*, або *об’єктами*. Програми, що розробляються на основі концепцій ООП, реалізують алгоритми, які описують власне взаємодію між об’єктами певних класів.

### Анатомія класу

Клас є описом того, як буде виглядати та як вести себе його представник – екземпляр класу.

Клас як тип даних дещо схожий на структуру. Клас, як і структура, являє собою набір даних та функцій, призначених для сумісного виконання певної задачі. Кажуть, що клас *інкапсулює* задачу.

**Інкапсуляція** – це обмеження доступу до даних і методів їх обробки. Інкапсуляція підвищує надійність програм, запобігаючи ненавмисному помилковому доступу до полів об’єкта.

Функції та змінні, що складають клас, називають його *членами*. Змінні, оголошені у класі, називають **полями класу**, а функції, оголошені у класі, називають **методами класу**. Передбачається, що доступ до полів класу можливий тільки через виклик відповідних методів класу.

Доступ до окремих частин класу регулюється за допомогою спеціальних засобів контролю доступу. Члени класу можуть мати три рівні доступу: ***закритий*** (private), ***відкритий*** (public) чи ***захищений*** (protected). Вони встановлюються при оголошенні класу.

У загальному випадку класи мають наступні характерні елементи:

- Засоби контролю доступу
- Члени-дані
- Функції-члени
- Конструктори та деструктори
- Спеціальний прихований покажчик з ім’ям this.

**Рівні доступу до членів класу**

Ми говорили, що ключовою особливістю ООП є можливість *приховування* даних. В тому розумінні, що дані знаходяться всередині класу та захищені від несанкціонованого доступу функцій, розташованих поза класом. Рівні доступу до членів класу управляють видимістю елементів класу, а отже визначають спосіб роботи користувачів із класом.

В мові C++ є три ключові слова, що встановлюють рівні доступу. Це слова private, public та protected.

Якщо необхідно захистити якісь дані, то їх описують з ключовим словом private. Такі дані доступні тільки всередині класу. Дані, описані з ключовим словом public, доступні за межами класу (рис. 10.2).

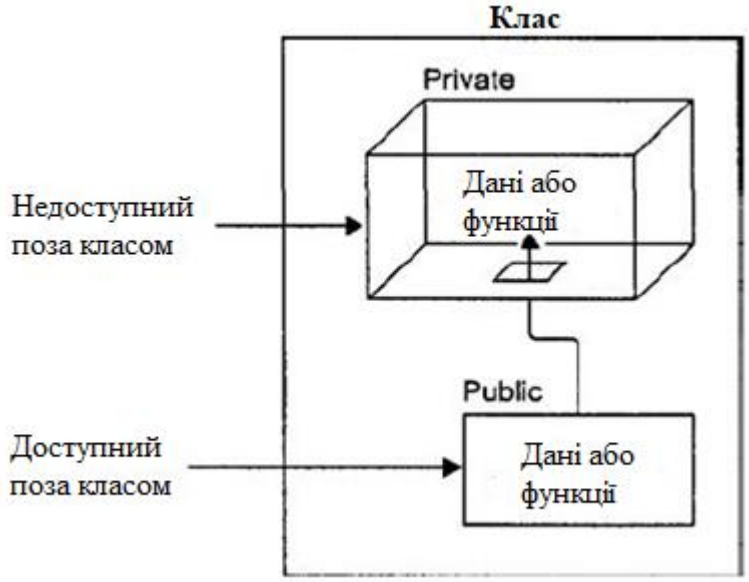



Рис.10.2. Рівні доступу до членів класу

Щоб зрозуміти роль рівнів доступу, перш за все необхідно розібратися в тому, як використовуються класи. Будь-який клас містить *відкриту* частину, та *закриту* частину. Методи, розташовані у *відкритій* частині, формують інтерфейс класу і можуть вільно викликатися клієнтом через відповідний об'єкт класу. *Закрита* частина класу визначає його внутрішню реалізацію. Доступ до *закритої* секції класу можливий тільки з його власних методів, а до *захищеної* – з його власних методів, а також з методів класів-нащадків. В гарно спроектованому класі від користувача приховано все, що йому не потрібно знати.

**Абстрактне представлення даних** *(data abstraction)* – це закриття внутрішньої реалізації властивостей класу від погляду ззовні.

Абстрактне представлення даних дає користувачу можливість знати про клас рівно стільки, скільки необхідно, й запобігає від втручання туди, куди втручатися не слід.

Коли ви сідаєте в авто й повертаєте ключ запалювання, чи потрібно вам знати всі подробиці щодо роботи машини? Звичайно, ні. Ви хочете знати рівно стільки, скільки необхідно для безпечної поїздки. В цій аналогії кермо, педалі, ручка перемикання передач, спідометр і т. і. являють собою інтерфейс між водієм та авто. Водій знає, якими з цих елементів потрібно користуватися, щоб машина їхала туди, куди він хоче. Можна сказати, що двигун, трансмісія та

електрообладнання авто приховані від погляду ззовні. Двигун буде працювати навіть в тому випадку, якщо ви ніколи не захочете відкрити капот. (Це робота СТО). Його будова – це деталі, які вам не потрібно знати. Вони скриті від вас. Уявіть, про яку кількість речей доведеться турбуватися, якщо слідкувати за всім, що відбувається в машині: чи правильно карбюратор готує суміш? Чи достатньо мастил у диференціалі? Чи вистачає напруги генератора? І т. д., і т. п. Кому це потрібно?

Таким самим чином в класах закритий доступ до внутрішніх деталей, щоб користувач міг не турбуватися про те, що відбувається „під капотом". Внутрішня робота класу закрита, тоді як інтерфейс користувача є відкритим.

Захищений рівень доступу пояснити дещо складніше. До захищених членів класу, як і до закритих, користувач звертатися не може. Проте ці члени можуть бути доступні для класів, які є похідними даного класу.

Рівні доступу до членів класу встановлюються при оголошенні класу.

**Оголошення класу**

Клас перед використанням має бути попередньо оголошений. Оголошення класу містить оголошення членів-даних і функцій-членів класу.

Оголошення класу зазвичай міститься в заголовному файлі. Для простих класів оголошення й визначення можуть бути розміщені в єдиному вихідному файлі програми. Проте при написанні реальних програм так не поступають. Як правило, для класу створюють окремий вихідний файл з ім'ям, близьким до імені класу, та розширенням .cpp. Заголовний файл для класу зазвичай має таке саме ім'я, що й вихідний файл, але з розширенням .h. Наприклад, якщо у вас є клас з ім'ям MyClass, то вихідний файл слід назвати myclass.cpp, а заголовний - myclass.h.

Клас оголошується за допомогою ключового слова **class**.

Загальний формат оголошення класу має наступний вигляд:

```
class ім'я_класу
{
 private:
```

```
   …          // закриті дані і функції
public:
   …          // відкриті дані і функції
protected:
   …          // захищені дані і функції
};
```

Не обов'язково використовувати всі три рівні доступу в своїх класах. Можна взагалі не встановлювати рівні доступу (за замовчуванням члени класу мають рівень доступу private). Тобто якщо ви не використовуєте при оголошенні класу модифікатори доступу, всі дані і функції класу будуть закритими (private). Дія будь-якого специфікатора поширюється до наступного специфікатора або до кінця класу. В оголошенні класу можна задавати кілька секцій private і public, порядок їх слідування значення не має.

Наприклад:

```
class MyClass
{
private:              // закриті дані і функції
  int date;
public:              // відкриті дані і функції
  void mfunc(int d)
   {date = d;}
};
```

Поля класу:

- можуть мати будь-який тип, крім типу цього ж класу (проте можуть бути покажчиками або посиланнями на цей клас);

- можуть бути описані з модифікатором const, при цьому вони ініціалізуються лише один раз (за допомогою конструктора) і не можуть змінюватися;

- можуть бути описані з модифікатором static, але не як auto, extern і register.

## Опис об’єктів

Ім’я класу стає ім’ям нового типу даних, яке можна використовувати для створення об’єктів класу. Об’єкти визначають у функціях або поза функціями як глобальні. Визначення об’єкта схоже на визначення змінної: воно означає виділення пам’яті, необхідної для зберігання об’єкта.

Наприклад:

```
MyClass D1, D2;          // Об’єкти класу з параметрами за замовчуванням
MyClass D3(200);         // Об’єкт з явною ініціалізацією
MyClass D4[100];         // Масив об’єктів з параметрами за замовчуванням
MyClass *D5 = new MyClass (10);   // Динамічний об’єкт
MyClass &D6=D1;          // Посилання на об’єкт
```

При виході об’єкта з області дії він знищується, при цьому автоматично викликається спеціальна функція – **деструктор**.

## Доступ до членів класу

Дані-члени класу та функції-члени мають **областю дії** клас. В області дії дані-члени класу безпосередньо доступні усім функціям-членам цього класу й на них можна посилатися просто за іменем. Поза областю дії до елементів класу можна звертатися або через ім’я об’єкта, або посиланням на об’єкт, або за допомогою вказівника на об’єкт.

Доступ до функцій (методів) класу можливий лише через конкретний об’єкт цього класу. Для цього використовують *операцію крапка* (**.**), яка пов’язує метод з ім’ям об’єкта.

При зверненні до елемента через покажчик використовують операцію стрілочка (->).

***Приклад 10.1*:** Цей приклад демонструє клас і два об’єкти цього класу. Незважаючи на свою простоту, він демонструє синтаксис і основні риси класів C++.

```
#include <iostream>
using namespace std;
class smallobj              // визначення класу
```

```
{
  private:
    int somedata;              // поле класу
  public:
    void setdata(int d)        // метод класу, що змінює значення поля
       { somedata = d; }
    void showdata()            // метод класу, що відображує значення поля
      { cout << "Значення поля дорівнює " << somedata << endl; }
};
int main()
{
  smallobj s1, s2;             // визначення двох об'єктів класу smallobj
  s1.setdata(1066);            // виклик методу setdata()
  s2.setdata(1776);
  s1.showdata();               // виклик методу showdata()
  s2.showdata();
  return 0;
}
```

Клас smallobj, визначений у цій програмі, містить одне поле даних і два методи. Методи забезпечують доступ до поля даних класу. Перший з методів setdata() привласнює полю значення, а другий showdata() – виводить це значення на екран. Тобто кожен з двох об'єктів має своє значення і здатний виводити це значення на екран.

Результат роботи програми виглядає наступним чином:

```
Значення поля дорівнює 1066
Значення поля дорівнює 1776
```

**Оголошення змінної класу**

Оголошуючи змінну класу, ми створюємо об'єкт. Після оголошення класу його об'єкти можна створювати за мірою необхідності.

Наприклад:

```
smallobj s1, s2;      // визначення двох об'єктів класу smallobj
```

Як варіант, об'єкти класу можна створити, вказавши їхні імена безпосередньо за фігурною дужкою, що закриває оголошення класу – в якості елемента списку об'єктів.

Наприклад:

```
class smallobj
{
  private:
    int somedata;
  public:
    void setdata(int d)
        { somedata = d; }
    void showdata()
      { cout << "Значення поля дорівнює " << somedata << endl; }
} s1, s2;
```

Визначення об'єкта схоже на визначення змінної: воно означає виділення пам'яті, необхідної для зберігання об'єкта.

Будь який клас може містити закриті, відкриті та захищені члени. Для управління доступом до членів класу використовують специфікатори private, public, protected в оголошенні класу. За замовчуванням всі елементи, визначені в класі, є закритими. Це означає, що до них можуть отримати доступ лише інші члени їхнього класу; ніякі інші частини програми цього зробити не можуть. Дані, що описані з ключовим словом public, напроти, не мають ніяких обмежень доступу за межами класу.

Як правило, приховуючи дані класу, його методи залишають доступними. Це пояснюється тим, що функції, які працюють з цими даними, мають забезпечувати взаємодію між даними й зовнішньою по відношенню до класу частиною програми.

**Методи класу всередині визначення класу**

Методи класу smallobj виконують дії, що є типовими для методів класу

взагалі: вони зчитують та присвоюють значення полям класу. Метод setdata( ) приймає аргумент й присвоює полю somedata значення, що дорівнює значенню аргумента. Метод showdata( ) відображує на екрані значення поля somedata.

Функції setdata( ) та showdata( ) визначені всередині класу, тобто код функції міститься безпосередньо у визначенні класу. Методи класу, визначені подібним чином, за замовчуванням є **вбудованими**. Однак, пізніше ми ще будемо говорити по те, що функції всередині класу можна не лише визначати, але й оголошувати, а визначення функції проводити в іншому місці. Функція, визначена поза класом, за замовчуванням вже не є вбудованою.

### Доступ до членів класу. Виклик методів класу

Для доступу до членів класу поза областю дії класу використовують **операцію «крапка» (.)**, яка зв'язує ім'я об'єкта з ім'ям його члена.

Так, для того, щоб отримати доступ до методів класу, необхідно використати операцію крапки (.), що зв'язує метод з ім'ям об'єкта. Доступ до методів класу можливий лише через конкретний об'єкт цього класу. Метод завжди виконує дії з конкретним об'єктом, а не з класом в цілому.

Наприклад:

```
s1.setdata(1066);     // метод надає полю somedata об'єкта s1 значення 1066
s2.setdata(1776);     // метод надає полю somedata об'єкта s2 значення 1776
```

### Клас як тип даних

Розглянемо приклад, що демонструє застосування об'єктів C++ в якості змінних типу, визначеного користувачем, тобто розглянемо клас як тип даних. Об'єкти будуть представляти собою довжини, що виражені в англійській системі мір:

```
// ENGLOBJ, довжини в англійській системі мір як об'єкти
#include <iostream>
using namespace std;
class Distance
{
  private:
```

```
    int feet;                    // число футів
    float inches;                // число дюймів
  public:
    void setdist( int ft, float in )// метод установки значень полів
     {
      feet = ft;
      inches = in;
     }
    void getdist()          // метод введення полів з клавіатури
     {
      cout << "\n Введіть число футів:  ";       cin >> feet;
      cout << "\n Введіть число дюймів:  ";     cin >> inches;
    }
    void showdist()         // метод виведення полів на екран
     { cout << feet << "\'-" << inches << "\" "; }
};
int main()
{
  Distance dist1, dist2;    // визначення двох об'єктів класу Distance
  dist1.setdist(11,6.25);   // установка значень для d1
  dist2.getdist();          // введення значень для dist2
  // виведення довжин на екран
  cout << "\ndist1 = ";    dist1.showdist();
  cout << "\ndist2 = ";    dist2.showdist();
  cout << endl;
  return 0;
}
```

В цій програмі клас Distance містить два поля (feet та inches) та три методи: setdist(), призначений для надання значень полів об'єкта через аргументи, що йому передаються; getdist(), який отримує ці значення з клавіатури, та showdist(),

що відображає на екрані довжини у футах і дюймах.

Зверніть увагу, що значення полів об’єкта класу Distance можуть бути задані двома способами. Значення полів для об’єкта dist1 задаються за допомогою функції setdist(), що викликана з аргументами 11 та 6.25:

dist1.setdist(11,6.25);

А значення полів для об’єкта dist2 вводяться з клавіатури користувачем:

dist2.getdist().

Результат роботи програми:

Введіть число футів: 10

Введіть число дюймів: 4.75

dist1 = 11’- 6.25”

dist2 = 10’- 4.75”

## Конструктори

Розглянутий приклад продемонстрував два способи використання методів класу для ініціалізації полів об’єкта класу: 1) як надання значень полів об’єкта через аргументи, що йому передаються, та 2) отримання цих значень з клавіатури.

Проте, зручніше ініціалізувати поля об’єкта автоматично в момент його створення, а не явно викликати в програмі відповідний метод. Для цього класи в C++ мають спеціальну функцію, яку називають *конструктором*.

**Конструктор –** це метод класу, що виконується автоматично в момент створення об’єкта класу [5].

## Ім’я конструктора

У конструкторів є декілька особливостей, що відрізняють їх від усіх інших методів класу.

По-перше, ім’я конструктора точно співпадає з ім’ям класу. Таким чином компілятор відрізняє конструктори від інших методів класу.

По-друге, конструктор не може повертати ніякого значення. Якщо в оголошенні конструктора вказати тип значення, що треба повернути, компілятор видасть повідомлення про помилку. Це пояснюється тим, що конструктор

автоматично викликається системою, і, відповідно, не існує програми чи функції, яка його викликає та якій конструктор міг би повернути значення. Тобто конструктори не можна викликати як звичайні функції. До речі, за цією ознакою компілятор також може відрізнити конструктори від інших методів класу.

Наприклад:

```
// COUNTER, лічильник в якості об'єкта
#include <iostream>
using namespace std;
class Counter
{
  private:
    unsigned int count;        // значення лічильника
  public:
    Counter() : count(0)       // конструктор
      {   }
    void inc_count()           // інкремент лічильника
      { count++; }
    int get_count()            // отримання значення лічильника
      { return count; }
};
int main()
{
  Counter c1, c2;              // визначення з ініціалізацією
  cout <<"\nc1=" << c1.get_count();          // виведення
  cout <<"\nc2=" << c2.get_count();
  c1.inc_count();              // інкремент c1
  c2.inc_count();              // інкремент c2
  c2.inc_count();              // інкремент c2
  cout <<"\nc1=" << c1.get_count();          // виведення
  cout <<"\nc2=" << c2.get_count();
```

```
  cout << endl;
  return 0;
}
```

Результат роботи програми:

```
c1=0
c2=0
c1=1
c2=2
```

### Автоматична ініціалізація

У розглянутому прикладі, коли створюється об’єкт типу Counter, його поле count ініціалізується нульовим значенням, оскільки більшість лічильників починають відлік саме з нуля. Ці дії виконує конструктор Counter(). **Конструктор викликається автоматично** при створенні кожного з об’єктів.

Таким чином, у функції main() оператор

```
Counter c1, c2;
```

створює два об’єкти типу Counter. При створенні кожного з них викликається конструктор Counter(), який ініціалізує поле count нульовим значенням. Тож, крім створення змінних c1, c2, даний оператор ще присвоює їхнім полям нульове значення.

### Список ініціалізації

Саме задача ініціалізації полів об’єкта класу є однією з найбільш класичних з точки зору використання конструкторів. У розглянутому прикладі для кожного об’єкта класу Counter конструктор виконує ініціалізацію поля нулем. Ініціалізація відбувається наступним чином:

```
Counter() : count(0)
  {    }
```

Як бачимо, ініціалізація розташована між прототипом методу й тілом функції й випереджена **символом «двокрапка» (:).** Значення, яким ініціалізують поле, розміщене у дужках після ім’я поля.

Якщо необхідно ініціалізувати відразу кілька полів класу, то значення

розділяють комами – отримуємо **список ініціалізації**.

Наприклад:

```
SomeClass() : m1(7), m2(33), m3(4)
   {     }
```

Звісно, ініціалізація поля нулем може бути прописана й у тілі конструктора.

Наприклад наступним чином:

```
Counter()
  { count = 0; }
```

Однак, у тілі конструктора, як правило, прописують більш складні дії, ніж звичайну ініціалізацію.

Наприклад:

```
// конструктор друкує повідомлення під час виконання
Counter() : count(O)
{ cout << "Конструктор\n"; }
```

У такій редакції результат роботи програми виглядатиме наступним чином:

```
Конструктор
Конструктор

c1=0
c2=0
c1=1
c2=2
```

## Конструктор за замовчуванням

У розглянутій програмі про довжини в англійській системі мір як об’єкти не було конструктора, проте визначення об’єктів класу Distance працювали без помилок. Чому ж вони працювали без конструктора?

Це пояснюється тим, що компілятор автоматично вбудовує в програму **конструктор без параметрів**, який й створює об’єкти класу, не дивлячись на те,

що явного визначення конструктора не було. Такий конструктор без параметрів називають **конструктором за замовчуванням**. Якби конструктор за замовчуванням не створювався автоматично, ми не змогли б визначати змінні класів, в яких відсутній конструктор.

Однак ми не можемо знати, якими значеннями були ініціалізовані поля об’єкта класу конструктором за замовчуванням. Якщо для нас це важливо, то ми повинні явно визначати конструктор.

Для наведеного прикладу це можна зробити наступним чином:

```
Distance() : feet(0), inches(0.0)        // конструктор за замовчуванням
  {  }
```

Цей конструктор ініціалізує члени класу Distance константними значеннями, в даному випадку цілим значенням 0 для змінної feet та дійсним значенням 0.0 для змінної inches. Отже, ми можемо використовувати об’єкти, які ініціалізує конструктор без параметрів, й бути впевненими, що поля об’єкта мають нульові, а не інші, випадкові, значення.

### Перевантажені конструктори. Конструктор з параметрами

Клас може мати кілька конструкторів з різними параметрами для різних видів ініціалізації (при цьому використовується механізм перевантаження).

Внесемо зміни у приклад ENGLOBJ, який був розглянутий вище:

```
// ENGLCON
#include <iostream>
using namespace std;
class Distance              // довжина в англійській системі
{
  private:
    int feet;
    float inches;
  public:
    Distance() : feet(0),inches(0.0)       // конструктор без аргументів
      {   }
```

```
   // конструктор з двома аргументами
   Distance(int ft,float in) : feet(ft),inches(in)
     {   }
   void getdist()                    // введення довжини користувачем
   {
     cout << "\nВведіть число футів: "; cin >> feet;
     cout << "Введіть число дюймів: "; cin >> inches;
   }
   void showdist()                   // виведення довжини на екран
     { cout << feet << "\'-" << inches << '\"'; }
};
int main()
{
  Distance dist1, dist3;             // дві довжини
  Distance dist2(11, 6.25);          // визначення та ініціалізація
  dist1.getdist();                   // введення dist1
  // виведення всіх довжин
  cout << "\ndist1 = "; dist1.showdist();
  cout << "\ndist2 = "; dist2.showdist();
  cout << "\ndist3 = "; dist3.showdist();
  cout << endl;
  return 0;
}
```

Ми додали конструктор

```
Distance(int ft, float in) : feet(ft), inches(in)
    {     }
```

який виконує ініціалізацію змінних класу Distance у момент їхнього створення. Ми ініціалізуємо поля feet та inches тими значеннями, які передаються конструктору в якості аргументів.

```
  Distance dist2(11, 6.25);     // визначення та ініціаліація
```

Тобто тепер у нас є два явно визначених конструктора з одним й тим самим ім'ям Distance(), і тому кажуть, що конструктор є **перевантаженим**. Який з цих двох конструкторів виконується під час створення нового об'єкта, залежить від того, скільки аргументів використовується при виклику конструктора:

```
Distance dist1, dist3;       // викликається перший конструктор
Distance dist2(11, 6.25); // викликається другий конструктор
```

**Конструктор копіювання за замовчуванням**

Ми розглянули два способи ініціалізації об'єктів за допомогою конструктора: конструктор за замовчуванням (без аргументів) може ініціалізувати поля об'єкта константними значеннями, а конструктор, який має хоча б один аргумент, – може ініціалізувати поля значеннями, які передаються йому в якості аргументів.

Тепер розглянемо третій спосіб ініціалізації об'єкта – за допомогою **конструктора копіювання**. Це спеціальний вид конструктора, що викликається в тих випадках, коли новий об'єкт створюється шляхом копіювання вже існуючого об'єкта. Такий конструктор надається компілятором для кожного створюваного класу автоматично.

Копіюючий конструктор має єдиний аргумент, який є об'єктом того ж самого класу, що й конструктор, і виконує поелементне копіювання полів.

Нприклад, наступна програма демонструє використання конструктора копіювання за замовчуванням:

```
// ECOPYCON, ініціалізація об'єктів за допомогою конструктора копіювання
#include <iostream>
using namespace std;
class Distance       // довжина в англійській системі мір
{
  private:
    int feet;
    float inches;
  public:
```

```
   Distance() : feet(0), inches(0.0)      // конструктор без аргументів
     {   }
   // конструктора з одним аргументом немає!
   // конструктор з двома аргументами
   Distance(int ft, float in) : feet(ft), inches(in)
     {   }
   void showdist()                  // виведення довжини
     { cout << feet << "\'-" << inches << '\"'; }
};
int main()
{
  Distance dist1(11, 6.25);        // конструктор з двома аргументами
  Distance dist2(dist1);           // конструктор з одним аргументом
  Distance dist3 = dist1;
  // виведення всіх довжин
  cout <<"\ndist1 = ";    dist1.showdist();
  cout <<"\ndist2 = ";    dist2.showdist();
  cout <<"\ndist3 = ";    dist3.showdist();
  cout << endl;
  return 0;
}
```

Об’єкт dist1 ініціалізується значенням 11’-6.25” за допомогою конструктора з двома аргументами.

Потім ми визначаємо та ініціалізуємо ще два об’єкти з іменами dist2 та dist3, і обидва ініціалізуються значенням об’єкта dist1.

Здавалося б, у даному випадку повинен бути викликаний конструктор з одним аргументом. Проте, оскільки аргументом є об’єкт того ж класу, що й об’єкти, які ініціалізуються, то в обох випадках викликаний конструктор копіювання за замовчуванням.

Об’єкт dist2 ініціалізований за допомогою оператора

Distance dist2(dist1);

Дія копіюючого конструктора за замовчуванням зводиться до копіювання значень полів об’єкта dist1 у відповідні поля об’єкта dist2.

Ідентичні дії для пари об’єктів dist1 та dist3 виконуються за допомогою оператора

Distance dist3 = dist1;

Можна подумати, що даний оператор виконує операцію присвоювання. Але ні – тут, як і в попередньому випадку, викликається конструктор копіювання за замовчуванням.

Обидва оператори виконують однакові дії та рівноправні у використанні.

Результат роботи програми виглядає наступним чином:

dist1 = 11’- 6.25”

dist2 = 11’- 6.25”

dist3 = 11’- 6.25”

Отже, конструктор викликається, якщо в програмі зустрілась якась з синтаксичних конструкцій:

**ім’я_класу імя_об’єкта [(список параметрів)];**

// *список параметрів не повинен бути порожнім*

**ім’я_класу (список параметрів);**

// *створюється об’єкт без імені (список може бути порожнім)*

**ім’я_класу імя_об’єкта = вираз;**

// *створюється об’єкт без імені та копіюється*

Наприклад:

monstr Super(200, 300), Vasia(50), Z;

monstr X = monstr(1000);

monstr Y = 500;

## Деструктори

Отже, особливий метод класу – конструктор – викликається при створенні об’єкта класу. Проте, існує інша функція, яка автоматично викликається при знищенні об’єкта. Зветься вона **деструктором** [7].

Деструктор має ім’я, що співпадає з ім’ям конструктора (а відповідно, і з ім’ям класу) та випереджається **символом «тильда» ( ~ )**.

Наприклад:

```
Class Demo
{
 private:
   int data;
 public:
   Demo() : data(0)        // конструктор
     {  }
   ~Demo()                 // деструктор
     {  }
};
```

Подібно до конструкторів, деструктори не повертають значень й не мають аргументів. Найбільш вживане застосування деструкторів – вивільнення пам’яті, що виділяється конструктором при створенні об’єкта.

Як ми зазначали, клас може мати кілька конструкторів, але деструктор у нього може бути лише один.

Наприклад, наступна програма демонструє використання деструктора.

```
#include <iostream>
using namespace std;
class MyClass {
public:
  int x;
  MyClass(int i);           // оголошення конструктора
  ~MyClass();               // оголошення деструктора
};
MyClass::MyClass(int i) // Реалізація конструктора
{
   x = i;
```

```
}
MyClass::~MyClass()     // Реалізація деструктора
 {
  cout << "Знищення об’єкта x із значенням " <<  x  <<" \n";
}
int main() {
  MyClass t1(5);
  MyClass t2(19);
  cout << t1.x << " " << t2.x << "\n";
  return 0;
}
```

**Контрольні запитання**

1. Що таке інкапсуляція в об'єктно-орієнтованому програмуванні? Яку головну мету вона переслідує з точки зору захисту внутрішнього стану об’єкта від зовнішнього втручання?

2. У чому полягає практична різниця між модифікаторами доступу public та private? Які частини програми мають доступ до приватних полів та методів класу?

3. Який модифікатор доступу використовується за замовчуванням у класах (class) та структурах (struct) у мові C++?

4. Що таке конструктор і в який саме момент життєвого циклу об’єкта він викликається?

5. Які існують жорсткі синтаксичні правила для оголошення конструктора? Яким має бути його ім’я та чи може він повертати якесь значення?

6. Що таке «конструктор за замовчуванням» (default constructor)? У якому випадку компілятор C++ згенерує його автоматично, а за яких умов відмовиться це робити, вимагаючи написати його власноруч?

7. Як працює перевантаження (overloading) конструкторів? Як компілятор розуміє, який саме з кількох написаних конструкторів потрібно викликати при

створенні конкретного об’єкта?

8. Що таке список ініціалізації (member initializer list) у конструкторі? Чому ініціалізація полів через цей список вважається більш ефективною практикою, ніж просте присвоювання значень у тілі конструктора?

9. Що таке деструктор і за яких умов він викликається автоматично?

10. Який синтаксис використовується для оголошення? Чи може деструктор приймати аргументи і чи може в класі бути більше одного деструктора?

11. У яких випадках програмісту критично необхідно власноруч прописувати логіку в тілі деструктора? Які проблеми можуть виникнути, якщо цього не зробити при роботі з оператором new всередині класу?

## Лекція №11. Наслідування

### Поняття про наслідування

Однією з найбільш потужних і цікавих проектних властивостей класів у C++ є можливість їх повторного використання та розширення шляхом *наслідування*.

**Наслідування** – це механізм створення нових класів шляхом додавання нових членів до вже існуючого класу [7 – 9].

У термінології C++ клас, до якого додаються нові члени, тобто клас, що наслідується, називають **базовим класом**. А клас, який наслідує базовий клас, називають **похідним класом**.

Наслідування, як слідує з назви, дозволяє одному класу запозичувати характеристики (атрибути і функції) іншого (рис. 11.1).

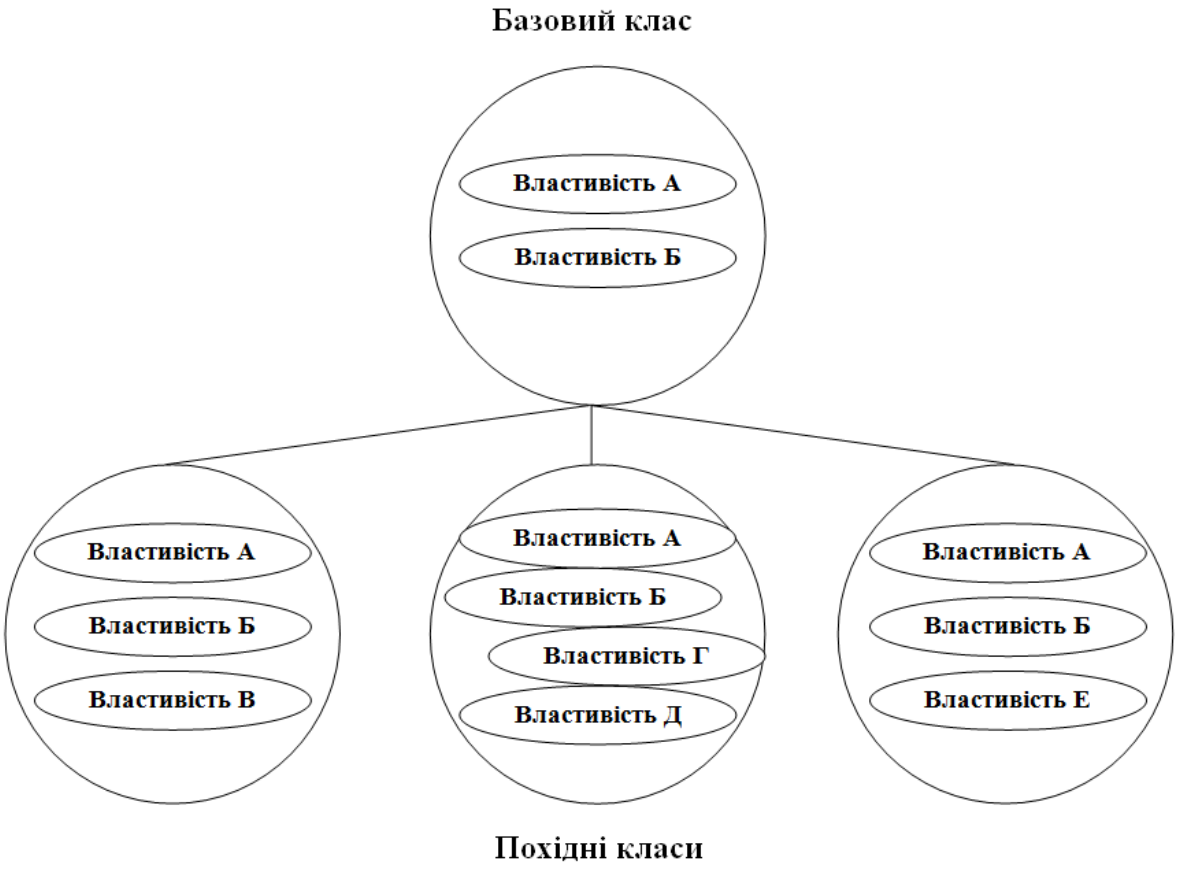



Рис. 11.1. Механізм наслідування

При цьому похідний клас отримує всі функціональні можливості базового класу, й може бути вдосконалений за рахунок додавання власних.

Зверніть увагу! Не можна знищувати нічого, що належить базовому класу. Базовий клас при наслідуванні залишається незмінним.

Похідний клас наслідує всі члени, визначені в базовому класі, й додає до них власні унікальні елементи. Тобто похідний клас у загальному випадку більше свого базового класу, з більшою кількістю властивостей, ніж базовий. Похідний клас являє собою спеціалізовану версію базового класу, однак він більш вузький, ніж базовий клас, й представляє меншу групу об’єктів.

Кожен об’єкт похідного класу є також об’єктом відповідного базового класу. Проте, зворотне твердження невірне: об’єкт базового класу не є об’єктом класів, породжених цим класом.

Зверніть увагу! Наслідування не працює у зворотному напрямку. Тобто базовому класу та його об’єктам недоступні похідні класи.

Наслідування – важлива частина ООП. Його перевага полягає в тому, що механізм наслідування дозволяє використовувати існуючий код декілька разів. Повторне використання коду економить час розробки програм. Маючи гарно написаний базовий клас, його можна застосувати для роботи в різних ситуаціях.

Очевидно, що кожен похідний клас сам є кандидатом на роль базового класу для майбутніх похідних класів. Тож, механізм наслідування дозволяє будувати деревовидні ієрархії класів.

Приклади наведено в табл. 11.1 та на рис. 11.2:

Табл. 11.1. Деякі приклади наслідування

| **Базовий клас** | **Похідний клас** |
|---|---|
| Студент | Випускник<br>Не закінчив навчання |
| Форма | Коло<br>Трикутник<br>Прямокутник |
| Позика | Позика на автомобіль<br>Позика на ремонт будинка<br>Іпотека |
| Працівник | Член факультету<br>Допоміжний персонал |
| Рахунок | Чековий рахунок<br>Ощадний рахунок |

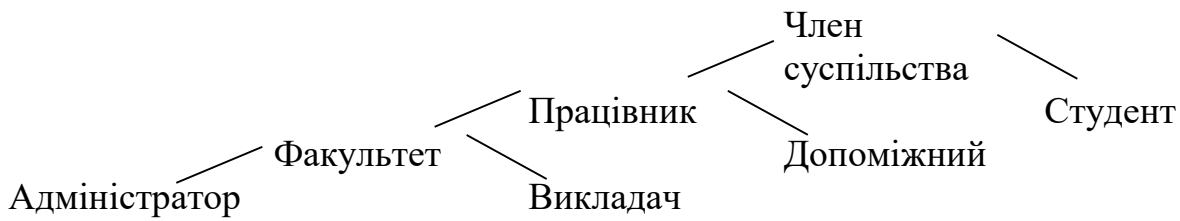



Рис. 11.2. Ієрархія наслідування для членів університетської спільноти

Класи, що знаходяться ближче до початку ієрархії, об’єднують в собі найбільш загальні риси для всіх класів. У міру просування вниз ієрархічною структурою класи набувають все більш конкретних рис.

Для різних методів класу існують різні правила наслідування – так зокрема, конструктори і операція присвоювання в похідному класі не наслідуються, а деструктори наслідуються.

**Визначення похідного класу**

Опис в програмі нового класу вказує на те, що він є похідним класом від іншого класу. Для цього використовують **знак двокрапки (:)**, за ним слідує специфікатор доступу та ім’я базового класу.

Загальний синтаксис оголошення класу, що наслідує базовий клас, має такий вигляд:

```
class ім’я_похідного_класу : специфікатор_доступу ім’я_базового_класу
{
  // тіло похідного класу
};
```

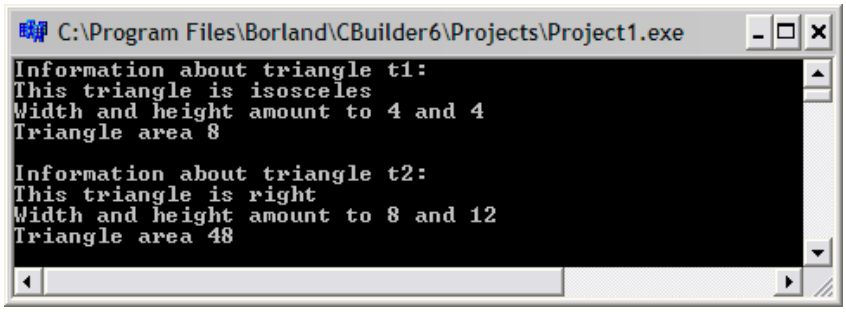
***Приклад 11.1*:** на основі базового класу TwoShape, в якому зберігаються основні параметри (ширина та висота) двовимірного об’єкта, створений похідний клас Triangle.

```
// TRIANGLE, створення похідного класу
#include <iostream>
#include <cstring>
using namespace std;
class TwoShape                    // оголошення базового класу
{
```

```
 public:
  double width;
  double height;
  void showDim()
  {
   cout << "Width and height amount to "
        << width << " and " << height << "\n";
  }
};
class Triangle : public TwoShape       // оголошення похідного класу
{
 public:
  char style[20];
  double area()
   { return width* height/2; }
  void showStyle()
   {cout << "This triangle is" << style << "\n"; }
};
int main()
{
 Triangle t1, t2;
 /*всі члени класу Triangle доступні для об'єктів класу Triangle, навіть ті, що
успадковані від класу TwoShape*/
 t1.width=4.0;
 t1.height =4.0;
 strcpy(t1.style,"isosceles");      // рівнобедрений
 t2.width=8.0;
 t2.height =12.0;
 strcpy(t2.style,"right");          // прямокутний
 cout << "Information about triangle t1:\n";
```

```
 t1.showStyle();
 t1.showDim();
 cout<<"Triangle area "<<t1.area()<<"\n"<<"\n";
 cout << "Information about triangle t2:\n";
 t2.showStyle();
 t2.showDim();
 cout<<"Triangle area "<<t2.area()<<"\n";
return 0;
}
```

Результат роботи програми виглядає наступним чином:




В даному прикладі клас TwoShape визначає атрибути «узагальненої» двохвимірної фігури (наприклад, квадрату, прямокутника, трикутника тощо). У класі Triangle створюється специфічний тип об’єкта класу TwoShape, в даному випадку – трикутник. Клас Triangle містить всі елементи класу TwoShape й, крім того, поле style, функцію area() та функцію showStyle(). У змінній style зберігається опис типу трикутника, функція area() обчислює й повертає його площу, а функція showStyle() відображає дані про тип трикутника. Схематично включення класу TwoShape у клас Triangle зображено на рис. 11.3:

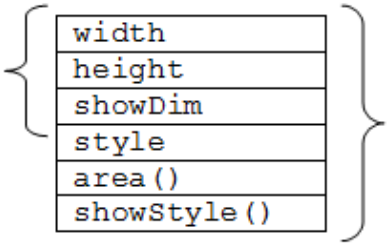



Рис. 11.3. Включення класу TwoShape у клас Triangle

Оскільки клас Triangle включає всі члени базового класу TwoShape, то він може звертатися до членів width й height всередині методу area(). Крім того, всередині функції main() об’єкти t1 й t2 можуть безпосередньо посилатися на члени width й height так, наче вони є частиною класу Triangle.

**Управління доступом до членів базового класу**

Знання того, коли для об’єктів похідного класу можуть бути використані методи базового класу, є важливою темою в наслідуванні. Це називають **правами доступу.**

Синтаксис оголошення похідного класу Triangle від базового класу TwoShape у наведеному прикладі наступний:

```
class Triangle : public TwoShape
{…};
```

Статус доступу членів базового класу в похідному класі визначається специфікатором доступу, який використовують для наслідування базового класу. Базовий клас може наслідуватися як закритий (private), відкритий (public) або захищений (protected). Специфікатор доступу базового класу виражають відповідно одним з ключових слів: **private, public** чи **protected** (табл. 11.2) [2 – 5].

Зверніть увагу! Якщо специфікатор доступу не вказаний, то за замовчуванням використовується специфікатор private.

Захищене й закрите наслідування зустрічається рідко й кожне з них треба використовувати дуже обережно. Як правило, використовують відкрите наслідування.

При породженні класу як public відкриті члени базового класу стають відкритими членами похідного класу, а захищені члени базового класу стають захищеними членами похідного класу. Закриті члени базового класу ніколи не бувають доступні для похідного класу.

При захищеному protected-наслідуванні відкриті та захищені члени базового класу стають захищеними членами похідного класу. Закриті члени базового класу ніколи не бувають доступні для похідного класу.

При закритому private-наслідуванні відкриті та захищені члени базового класу стають закритими членами похідного класу. Закриті члени базового класу ніколи не бувають доступні для похідного класу.

Зверніть увагу! Для закритого та захищеного наслідування не є справедливим відношення, що об’єкт похідного класу є об’єктом базового класу. В усіх випадках private-члени базового класу залишаються закритими в рамках цього класу й не є доступними для членів похідного класу.

Табл. 11.2. Типи наслідування

| **Специфікатор доступу до членів у базовому класі** | **Тип наслідування** | | |
|---|---|---|---|
| | **public**<br>**відкрите**<br>**наслідування** | **protected**<br>**захищене**<br>**наслідування** | **private**<br>**закрите**<br>**наслідування** |
| **public** | **public**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам, дружнім функціям та функціям, що не є членами. | **protected**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам та дружнім функціям. | **private**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам та дружнім функціям. |
| **protected** | **protected**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам та дружнім функціям. | **protected**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам та дружнім функціям. | **private**<br>у похідному класі<br><br>Може бути доступним безпосередньо будь-яким нестатичним функціям-членам та дружнім функціям. |
| **private** | Невидимий<br>у похідному класі<br><br>Може бути доступним нестатичним функціям-членам та дружнім функціям через відкриті чи захищені функції-члени базового класу. | Невидимий<br>у похідному класі<br><br>Може бути доступним нестатичним функціям-членам та дружнім функціям через відкриті чи захищені функції-члени базового класу. | Невидимий<br>у похідному класі<br><br>Може бути доступним нестатичним функціям-членам та дружнім функціям через відкриті чи захищені функції-члени базового класу. |

Як видно з таблиці, private елементи базового класу в похідному класі недоступні незалежно від ключа. Звернення до них можливе лише через методи базового класу.

Якщо базовий клас наслідується з ключем private, можна вибірково зробити деякі його елементи доступними в похідному класі, оголосивши їх в секції public похідного класу за допомогою операції доступу до області видимості.

Наприклад:

```
class Base
{
  …
  public: void f ();
};
class Derived : private Base
{
  …
  public: Base :: void f ();
};
```

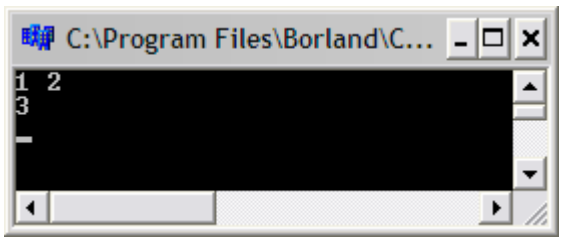
***Приклад 11.2*:** Розглянемо на простому прикладі як статус доступу впливає на роботу з даними базового класу.

```
// демонстрація public-наслідування.
#include <iostream>
using namespace std;
class B                    // оголошення базового класу
{
 private:
  int i, j;
 public:
  void set(int a, int b)
   { i=a; j=b; }
```

```
  void show()
   { cout<<i<<" "<<j<<"\n"; }
};
class D : public B          // оголошення похідного класу
{                           // клас B наслідується відкритим способом
 private:
  int k;
 public:
  D(int x)
   { k=x; }
  void showk()
   { cout<<k<<"\n"; }
};
int main()
{
 D obj(3);
 obj.i=10;                  // Помилка! Член i закритий в рамках класу B
 obj.set(1,2);              // Отримуємо доступ до членів базового класу
 obj.show();                // Отримуємо доступ до членів базового класу
 obj.showk();               // Використовується член похідного класу
return 0;
}
```

Оскільки функції set() та show() є відкритими public-членами класу B, то їх можна викликати для об'єкта obj типу D з функції main(). Але оскільки змінні i та j визначені як закриті private-члени, вони залишаються закритими в рамках свого класу B. Клас D не може отримати доступ до закритого члену класу B. Звідси маємо помилку при зверненні до змінної i.

Результат роботи програми виглядає наступним чином:



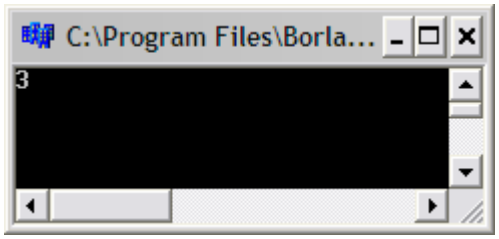

***Приклад 11.3*:** Протилежністю до відкритого наслідування є закрите (private) наслідування .

```
// демонстрація закритого способу наслідування
#include <iostream>
using namespace std;
class B                    // оголошення базового класу
{
 private:
  int i, j;
 public:
  void set(int a, int b)
   { i=a; j=b; }
  void show()
   { cout<<i<<” ”<<j<<”\n”; }
};
class D : private B          // клас B наслідується закритим способом
 private:
  int k;
 public:
  D(int x)
   { k=x; }
  void showk()
   { cout<<k<<”\n”; }
};
int main()
```

```
{
 D obj(3);
// тепер функції set() та show() недоступні для частин програми поза класом D
 obj.set(1,2);      // Помилка! Доступ до функції set() неможливий
 obj.show();        // Помилка! Доступ до функції show() неможливий
 obj.showk();
return 0;
}
```

Отже, якщо базовий клас наслідується як private-клас, його відкриті члени стають закритими (private) членами похідного класу. Це означає, що вони доступні для членів похідного класу, але не доступні для інших частин програми.

Результат роботи програми виглядає наступним чином:




## Множинне наслідування

**Простим** називають наслідування, при якому похідний клас має одного предка.

Клас може бути похідним не лише від одного базового класу, але й від кількох (рис. 11.4).

Створення класу на основі двох чи більше класів називають **множинним наслідуванням**.

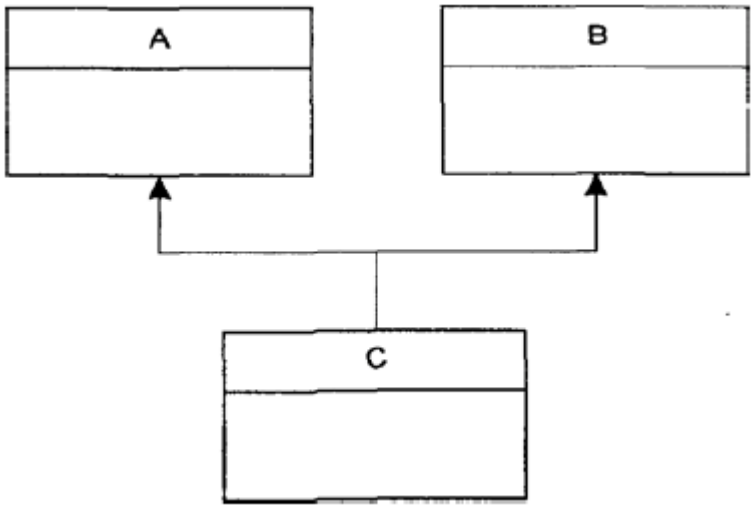



Рис. 11.4. Діаграма класів при множинному наслідуванні

Синтаксис опису множинного наслідування схожий на синтаксис простого наслідування: при оголошенні класу в його заголовку після **символу двокрапки (:)** перераховують через кому всі класи, що є для нього базовими. Ключ доступу може стояти перед кожним базовим класом.

Наприклад:

```
class A
 {
 };
class B
 {
 };
class C : public A, public B
 {
 };
```

Проте, множинне наслідування використовується не часто, але може виявитися дуже корисним.

### Заміщення функцій базового класу

Іноді в похідному класі потрібна дещо інша реалізація методу, наслідуваного з базового класу. Мова C++ дозволяє це зробити за рахунок *заміщення (перевизначення)* методу.

**Заміщення (перевизначення)** методу проводиться шляхом оголошення у п
о Якщо виникає необхідність все ж таки викликати з похідного класу метод б
азового, використовують операцію доступу до області видимості – **символ (::)**.
і Наприклад:

```
// DERIVED, заміщення функцій базового класу
#include <iostream>
using namespace std;
class Base                          // оголошення базового класу
{
  public:
  void Display( )
  { cout << " Hello, world! " << endl; }
};
class Derived : public Base         // оголошення похідного класу
{
  public:
  void Display()
  {
   Base::Display() ;                // виклик методу базового класу
   cout << " How are you? "<< endl;
  }
};
class SubDerived : public Derived   // оголошення похідного класу
{
  public:
  void Display()
  {
   Derived::Display() ;             // виклик методу базового класу
   cout << " Bye! " << endl;
  }
```



```
c
```

```
    }
};
int main()
{
 SubDerived sd;
 sd.Display();
 return 0;
}
```

Результат виконання програми:

```
Hello, world!
How are you?
Bye!
```

### Конструктори і деструктори в похідному класі

**Конструктори** з базового класу не наслідуються**,** тому похідний клас повинен мати власні конструктори (якщо вони потрібні).

У питанні «бути чи не бути» конструкторам похідного класу, керуються наступними правилами:

- Якщо в базовому класі взагалі немає конструктора або є конструктор за замовчуванням, то похідному класу конструктор потрібен тільки в тому випадку, коли необхідно ініціалізувати поля, введені в цьому класі.

- Якщо в похідному класі не визначено жодного конструктора, компілятор самостійно створить конструктор за замовчуванням, з якого буде викликаний конструктор за замовчуванням базового класу.

- Якщо в базовому класі є конструктор з параметрами, то в похідному класі, як правило, потрібно задати конструктор зі списком аргументів, який включатиме значення для передачі конструктору базового класу. Цей конструктор базового класу треба викликати в списку ініціалізації.

Зверніть увагу! Якщо конструктор базового класу вимагає зазначення параметрів, то він має бути викликаний явним чином в конструкторі похідного класу у списку ініціалізації.

Розглянемо цей випадок на прикладі наступної програми.

```
// MULTISHAPE, геометричні фігури
#include <msoftcon.h>
class shape                    // оголошення базового класу
{
  protected:
    int xCo, yCo;              // координати фігури
    color fillcolor;           // колір
    fstyle fillstyle;          // стиль замальовування
  public:
    // конструктор без параметрів
    shape ( ) : xCo ( 0 ), yCo ( 0 ), fillcolor ( cWHITE ), fillstyle ( SOLID_FILL )
      { }
    // конструктор с чотирма параметрами
    shape ( int x, int y, color fc, fstyle fs ) : xCo ( x ), yCo ( y ), fillcolor ( fc ),
                fillstyle ( fs )
      {    }
    void draw ( ) const                    // функція установки кольору та стилю
      {
        set_color ( fillcolor );
        set_fill_style ( fillstyle );
      }
};
class circle : public shape                // оголошення похідного класу circle
{
  private:
    int radius;              // радіус, а xCo та yCo будуть координатами центру
  public:
    circle ( ) : shape ( )                 // конструктор без параметрів
      {   }
```

```
   // конструктор с п'ятьма параметрами
   circle ( int x, int y, int r, color fc, fstyle fs ) : shape ( x, y, fc, fs ), radius ( r )
    {   }
   void draw ( ) const                    // функція малювання кола
    {
     shape::draw ( );
     draw_circle ( xCo, yCo, radius );
    }
};
class rect : public shape                 // оголошення похідного класу rect
{
 private:
   int width, height;        /* ширина та висота, а xCo и yCo будуть координатами
                                правого верхнього кута*/
 public:
   // конструктор без параметрів
   rect ( ) : shape ( ), height ( 0 ), width ( 0 )
    {   }
   // конструктор с шістьма параметрами
   rect ( int x, int y, int h, int w, color fc, fstyle fs ) : shape ( x, y, fc, fs ),
            height ( h ), width ( w )
    {    }
   void draw ( ) const                    // функція малювання прямокутника
    {
     shape::draw ( );
     draw_rectangle ( xCo, yCo, xCo + width, yCo + height );
     set_color ( xWHITE );                // малюємо діагональ
     draw_line ( xCo, yCo, xCo + width, yCo + height );
    }
};
```

```
class tria : public shape                    // оголошення похідного класу tria
{
  private:
    int height; // висота піраміди, а xCo і yCo будуть координатами вершини
  public:
    tria ( ) : shape ( ), height ( 0 )       // конструктор без параметрів
      { }
    // конструктор с п'ятьма параметрами
    tria ( int x, int y, int h, color fc, fstyle fs ) : shape ( x, y, fc, fs ), height ( h )
      { }
    void draw ( ) const                      // функція малювання піраміди
      {
        shape::draw ( );
        draw_pyramid( xCo, yCo, height );
      }
};
int main ( )
{
  init_graphics ( ); // ініціалізуємо систему відображення графіки
  circle cir ( 40, 12, 5, cBLUE, X_FILL );          // створюємо коло
  rect rec ( 12, 7, 10, 15, cRED, SOLID_FILL );     // створюємо прямокутник
  tria tri ( 60, 7, 11, cGREEN, MEDIUM_FILL );  // створюємо піраміду
  cir.draw ( );        // малюємо коло
  rec.draw ( );        // малюємо прямокутник
  tri.draw ( );        // малюємо піраміду
  set_cursor_pos( 1, 25 );          // переводимо курсор у низ екрану
  return 0;
}
```

**Порядок виклику конструкторів** визначається наведеними нижче правилами:

- Якщо в конструкторі похідного класу явний виклик конструктора базового класу відсутній, автоматично викликається конструктор базового класу за замовчуванням (тобто той, який можна викликати без параметрів).

- Для ієрархії, що складається з декількох рівнів, конструктори базових класів викликаються починаючи з самого верхнього рівня. Після цього виконуються конструктори тих елементів класу, які є об’єктами, в порядку їх оголошення в класі, а потім виконується конструктор похідного класу.

- У випадку декількох базових класів їхні конструктори викликаються в порядку оголошення.

Зверніть увагу! Не наслідується й *операція присвоювання*, тому її також потрібно явно визначати у похідному класі.

Необхідність в деструкторі для похідного класу визначається тим, чи потрібно звільняти будь-які ресурси, виділені в конструкторі. Якщо такої необхідності немає, то можна довірити компілятору створити деструктор за замовчуванням. У ньому забезпечується виклик деструктора базового класу.

Нижче перераховано **правила наслідування деструкторів**:

- *Деструктори не наслідуються*, і якщо програміст не описав у похідному класі деструктор, він формується за замовчуванням і викликає деструктори всіх базових класів.

- На відміну від конструкторів, при написанні деструктора похідного класу в ньому не потрібно явно викликати деструктори базових класів, оскільки це буде зроблено автоматично.

- Для ієрархії класів, що складається з декількох рівнів, деструктори викликаються в порядку, строго зворотному виклику конструкторів: спочатку викликається деструктор похідного класу, потім - деструктори елементів класу, а потім деструктор базового класу.

### Доступ до об’єктів ієрархії

Найефективніше працювати з об’єктами однієї ієрархії через покажчик на базовий клас. Справа в тім, що при відкритому (public) наслідуванні такому

покажчику можна присвоювати адресу об’єкта як базового класу, так і будь-якого похідного класу.

Наприклад:

```
// Описується покажчик на базовий клас
  Base* p:
// Покажчик посилається на об’єкт похідного класу
  p = new Derived:
```

Така можливість видається цілком логічною: в якості представника більш загального класу (наприклад, класу Багатокутники) використовується об’єкт спеціалізованого класу (наприклад, класу Трикутники або Чотирикутники).

Однак, якщо не вжити спеціальних заходів, то при роботі з таким покажчиком – незалежно від того, яку адресу йому присвоєно – виявляються доступними лише елементи базового класу. Причиною є те, що в загальному випадку під час компіляції компілятор не може знати, з яким об’єктом насправді пов’язаний цей покажчик.

Розглянемо наступний фрагмент програми:

```
class Base
{
 public:
 // . . .
   void Modify();
};
class Derived : public Base              // оголошення похідного класу
{
 public:
   void Modify();
};
int main()
{
 int mode;
```

```
 Base* pB;
 cin >> mode;
 if (mode == 1) pB = new Base;
    else            pB = new Derived;
 pB->Modify();                          // на що вказує pB?
return 0;
}
```

Виклик методів об’єкта відбувається відповідно до типу покажчика, а не за фактичним типом об’єкта, на який він посилається.

Оскільки компілятор не може передбачити, який вибір буде зроблений на етапі виконання, то він обирає метод за типом покажчика pB, тобто
Base :: Modify();

Таку стратегію називають **раннім**, або **статичним, зв’язуванням**. Тому, якщо в похідному класі був заміщений деякий метод базового класу (наприклад, Derived::Modify()), то він виявляється недоступним.

Продемонструємо це на прикладі:

```
#include <iostream>
using namespace std;
class Base                          // оголошення базового класу
{
  public:
    Base(int _x = 10) { x = _x; }
    void ShowX() { cout<< "x = " << x << "; "; }
    void ModifyX() { x *= 2; }
  protected:
    int x;
};
class Derived : public Base       // оголошення похідного класу
{
  public:
```

```
    void ModifyX() { x /= 2; }    // заміщений метод базового класу
};
int main()
{
  Base b;
  Derived d;
  b.ShowX();
  d.ShowX();
  Base* pB;
  pB = &b;  pB->ModifyX();  pB->ShowX();
  pB = &d;  pB->ModifyX();  pB->ShowX();
return 0;
}
```

Результат виконання програми:

x = 10; x = 10; x = 20; x = 20;

Очевидно, що другий виклик методу ModifyX() відбувається також з базового класу.

**Віртуальні методи**

Проблема доступу до методів, перевизначених у похідних класах, через покажчик на базовий клас вирішується в C++ за допомогою використання **віртуальних методів** [5, 8].

Щоб зробити деякий метод віртуальним, треба в базовому класі його заголовок випередити **специфікатором virtual**. Після цього він буде сприйматися як віртуальний у всіх похідних класах ієрархії.

Наприклад:

```
virtual void ModifyX()
{
   x *= 2;
}
```

По відношенню до всіх віртуальних методів компілятор застосовує стратегію **пізнього**, або **динамічного, зв'язування**.

Це означає, що на етапі компіляції він не визначає, який з методів повинен бути викликаний, а передає відповідальність програмі, яка приймає рішення на етапі виконання, коли вже точно відомо, яким є тип об'єкта, на який вказує покажчик.

Все сказане відноситься також до виклику методів за посиланням на базовий клас.

Щоб побачити динамічне зв'язування в дії, модифікуємо наведену вище програму додаванням специфікатора virtual перед заголовком методу ModifyX() в базовому класі, так що тепер визначення методу набуде вигляду:

```
virtual void ModifyX() { x *= 2; }
```

Результат роботи програми виглядатиме наступним чином:

x = 10; x = 10; x = 20; x = 5;

тобто другий виклик методу ModifyX() відбувається тепер з похідного класу.

Віртуальний механізм працює тільки при використанні покажчиків і посилань на об'єкти. Об'єкт, який визначений через покажчик або посилання і який містить віртуальні методи, називають **поліморфним**.

Якщо базовий клас містить хоча б один віртуальний метод, то рекомендується завжди визначати в цьому класі віртуальний деструктор, навіть якщо він нічого не робить. Наявність такого віртуального деструктора запобіжить некоректному видаленню об'єктів похідного класу, адресованих через покажчик на базовий клас, тому що в противному випадку деструктор похідного класу викликаний не буде.

## Абстрактні класи

Іноді, розробляючи ієрархію класів, можна отримати базовий клас, для якого створення об'єкта як би втрачає сенс. Наприклад, ієрархія класів геометричних фігур може виходити з базового класу Shape, а в якості похідних виступатимуть класи Polygon, Ellipse тощо. Одним з методів базового класу буде,

мабуть, функція Show () – показати об’єкт. Але як можна показувати об’єкт, який не має конкретної форми?

Класи, для яких немає сенсу створювати об’єкти, оголошують як *абстрактні*.

**Абстрактний клас** – це клас, що містить хоча б один чисто віртуальний метод.

**Чисто віртуальний метод** – це метод, оголошений в класі, але який не має конкретної реалізації.

Синтаксис оголошення чисто віртуального методу доповнюється конструкцією **= 0**.

Наприклад:

```
virtual void Show() = 0;
```

Компілятор не допустить створення об’єкта для абстрактного класу.

Якщо в похідному класі хоча б одна чисто віртуальна функція базового класу залишиться без конкретної реалізації, то такий похідний клас також буде абстрактним класом, для якого заборонено створювати об’єкти.

**Наслідування в класі Distance**

Розглянемо наслідування на прикладі програми про довжини в англійській системі мір. Будемо вимірювати рівень Тихого океану з урахуванням припливів та відливів. Нам може знадобитися від’ємне значення інтервалу. Для цього створимо новий похідний клас від класу Distance. В цей клас додамо поле для наших вимірювань, яке міститиме знак „+” чи „-”. А також методи роботи зі знаковими значеннями інтервалу.

```
// ENGLEN, наслідування в програмі англійських мір довжини
#include <iostream>
using namespace std;
enum posneg { pos, neg };
class Distance        // клас для англійських мір довжини
{
  protected:          // увага! private використовувати не можна
```

```
    int feet;
    float inches;
  public:
    Distance ( ) : feet ( 0 ), inches ( 0.0 )    // конструктор без параметрів
      { }
// конструктор с двома параметрами
    Distance ( int ft, float in ) : feet ( ft ), inches ( in )
      { }
    void getdist ( )                    // отримання значень від користувача
      {
        cout << "\nВведіть фути: "; cin >> feet;
        cout << "Введіть дюйми: "; cin >> inches;
      }
    void showdist ( ) const          // виведення значень на екран
      { cout << feet << "\' " << inches << "\' "; }
};
class DistSign : public Distance          // похідний клас
{
  private:
    posneg sign;                    // значення може бути pos чи neg
  public:
    // конструктор без параметрів - виклик конструктора базового класу
    DistSign ( ) : Distance ( )
      { sign = pos; }
    // конструктор с двома чи трьома параметрами,
    // виклик конструктора базового класу
    DistSign ( int ft, float in, posneg sg = pos ) : Distance ( ft, in )
      { sign = sg; }             // початковий знак
    void getdist ( )             // введення користувачем довжини
      {
```

```
      Distance::getdist ( );          // виклик функції getdist з базового класу
      char ch;                        // запит знаку
      cout << "Введіть знак (+ чи -): "; cin >> ch;
      sign = ( ch == '+' ) ? pos : neg;
    }
  void showdist ( ) const          // виведення відстані
    {
     // виведення всієї інформації, включаючи знак
      cout << ( ( sign == pos ) ? "(+)" : "(-)" );
      Distance::showdist ( );
    }
};
int main ( )
{
 DistSign alpha;       // використовуємо конструктор за замовчуванням
 alpha.getdist ( );    // отримаємо дані від користувача
 DistSign beta ( 11, 6.25 );              // конструктор с двома аргументами
 DistSign gamma ( 100, 5.5, neg );    // конструктор с трьома аргументами
 // виведення даних для всіх змінних
 cout << "\nA = "; alpha.showdist ( );
 cout << "\nB = "; beta.showdist ( );
 cout << "\nC = "; gamma.showdist ( );
 cout << endl;
 return 0;
}
```

**Специфікатор доступу protected**

Клас Distance в цій програмі такий самий, як і раніше, за виключенням того, що його дані feet та inches оголошені як protected. Це зроблене для того, щоб мати до них доступ з похідного класу, проте не втрачати можливість приховування даних. Член класу, оголошений як protected, доступний методам

свого класу та методам будь-якого похідного класу. При цьому він не доступний з функції, що не належить до цих класів, наприклад з функції main().

Таким чином, якщо ви пишете клас, який надалі буде використовуватись як базовий клас при наслідуванні, то дані, до яких потрібно буде мати доступ, слідує оголошувати як protected.

**Клас DistSign.** Клас DistSign є похідним від класу Distance. В нього додано поле sign типу posneg. Це поле призначене для зберігання знаку інтервалу. Тип posneg визначений в операторі enum та має два можливі значення – pos та neg.

**Методи класу DistSign.** Додавання поля sign в клас DistSign має значення для обох його методів. Метод getdist() повинен надіслати запит користувачеві про знаки значень футів та дюймів. Метод showdist() повинен вивести знаки для футів та дюймів. Ці методи викликають відповідні методи з класу Distance в рядках

```
Distance::getdist ( );
Distance::showdist ( );
```

**Конструктори класу DistSign.** Клас DistSign має два конструктори, такі самі, як і клас Distance. Перший не має аргументів, у другого два та три аргументи. Третій, необов'язковий, аргумент другого конструктора – це змінна sign, що має значення pos чи neg. Значенням за замовчуванням є pos. Ці конструктори дозволяють визначати об'єкти типу DistSign в різний спосіб.

Обидва конструктори в класі DistSign викликають відповідні конструктори з класу Distance для встановлення значень футів та дюймів. Вони також встановлюють значення поля sign. Конструктор за замовчуванням завжди встановлює поле sign у значення pos. Другий конструктор встановлює поле sign у значення pos, якщо воно не визначене явно.

**Робота програми.** У функції main() оголошено три інтервали. Інтервал alpha отримує своє значення від користувача, beta ініціалізуються конструктором з двома аргументами значенням (+)11'-6.25", gamma – конструктором з трьома аргументами значенням (-)100'- 5.5".

Результат роботи програми виглядає наступним чином:

Введіть фути: 6

Введіть дюйми: 2.5

Введіть знак (+ чи -): -

A = (-)6’-2.5”

B = (+)11’- 6.25”

C =(-)100’-5.5”

## Контрольні запитання

1. Що таке наслідування в ООП? Яку головну проблему розробки програмного забезпечення воно вирішує?

2. Який синтаксис створення похідного класу в C++? Що саме успадковується від базового класу, а які спеціальні елементи НЕ успадковуються автоматично?

3. Як правильно ініціалізувати успадковані поля при створенні об’єкта похідного класу? Як і де саме потрібно викликати конструктор базового класу?

4. У чому полягає призначення модифікатора доступу protected? Чим він відрізняється від private для зовнішніх користувачів класу і для його нащадків?

5. Чи має похідний клас прямий доступ до private полів свого базового класу? Якщо ні, то за допомогою яких інструментів він може безпечно зчитувати або змінювати ці дані?

6. Як тип (специфікатор) самого наслідування впливає на права доступу до успадкованих членів? Наприклад, що станеться з public методами базового класу при приватному наслідуванні (class Derived : private Base)?

7. Що таке множинне наслідування? Наведіть приклад життєвої моделі, де похідний клас логічно повинен успадковувати властивості одразу двох абсолютно різних базових класів.

## Лекція №12. Перевантаження

### Поняття про перевантаження

Ще однією з найзахоплюючих можливостей ООП є **перевантаження** [2 – 7]. Воно може перетворити складний і малозрозумілий лістинг програми в інтуїтивно зрозумілий.

На попередніх лекціях ми торкнулися питання перевантаження функцій, зокрема конструкторів класу.

Нагадаємо, що під **перевантаженням функції** розуміється визначення декількох функцій (двох або більше) з однаковим ім’ям, але різними параметрами і з різною програмною логікою. Набори параметрів перевантажених функцій можуть відрізнятися порядком слідування, кількістю, типом тощо.

Сьогодні говоритимемо про **перевантаження операцій**. Якщо ви вважаєте, що обмежені стандартними можливостями операцій, то ви можете «навчити» їх виконувати потрібні вам дії. C++ дозволяє перевизначити дію більшості операцій так, щоб при використанні з об’єктами конкретного класу вони виконували необхідні функції.

Коли існуюча операція, наприклад = чи +, наділяється можливістю здійснювати дії над операндами нового користувацького типу, кажуть, що така операція є **перевантаженою**.

Тож, перевантаження операторів виконують для того, щоб мати можливість використовувати стандартні вбудовані операції C++ для виконання дій над об’єктами класів.

Перевантаження операцій дає можливість використовувати власні типи даних так само, як стандартні. А також органічно інтегрувати нові класи в існуюче програмне середовище, оскільки після перевантаження операції над об’єктами нових класів виглядають аналогічно операціям над змінними вбудованих типів.

Для класу можна визначити (перевантажити) практично всі C++-оператори, наприклад: +, –, *, /, %, [], (), ^, !, &, <, <=, > чи >=.

Зверніть увагу! Перевантажувати можна тільки існуючі операції. Неможливо визначити свій власний оператор, наприклад **$=** чи ******.

Оператори, перевантаження яких **заборонено** з міркувань безпеки:

- оператор доступу до члена класу – крапка (.),
- оператор розйменування покажчика на член класу – (.*),
- оператор глобального дозволу – подвійна двокрапка (::),
- тернарний оператор умови – (? :),
- символ препроцесора – дієз (#),
- символ препроцесора – подвійний дієз (##),
- оператор знаходження розміру об’єкта в байтах – sizeof.

На перевантаження операторів накладається **ряд обмежень**:

- Не можна змінювати кількість операндів, що приймаються оператором. Наприклад, можна визначити унарний оператор –, але неможливо перевантажити як унарний оператор <= («менше або дорівнює»). Аналогічно можна перевантажити бінарний оператор +, але не можна перевантажити оператор ! («Ні») як бінарний.

- Не можна змінювати порядок виконання операцій та їх пріоритет.

- Не можна змінювати правила асоціації (справа наліво чи зліва направо).

- Не можна перевантажувати операції для стандартних типів даних.

- Операторні функції не можуть визначатися як static.

- Операторні функції наслідуються (за виключенням операції присвоювання =).

## Операторні функції

Перевантаження операцій здійснюється за допомогою функцій спеціального виду – **операторних функцій** з ім’ям **operator** [3].

Загальний формат оголошення операторної функції має такий вигляд:

```
тип operator операція (список_параметрів)
{
  …  // тіло функції
};
```

Тип – це тип значення, що повертається функцією operator. І хоча він в принципі може бути будь-яким, тип значення, що повертається, часто збігається з ім'ям класу, для якого перевантажується даний оператор. Така кореляція полегшує використання перевантаженого оператора у складених виразах.

Наприклад:

```
void operator++ ( ) { };
ThreeD operator+ (ThreeD) { };
Distance operator+ (Distance) const { };
```

Взагалі, існує два основних способи реалізації операторної функції: як функції-члена самого класу або як глобальної (тобто зовнішньої) функції, дружньої для класу.

Роб Мюррей, в своїй книзі «C++ Strategies and Tactics» визначив наступні рекомендації щодо вибору форми перевантаження оператора (табл. 12.1):

Табл. 12.1. Рекомендації щодо вибору форми перевантаження оператора

| **Оператор** | **Рекомендована форма** |
|---|---|
| Всі унарні оператори | Член класу |
| = () [] -> ->* | Обов'язково член класу |
| += -= /= *= ^= &= \|= %= >>= <<= | Член класу |
| Інші бінарні оператори | Не член класу |

У загальному випадку при використанні функції-члена для перевантаження унарного оператора параметри не використовуються взагалі, а для перевантаження бінарного – лише один параметр.

У більшості випадків, оператори (крім умовних) повертають об'єкт, або посилання на тип, до якого відносяться його аргументи.

Наприклад:

```
int operator+(int,int);      // помилка: неможливо перевантажити
                             // вбудований оператор додавання +
Vector operator+(const Vector&, const Vector &);      // OK
Vector operator+=(const Vector&, int);                // OK
```

**Перевантаження унарних операцій**

Унарні операції мають лише один операнд (*операнд* – це змінна, на яку діє

операція). Прикладом унарної операції можуть служити операції інкремента та декремента (++ та --) чи унарний мінус (наприклад -33).

Унарна операторна функція, що визначається всередині класу, повинна бути представлена за допомогою нестатичного методу без параметрів, при цьому операндом є об’єкт, що її викликав.

У прикладі COUNTER ми створили клас Counter для зберігання значення лічильника. Об’єкт цього класу ми збільшували викликом методу:

```
c1.inc_count( );     // інкремент лічильника
```

Метод виконував свою роботу, але лістинг програми стане більш зрозумілим, якщо ми застосуємо замість цього запису операцію інкремента ++.

Перепишемо програму COUNTER, щоб зробити цю дію можливою.

```
// COUNTPP1, збільшення змінної операцією ++
#include <iostream>
using namespace std;
class Counter
{
  private:
    unsigned int count;              // значення лічильника
  public:
    Counter ( ) : count ( 0 )        // конструктор за замовчуванням
      { }
    unsigned int get_count ( )       // отримати значення
      { return count; }
    void operator++ ( )               // збільшити значення
      { ++count; }
};
int main ( )
{
  Counter c1, c2;                    // визначаємо змінні типу Counter
  cout << "\nc1 = " << c1.get_count ( );        // виводимо на екран
```

```
  cout << "\nc2 = " << c2.get_count ( );
  ++c1;                          // збільшуємо c1
  ++c2;                          // збільшуємо c2
  ++c2;                          // збільшуємо c2
  cout << "\nc1 = " << c1.get_count ( );       // виводимо на екран
  cout << "\nc2 = " << c2.get_count ( ) << endl;
  return 0;
}
```

Результат виконання програми:

```
c1 = 0
c2 = 0
c1 = 1
c2 = 2
```

Як бачимо, у функції main() операція інкремента ++ застосована до об'єкта (++c1, ++c2). Вона збільшує змінну count об'єкта, до якого застосовується. Оскільки методи класу завжди мають доступ до об'єктів класу, для яких вони були викликані, то для цієї операції не потрібні аргументи. Це показано на рис. 12.1:

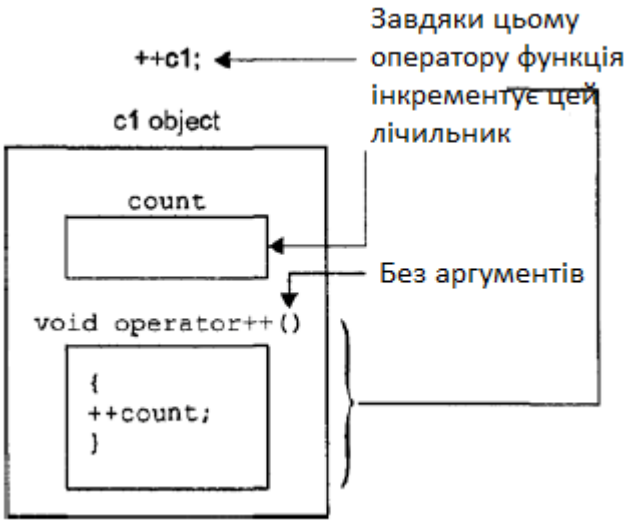



Рис.12.1. Перевантаження унарної операції без аргументів

У цій програмі ми визначили тип void для значення, що повертається функцією operator ++ (). І при спробі виконати, наприклад, інструкцію

```
c1 = ++c2;          // помилка: неможливо присвоїти
```

компілятор буде протестувати, оскільки в такому виразі присвоювання буде запитана змінна типу Counter.

Модифікуємо програму COUNTPP1. Для того щоб мати можливість використовувати написану операторну функцію operator ++ () у виразах присвоювання, нам необхідно правильно визначити тип значення, що нею повертається. Це зроблено в наступній програмі COUNTPP2.

```
// COUNTPP2, операція ++, що повертає значення
#include <iostream>
using namespace std;
class Counter
{
  private:
    unsigned int count;              // значення лічильника
  public:
    Counter ( ) : count ( 0 )        // конструктор за замовчуванням
      { }
    unsigned int get_count ( )       // отримати значення
      { return count; }
    Counter operator++ ( )           // префіксний інкремент
      {
         ++count;
         Counter temp;
         temp.count = count;
         return temp;
      }
};
  Counter c1, c2;                    // визначаємо змінні типу Counter
  cout << "\nc1 = " << c1.get_count ( );        // виводимо на екран
  cout << "\nc2 = " << c2.get_count ( );
  ++c1;                              // збільшуємо c1
```

```
  c2=++c1;                          // c1=2, c2=2
  cout << "\nc1 = " << c1.get_count ( );       // виводимо на екран
  cout << "\nc2 = " << c2.get_count ( ) << endl;
  return 0;
}
```

Результат виконання програми:

```
c1 = 0
c2 = 0
c1 = 2
c2 = 2
```

У прикладі COUNTPP2 ми створили тимчасовий об’єкт temp типу Counter, єдиною метою якого було збереження значення, що повертається для операції ++.

Існує багато шляхів повернення тимчасового об’єкта з функції або перевантаженої операції. Розглянемо ще один із способів в програмі COUNTPP3:

```
// COUNTPP3, операція ++, що повертає значення
#include <iostream>
using namespace std;
class Counter
{
  private:
    unsigned int count;             // значення лічильника
  public:
    Counter ( ) : count ( 0 )       // конструктор за замовчуванням
      { }
    Counter ( int c ) : count ( c )
      { }
    unsigned int get_count ( )      // отримати значення
      { return count; }
    Counter operator++ ( )          // збільшити значення
      {
         ++count;
```

```
        return Counter (count);
    }
};
 Counter c1, c2;                    // визначаємо змінні типу Counter
 cout << "\nc1 = " << c1.get_count ( );       // виводимо на екран
 cout << "\nc2 = " << c2.get_count ( );
 ++c1;                              // збільшуємо c1
 c2=++c1;                           // c1=2, c2=2
 cout << "\nc1 = " << c1.get_count ( );       // виводимо на екран
 cout << "\nc2 = " << c2.get_count ( ) << endl;
 return 0;
}
```

Результат роботи програми виглядає наступним чином:

```
c1 = 0
c2 = 0
c1 = 2
c2 = 2
```

У рядку цієї програми

return Counter (count);

створюється об’єкт типу Counter. Він не має імені, оскільки воно ніде не буде використовуватися. Цей об’єкт ініціалізується значенням, отриманим в якості параметра count. Для цього нам знадобився конструктор з одним аргументом у списку методів класу.

**Перевантаження бінарних операцій**

Бінарні операції можуть бути перевантажені так само, як і унарні операції.

Існує правило: об’єкт, що стоїть ліворуч від знака операції, викликає функцію оператора. Об’єкт, що стоїть праворуч від знака операції, повинен бути переданий у функцію в якості аргумента [4].

Отже, перевантаженій операції завжди потрібна кількість аргументів на одиницю менша, ніж кількість операндів, оскільки одним з операндів є об’єкт, що викликає функцію. Власне тому для унарних операцій не потрібні аргументи (це правило не є вірним для функцій та операторів, які є дружніми для класу).

В наступному прикладі створюється клас ThreeD, який забезпечує підтримку координат об’єктів у тривимірному просторі. Для класу ThreeD реалізоване перевантаження операторів додавання (+) та присвоювання (=).

```
// THREED, перевантаження операторів (+) та (=)
include <iostream>
using namespace std;
class ThreeD                    // оголошення класу
{
   int x, y, z;                 // 3-вимірні координати
   public:
        ThreeD () { x = y = z = 0; }
        ThreeD (int i, int j, int k) { x = i; y = j; z == k; }
        ThreeD operator+(ThreeD op2);       // операнд op1 передається неявно
        ThreeD operator=(ThreeD op2);       // операнд op1 передається неявно
void show() ;
};
// перевантаження оператора (+) для класу ThreeD
ThreeD ThreeD :: operator+ (ThreeD op2)
{
  ThreeD temp;
  temp.x = x + op2.x;       // При виконанні операції додавання
  temp.y = y + op2.y;       // цілочисельних значень зберігається
  temp.z = z + op2.z;       // оригінальний зміст оператора "+"
  return temp;              // повертається новий об’єкт
}
// перевантаження оператора (=) для класу ThreeD
ThreeD ThreeD :: operator= (ThreeD op2)
{
  x = op2.x;          // При виконанні операції присвоювання
  y = op2.y;          // цілочисельних значень зберігається
```

```
  z = op2.z;            // оригінальний зміст оператора "="
  return *this;         // повертається модифікований об'єкт
}
// відображення координат x, y, z
void ThreeD::show ()
{
cout <<x <<", ";
cout <<y <<", ";
cout <<z <<"\n ";
}
int main()
{
  ThreeD a(1, 2, 3), b(10, 10, 10), c;
  cout << " Початкове значення координат об'єкта a: ";
  a.show ();
  cout << " Початкове значення координат об'єкта b: ";
  b.show();
  cout << "\n";
  c = a + b;                    // додавання об'єктів a та b
  cout << " Значення координат об'єкта c = a + b: ";
  c.show ();
  cout << "\n";
  c = a + b + c;                // додавання об'єктів a, b та c
  cout << "Значення координат об'єкта c = a + b + c : ";
  c.show () ;
  cout <<"\n";
  c = b = a;                    // демонстрація множинного присвоювання
  cout << " Значення координат об'єкта c після c = b = a: ";
  c.show ();
  cout << " Значення координат об'єкта b після c = b = a: ";
```

```
  b.show ();
  return 0;
}
```

Результат роботи програми виглядає наступним чином:

```
Початкове значення координат об’єкта a: 1, 2, 3
Початкове значення координат об’єкта b: 10, 10, 10
Значення координат об’єкта c = a + Ь: 11, 12, 13
Значення координат об’єкта c = a + b + c : 22, 24, 26
Значення координат об’єкта c після c = b = a:  1, 2, 3
Значення координат об’єкта b після c = b = a:  1, 2, 3
```

Принциповий виграш від перевантаження операторів полягає в тому, що воно дозволяє органічно інтегрувати нові типи даних у середовище програмування.

Насправді використання класів для створення нових типів змінних і перевантаження для визначення нових операцій дають можливість фактично перевизначити мову C++.

## Контрольні запитання

1. Що таке перевантаження функцій? Яку проблему вирішує ця можливість і як вона покращує читабельність коду?

2. Що входить до «сигнатури функції»? За якими саме критеріями компілятор відрізняє одну перевантажену функцію від іншої з таким самим іменем?

3. Чи можна перевантажити дві функції, якщо вони відрізняються лише типом поверненого значення (наприклад, int getValue() та double getValue())? Обґрунтуйте свою відповідь з точки зору логіки роботи компілятора.

4. Що таке значення аргументів за замовчуванням (default arguments)? Як їхнє використання може призвести до неоднозначності (ambiguity) під час виклику перевантажених функцій?

5. Для чого використовується перевантаження операторів в ООП? Яка

головна мета застосування стандартних математичних чи логічних символів (наприклад, +, ==) до об'єктів користувацьких класів?

6. Який синтаксис (яке ключове слово) використовується для визначення перевантаженого оператора?

7. Які фундаментальні правила не можна порушувати при перевантаженні операторів? Чи можна змінити пріоритет виконання оператора, його асоціативність або кількість операндів?

8. Назвіть оператори в C++, які категорично заборонено перевантажувати.

9. У чому полягає різниця між перевантаженням бінарного оператора (наприклад, +) як методу класу та як глобальної функції? Скільки аргументів приймає функція в кожному з цих випадків?

10. Що таке «дружні функції» (friend functions) і яку роль вони відіграють при перевантаженні операторів? Навіщо глобальній функції давати прямий доступ до приватних полів класу?

11. Чому оператори потокового введення та виведення (<< та >>) традиційно перевантажуються саме як глобальні (часто дружні) функції, а не як методи вашого класу?

12. Як технічно реалізується перевантаження операторів інкременту та декременту (++, --)? Як компілятор розрізняє префіксну (++obj) та постфіксну (obj++) форми запису на рівні сигнатури методу?

## Лекція №13. Шаблони та STL

### Концепція шаблонів

Концепція шаблонів може бути використана в двох видах: по відношенню до функцій і по відношенню до класів.

Шаблони дозволяють використовувати одні й ті самі функції або класи для обробки даних різних типів.

### Шаблон простої функції

Перевантажені функції зазвичай використовують для виконання специфічних операцій над різними типами даних. Якщо ж для кожного типу даних мають виконуватися ідентичні операції, то більш компактним і зручним прийомом є використання **шаблонів функцій** [6].

Щоб зрозуміти концепцію шаблонів C++, розглянемо наступну ситуацію. Припустимо, потрібно написати функцію обчислення модуля числа. Скоріш за все, функція обчислення модуля буде використовуватися з якимось одним типом даних.

Наприклад:

```
int abs(int x)                  // обчислення модуля цілих чисел
  {
  if (x<0) return -x;           // якщо число від'ємне, повернути -x
     else return x;
  }
```

Визначена вище функція бере аргумент типу int і повертає результат того самого типу.

Тепер уявіть, що знадобилося знайти модуль числа типу float. Доведеться писати ще одну функцію:

```
float abs(float x)              // обчислення модуля дійсних чисел
  {
  if (x<0) return -x;
     else return x;
  }
```

І ще одну для обчислення модуля числа типу double:

```
double abs(double x)            // обчислення модуля чисел типу double
  {
  if (x<0) return -x;
      else return x;
  }
```

Тіло функцій у наведених прикладах нічим не відрізняється. І все ж таки ці функції абсолютно різні, оскільки обробляють аргументи і повертають значення різних типів. Вони можуть бути перевантажені та мати однакові імена, це правда, але все одно для кожної з них потрібно писати окреме визначення.

Багаторазове переписування цих функцій-близнюків займає час, ускладнює читабельність коду, зате робить його громіздким. А якщо в алгоритмі виявиться помилка, доведеться виправляти її в тілі кожної функції.

Було б добре написати функцію всього один раз і примусити її працювати з різними типами даних, повертаючи, відповідно, результати різного типу. Тобто відокремити алгоритм реалізації від конкретного типу даних. Це якраз те, для чого в C++ існують шаблони функцій.

### Синтаксис шаблона функції

Ключовою ідеєю концепції шаблонів функції є представлення типу даних, що використовується функцією, не у вигляді якогось конкретного специфічного типу (наприклад, int), а за допомогою **назви**, замість якої може бути підставлений **будь-який тип**.

Шаблон для функцій обчислення модуля, розглянутих вище, може бути записаний наступним чином:

```
template <class Type>           // Шаблон функції обчислення модуля
Type abs(Type x)
  {
  if (x<0) return -x;
      else return x;
  }
```

Всю цю конструкцію називають **шаблоном функції**. Шаблон функції складається з двох частин – заголовку шаблона і звичайного визначення функції, в якому замість типу значення, що повертається і/або типу параметрів, записується ім'я типу, вказане у заголовку шаблона.

У наведеному синтаксисі назвою типу, що *параметризується*, є Type. Проте ніякого сакрального змісту саме в такій назві немає. На її місці може бути будь-яка назва, наприклад T, anyType чи YooHoo.

Ключове слово **template** повідомляє компілятору про те, що ми визначаємо шаблон функції.

Ключове слово **class** (або **typename**), укладене в кутові дужки, виступає синонімом слова «тип» і фактично означає «будь-який вбудований тип чи тип, визначений користувачем». Як ми вже бачили, використовуючи класи, можна визначати власні типи, тому різниці між типами і класами в даному розумінні немає.

Змінну, яка слідує за словом class (Type в нашому прикладі), називають **аргументом шаблона**. Всередині визначення шаблона аргумент Type замінює будь-який конкретний тип даних. В функції abs() це ім'я зустрічається лише двічі у першому рядку: в якості типу аргумента і одночасно в якості типу функції. У більш складному випадку воно могло б зустрітися багато разів, в тому числі в тілі функції.

Загальний синтаксис шаблона функції наступний:

```
template <class параметри_шаблона>
заголовок_функції
   {
      // тіло функції
   }
```

Конкретні описи функцій формуються по тим викликам, які компілятор виявляє в програмі.

***Приклад 13.1.*** Розглянемо визначення шаблонної функції обчислення модуля числа. Шаблон працює з базовими числовими типами int, long, double.

```
// TEMPABS, шаблон функції обчислення модуля числа
#include <iostream>
using namespace std;
template <class Type>              // Шаблон функції
Type abs(Type x)
   {
   if (x<0) return -x;
        else return x;
   }
int main()
{
   int int1 = 5;
   int int2 = -6;
   long lon1 = 70000L;
   long lon2 = -80000L;
   double dub1 = 9.95;
   double dub2 = -10.15;
// здійснення викликів
   cout << "\n abs(" << int1 << ") =" << abs(int1);          // виклик abs(int)
   cout << "\n abs(" << int2 << ") =" << abs(int2);          // виклик abs(int)
   cout << "\n abs(" << lon1 << ") =" << abs(lon1);          // виклик abs(long)
   cout << "\n abs(" << lon2 << ") =" << abs(lon2);          // виклик abs(long)
   cout << "\n abs(" << dub1 << ") =" << abs(dub1);          // виклик abs(double)
   cout << "\n abs(" << dub2 << ") =" << abs(dub2);          // виклик abs(double)
   cout << endl;
   return 0;  }
```

Результат виконання програми:

```
abs(5)=5
abs(-6)=6
abs(70000)=70000
abs(-80000)=80000
abs(9.95)=9.95
abs(-10.15)=10.15
```

Отже, функція abs() може працювати з трьома типами даних (int, long, double). Типи визначаються функцією під час передачі аргумента. Вона буде працювати й з іншими базовими числовими типами, у тому числі й з користувацькими (за умови належного перевантаження оператора «менше» (<) та унарного оператора «-»).

Важливо розуміти, що сам по собі шаблон не викликає генерацію компілятором будь-якого коду. Та й не може він цього зробити, оскільки ще не знає, з яким типом даних буде працювати ця функція. Він хіба що запам’ятовує шаблон для майбутнього використання.

Генерація коду не відбувається до тих пір, поки функція не буде реально викликана в ході виконання програми:

```
cout << "\n abs(" << int1 << ") =" << abs(int1);          // виклик abs(int)
```

Коли компілятор бачить такий виклик функції, він знає, що потрібно використовувати тип int, оскільки це – тип аргумента int1, переданого в шаблон. Тому компілятор генерує версію abs(int), що працює з типом int, та її виклик. Це називають **реалізацією шаблона функції**; при цьому кожний реалізований шаблон функції називають **шаблонною функцією**.

Зверніть увагу! Компілятор приймає рішення про те, як саме компілювати функцію, ґрунтуючись лише на типі даних аргумента (чи аргументів), використаного в шаблоні. Тип даних, що повертається функцією, не грає при цьому ролі. Це трохи схоже на те, як компілятор приймає рішення, яку з перевантажених функцій викликати.

Просту і геніальну ідею використання шаблонів функції схематично зображено на рис.1:

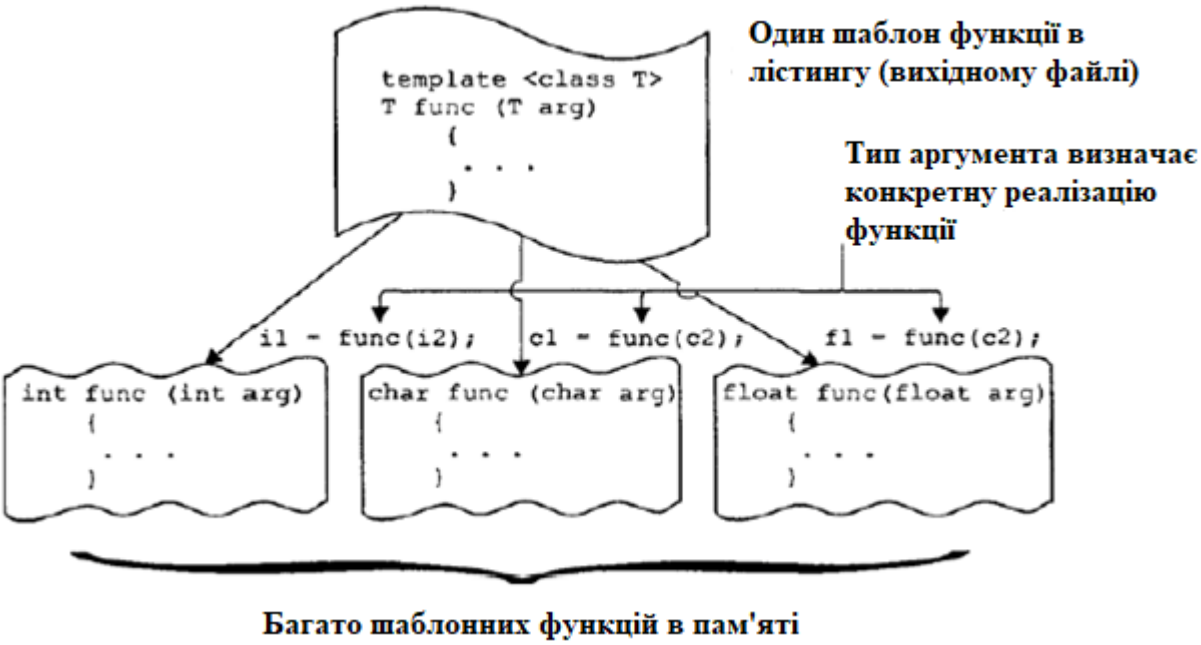



Рис. 13.1. Шаблон функції

## Шаблон функції з кількома аргументами

В якості наступного прикладу розглянемо шаблон функції з трьома аргументами.

***Приклад 13.2.*** Функція призначена для пошуку в числовому масиві заданого числа. Вона повертає індекс знайденого значення або -1 у випадку його відсутності в масиві.

```
// TEMPFIND, шаблон функції пошуку в масиві чисел
#include <iostream>
using namespace std;
template <class atype>                    // шаблон функції
int find(atype* array, atype value, int size)
{
   for(int j=0; j<size; j++)
   if(array[j]==value)
   return j;
   return -1;
}
int main()
```

```
{
   char chrArr[] = {1, 3, 5, 9, 11, 13};              // масив типу char
   char ch = 5;                                       // шукане значення
   int intArr[] = {1, 3, 5, 9, 11, 13};               // масив типу int
   int in = 6;                                        // шукане значення
   long lonArr[] = {1L, 3L, 5L, 9L, 11L, 13L};        // масив типу long
   long lo = 11L;                                     // шукане значення
   double dubArr[] = {1.0, 3.0, 5.0, 9.0, 11.0, 13.0};       // масив типу double
   double db = 4.0;                                   // шукане значення
   cout << "\n 5 in chrArray: i=" << find(chrArr, ch, 6);
   cout << "\n 6 in intArray: i=" << find(intArr, in, 6);
   cout << "\n11 in lonArray: i=" << find(lonArr, lo, 6);
   cout << "\n 4 in dubArray: i=" << find(dubArr, db, 6);
   cout << endl;
   return 0;
}
```

Тут ми аргумент шаблона називаємо ім’ям atype. Він з’являється двічі в аргументах функції: як тип покажчика на масив і як тип шуканого значення. Третій аргумент – розмір масиву – завжди типу int. Розмір масиву – не шаблонний аргумент.

В main() ми визначаємо чотири різних масиви різних типів і чотири значення, які потрібно знайти. Тип char в даному прикладі сприймається як число. Компілятор генерирує чотири різні варіанти функції – для кожного типу масиву викликається своя шаблонна функція.

Результат роботи програми виглядає наступним чином:

```
 5 in chrArray: i=2
 6 in intArray: i=-1
11 in lonArray: i=4
 4 in dubArray: i=-1
```

Зверніть увагу! Аргументи шаблона мають бути узгоджені. Тобто під час виклику шаблонної функції всі екземпляри даного аргумента шаблона мають бути строго одного типу.

Наприклад, якщо в функції find() є масив типу int, то, відповідно, шукане значення має бути того ж типу. НЕ можна робити так:

```
int intarray[]={1, 3, 5, 7};          // масив типу int
float fl = 5.0;                       // шукане значення типу float
int value = find(intarray, f1, 4);    // помилка!
```

Компілятору потрібно, щоб всі екземпляри atype були одного типу. Він може згенерувати код функції

```
find(int*, int, int);
```

проте видасть повідомлення про помилку у випадку:

```
find(int*, float, int);
```

тому що перший та другий аргументи мають бути одного типу.

**Різні аргументи одного шаблона**

У шаблоні функції можна використовувати кілька шаблонних аргументів [3].

Наприклад, не відомо, до масивів якого розміру потім потрібно буде застосовувати шаблон функції find(). Якщо масив занадто великий, може знадобитися тип long для зберігання його розміру. З іншого боку, якось не хочеться за замовчуванням застосовувати long, тому що він може й не знадобитися. Хотілося б обрати як тип розміру масиву, як і тип збережених в ньому даних прямо під час виклику функції. Для цього можна зробити розмір масиву ще одним аргументом шаблону. Назвемо його btype:

```
template <class atype, class btype>
btype find(atype* array, atype value, btype size)
{
   for(btype j=0; j<size; j++)          // використання btype
      if(array[j]==value)
         return j;
   return static_cast<btype>(-1);
```

}

Тепер можна використовувати значення як типу int, так і long (і навіть будь-якого іншого користувацького) в якості розміру масиву. Компілятор при своїй роботі тепер буде орієнтуватися не тільки на різні типи самого масиву і шуканого числа, а й на різні типи значень його розміру.

Наступний приклад демонструє використання шаблонної функції з двома шаблонними аргументами:

```
#include <iostream>
using namespace std;
template <class Type1, class Type2>
void myfunc(Type1 x, Type2 y)
{
cout << x << "\t" << y << "\n"; }
int main()
{
   myfunc(10, "Hi");
   myfunc(0.23, 10L);
   return 0;
}
```

У цьому прикладі при виконанні функції main(), коли компілятор генерує конкретні екземпляри шаблонної функції myfunc(), шаблонні аргументи type1 та type2 замінюються спочатку парою типів даних int та char *, а потім парою типів double та long відповідно.

Результат роботи програми виглядає наступним чином:

```
10      Hi
0.23    10
```

Зверніть увагу, що кількість примірників шаблонної функції в пам’яті різко зростає із зростанням числа аргументів шаблона. Вже два різних аргументи у поєднанні з шістьма базовими типами призводять до створення в пам’яті 36 копій шаблонної функції. Функції бувають досить великими, і можуть виникнути

проблеми з тим, що програмі потрібно занадто багато пам’яті. З іншого боку, не завжди реально викликаються шаблонні функції, незважаючи на наявність їх шаблонів.

### Концепція шаблонів класів

Шаблони класів надають можливість відокремити алгоритм реалізації функції від конкретних типів даних, з якими він працює, шляхом передачі типу в якості аргумента, дозволяючи створювати **параметризовані класи**.

Параметризований клас створює сімейство споріднених класів, які можна застосовувати до будь-якого типу даних, який передається в якості параметра.

Тобто шаблони класів, так само як і шаблони функцій, підтримують в C++ парадигму **узагальненого програмування**, тобто програмування з використанням типів в якості параметрів.

Механізм шаблонів в C++ допускає застосування абстрактного типу в якості параметра при визначенні класу. Після того як шаблон класу визначено, він може виконувати інкапсульовану задачу для будь-якого типу даних. Компілятор автоматично згенерує коректний екземпляр класу на основі типу, який буде заданий при створенні об’єкта.

Процес генерації компілятором визначення конкретного класу за шаблоном класу і аргументами шаблона називають **інстанціюванням шаблону** (template instantiation).

### Визначення шаблона класу

Загальний формат визначення шаблонного (узагальненого, родового) класу має наступний вигляд:

```
template < class параметри_шаблона> class ім’я_класу
  {
    // тіло класу
  };
```

Так само як і шаблон функції, шаблон класу складається з двох частин – власне заголовку шаблона і звичайного визначення класу, в якому замість типу

членів класу і/або типу параметрів методів класу записується ім’я типу, вказане у заголовку шаблона.

Наприклад:

```
template <class T> class MyClass {
/* …. */
};
```

Префікс template <class T> вказує, що оголошений шаблон класу, в якому T – деякий абстрактний тип. Тобто ключове слово class в цьому контексті означає «тип», а T – це параметр шаблона, тобто «замінник» для ім’я типу, який буде заданий при реалізації класу. Замість T можна використати будь-яке ім’я. Після оголошення T використовують всередині шаблона так само, як імена інших типів.

Зауважимо, що мова C++ дозволяє замість ключового слова class перед параметром шаблона використовувати інше ключове слово – typename, тобто записати:

```
template <typename T> class MyClass {
/* ... */
};
```

**Використання шаблона класу**

При включенні шаблона класу в програму ніякі класи насправді не генеруються до тих пір, поки не буде створено екземпляр шаблонного класу, в якому замість абстрактного типу T вказується певний конкретний тип. Така підстановка приводить до актуалізації шаблону.

Як й для звичайного класу, екземпляр шаблонного класу створюється через оголошення об’єкта. Загальний формат оголошення наступний:

```
ім’я_класу <тип> ім’я_об’єкта
```

Тут елемент <тип> означає ім’я типу даних, який буде оброблятися (використовуватися) екземпляром шаблонного класу. Зустрівши подібне оголошення, компілятор генерує код відповідного класу.

Наприклад:

Point <int> anyPoint(13, -5);

Інший спосіб створення екземпляру шаблонного класу – через оголошення покажчика на актуалізований шаблонний тип з присвоюванням йому адреси, що повертається операцією new.

Наприклад:

Point <double>* p OtherPoint = new Point <double>(9.99, 3.33);

Розглянемо простий приклад використання шаблонного класу:

```
#include <iostream>
using namespace std;
template <class T> class MyClass {              // Шаблон класу
private:
  T x, y;
public:
  MyClass(T a, T b)
  {
    x = a;
    y = b;
  }
  T div() { return x/y; }
};
int main()
{
  MyClass<double> d_ob(10.0, 3.0);              // версія MyClass для типу double
  cout << "double division: " << d_ob.div() << "\n";
  MyClass<int> i_ob(10, 3);                     // версія MyClass для типу int
  cout << "integer division: " << i_ob.div() << "\n";
   return 0;
}
```

Результат роботи програми виглядає наступним чином:

```
double division: 3.33333
integer division: 3
```

Як видно з результатів, об’єкт типу double виконав ділення чисел з плаваючою крапкою, а об’єкт типу int – цілочисельне ділення.

Після оголошення конкретного екземпляра класу MyClass компілятор автоматично генерує всі версії функції div() і змінних x та y, необхідні для обробки реальних даних. В даному прикладі оголошуються два об’єкти різного типу. Перший, d_ob, обробляє дані типу double. Це означає, що змінні x та y, а також результат ділення (значення, що повертається функцією div()) мають тип double. Другий об’єкт, i_ob, обробляє цілочисельні дані. Таким чином, змінні x та y, а також значення, що повертається функцією div(), мають тип int.

Зверніть увагу на оголошення об’єктів:

```
MyClass<double> d_ob(10.0, 3.0);
MyClass<int> i_ob(10, 3);
```

Тип даних, необхідний для реалізації екземпляра класу береться у кутові дужки. Змінюючи тип даних при створенні об’єктів класу MyClass, можна змінити тип даних, які обробляються цим класом.

Зверніть увагу! Функції-члени шаблонного класу автоматично стають шаблонними. Тому вже не потрібно використовувати ключове слово **template** для явного визначення їх як таких.

### Визначення методів поза шаблоном класу

Визначення вбудованих методів усередині шаблону класу практично не відрізняється від запису у звичайному класі. Проте якщо визначення методу виноситься за межі класу, то синтаксис його заголовку ускладнюється.

Покажемо це на прикладі шаблонного класу Point, який інкапсулює точку на площині:

```
// MAIN, зовнішнє визначення методів класу
#include <iostream>
using namespace std;
template <class T> class Point {
  private:
    T x, y;
```

```
  public:
    Point (T _x=0, T _y=0) : x(_x), y(_y) { }
    void Show() const;
};
// зовнішнє визначення  методу Show()
template <class T> void Point <T> :: Show() const
{
    cout << "("<<x<<", "<<y<<")"<<endl;
}
int main()
{
    Point <double> p1;                  // 1
    Point <double> p2(7.32, -2.6);      // 2
    p1.Show(); p2.Show();
    Point <int> p3(13, 15);             // 3
    Point <short> p4(17, 21);           // 4
    p3.Show(); p4.Show();
    return 0;
    }
```

У зовнішньому визначенні функції Show() зверніть увагу на появу того ж префікса template <class T>, який передував оголошенню шаблона класу. А також на більш складний запис операції глобального дозволу (області видимості) для цієї функції: якщо раніше ми писали Point ::, то тепер пишемо Point <T> ::, додаючи до імені класу список параметрів шаблона, укладений в кутові дужки (в даному випадку це один параметр T).

Результат вмконання програми:

```
(0, 0)
(7.32, -2.6)
(13, 15)
(17, 21)
```

Як видно з результатів, в рядках 1 і 2 шаблон інстанціюється в конкретний клас Point з підстановкою замість T типу double, в рядку 3 – у клас Point з

підстановкою типу int, в рядку 4 – з підстановкою типу short.

### Шаблон класу з кількома аргументами

За необхідності можна визначити кілька аргументів шаблона, використовуючи список елементів, розділених комами.

В якості наступного прикладу розглянемо шаблон класу з двома аргументами.

```
// шаблон класу з двома аргументами
#include <iostream>
using namespace std;
template <class T1, class T2> class MyClass {
   private:
     T1 i;
     T2 j;
   public:
     MyClass(T1 a, T2 b) { i = a; j = b; }
     void show() { cout << i << '\t ' << j << '\n'; }
};
int main()
{
     MyClass<int, double> ob1(10, 0.23);
     MyClass<char, char *> ob2('X', "This is a test");
     ob1.show();     // виведення int, double
     ob2.show();     // відображення char, char *
     return 0;
   }
```

Результат виконання програми:

```
10      0.23
X       This is a test
```

### Організація вихідного коду

У багатофайловому проекті всі визначення шаблонного класу прийнято розміщувати в деякому заголовному файлі *.h і підключати його до необхідних

файлів за допомогою директиви #include. Для запобігання повторного включення цього файлу, яке може мати місце у багатофайловому проекті, обов'язково використовують «вартових включення», які реалізуються за допомогою директиви #ifndef.

### Застосування шаблонів класів

Найбільш широке застосування шаблони знаходять при створенні *контейнерних класів.*

**Контейнерним** називають клас, який призначений для зберігання певним чином організованих даних і роботи з ними.

*Стеки* та *зв'язні списки* – типові приклади класів-сховищ даних. Стандартна бібліотека C++ надає великий набір шаблонів контейнерних класів для різних способів організації зберігання і обробки даних.

### Стандартна бібліотека шаблонів (STL)

Більшість комп'ютерів призначені для обробки інформації. В якості даних можуть виступати різні види характеристик реального світу: це може бути досьє на працівників, інформація про наявні на складі запаси, текстовий документ, результати наукових експериментів тощо. Якими б не були дані, зберігаються вони та обробляються приблизно однаковими способами.

В стандарт C++ входить власна вбудована бібліотека класів-контейнерів для різних способів організації, зберігання і обробки даних. Вона має назву Стандартна бібліотека шаблонів (Standard Template Library, STL) [5]. STL – це частина Стандартної бібліотеки C++.

У будь-якій програмі крім операторів мови використовують засоби бібліотек, включених у середовище програмування. Для використання засобів стандартної бібліотеки в програму за допомогою директиви #include потрібно включити відповідні файли заголовків. Наприклад, потоки описані у заголовку <iostream>, а списки – у заголовку <list>. Елементи заголовних файлів без розширення .h визначені у просторі імен std, а однойменні файли з розширенням .h – у глобальному просторі імен.

## Основні сутності STL

**STL** – це складний набір шаблонних класів і функцій, що реалізують популярні та часто вживані структури даних та алгоритми.

STL містить кілька основних сутностей. Три найбільш важливі з них – це *контейнери*, *алгоритми* та *ітератори*.

**Контейнер** – це шаблонний клас, що визначає спосіб організації зберігання даних. Іншими словами, це об'єкт, який може містити у собі інші об'єкти. Прикладами контейнерів є вектори, списки, черги або стеки. Оскільки контейнерні класи є шаблонними, то вони можуть зберігати об'єкти як стандартних вбудованих, так і визначених користувачем типів даних.

**Алгоритми** в STL – це методи, які дозволяють вирішувати типові задачі обробки даних. Алгоритми застосовують до контейнерів для обробки їхніх даних. Наприклад, є алгоритми сортування, копіювання, пошуку, об'єднання. Алгоритми представлені в STL у вигляді шаблонних функцій. Проте вони не є методами класів-контейнерів. Навпаки, це абсолютно незалежні глобальні функції. Насправді, однією з найбільш привабливих рис STL є універсальність її алгоритмів. Їх можна використовувати не лише до об'єктів різних класів-контейнерів, але й до звичайних масивів і навіть до власних контейнерів. (Контейнери, між тим, містять методи для виконання деяких специфічних завдань).

Реалізація механізму взаємодії між контейнерами (даними) та алгоритмами (обробкою) базується на використанні так званих *ітераторів*. **Ітератори** – це узагальнення концепції покажчиків: вони посилаються на елементи контейнера. Їх можна інкременувати, як звичайні покажчики, і вони будуть посилатися послідовно на всі елементи контейнера, а також розіменувати для отримання або зміни значення елемента. Ітератори – ключова частина всієї STL, оскільки вони зв'язують алгоритми з контейнерами. Їх можна представити собі у вигляді кабелю, що зв'язує колонки стереосистеми або комп'ютер з його периферією.

На рис. 13.2 представлена концепція побудови STL:

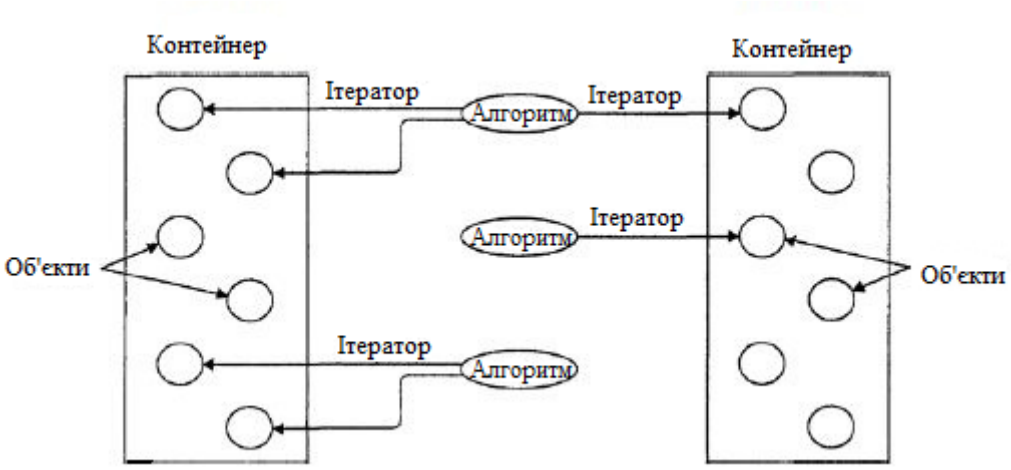



Рис. 13.2. Алгоритми використовують ітератори для роботи з об’єктами контейнерів

## Контейнери

Як ми вже говорили, контейнери представляють собою різні структури для зберігання даних. При цьому не має значення, які саме дані зберігаються: чи то деякі базові типи, такі, як int, float, або ж об’єкти класів.

Контейнери STL поділяють на дві категорії [3]: *послідовні* та *асоціативні*.

**Послідовні контейнери** забезпечують зберігання кінцевої кількості однотипних об’єктів у вигляді безперервної послідовності (лінійного списку).

До ***базових послідовних*** контейнерів відносять:

- вектори (vector)
- списки (list)
- черги з двостороннім доступом (deque)
- статичні неперервні масиви (array) (з C++11)
- однозв’язні списки (forward_list) (з C++11)

Є ще ***спеціалізовані*** контейнери (чи ***адаптери*** контейнерів), реалізовані на основі базових:

- стеки (stack)
- черги (queue)
- пріоритетні черги (priority_queue).

До **асоціативних** контейнерів відносять:

- множини (set)
- мультимножини (multiset)

- відображення/словники (map)
- мультивідображення/словники з дублікатами (multimap).

Кожен контейнер володіє своїми можливостями та атрибутами. Розглянемо пункти цієї класифікації більш детально.

**Послідовні контейнери**

У послідовних контейнерах дані зберігаються подібно до того, як будинки стоять на вулиці – в ряд. Кожен елемент зв'язується з іншими за допомогою номера своєї позиції в ряду. Всі елементи, крім кінцевих, мають по одному сусіду з кожного боку. Прикладом послідовного контейнера є динамічний масив – ***вектор (vector)***.

Проблема з масивами полягає у додаванні елемента у відсортований масив. Наприклад, у масиві зберігаються дані про працівників, і він відсортований за алфавітом відповідно до прізвищ. Тепер, якщо потрібно буде додати співробітника, прізвище якого починається з літери Л, то доведеться посунути всіх працівників з прізвищами на М – Я, щоб звільнити місце для вставки нового значення. Все це може займати досить багато часу. Для вирішення цієї проблеми в STL є контейнер ***список (list),*** який організує зберігання об'єктів у вигляді двозв'язного списку. Кожен елемент списку містить три поля: значення елемента, покажчик на попередній та покажчик на наступний елементи списку. Вставка і видалення ефективні для будь-якої позиції елемента у списку.

Ще одним представником послідовних контейнерів є ***черга з двостороннім доступом (deque).*** Це – комбінація стека та звичайної черги. Стек, як ви пам'ятаєте, працює за принципом LIFO (last in – first out, «останній зайшов – перший вийшов»). Тобто і введення, і виведення даних у стеку відбувається «зверху». (Принцип роботи стеку можна порівняти із пірамідкою, стопкою книг, тарілок). У черзі ж, навпаки, використовується принцип FIFO (first in – first out, «перший зайшов – перший вийшов»): дані надходять «зверху», а виходять «знизу». Черга з двостороннім доступом використовує комбінований порядок обміну даних: і вносити значення в чергу з двостороннім доступом, і виводити з неї дані можна з обох боків (рис. 13.2).

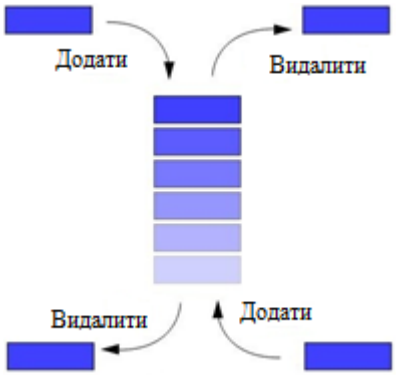



Рис. 13.2. Черга з двостороннім доступом

Кожен вид контейнера забезпечує свій набір дій над даними. Вибір виду контейнера залежить від того, що потрібно робити з даними в програмі. Наприклад, за необхідності часто вставляти і видаляти елементи з середини послідовності слід використовувати список, а якщо включення елементів виконується головним чином в кінець або початок – чергу з двостороннім доступом.

В табл. 13.1 наведено характеристики послідовних контейнерів STL.

Табл. 13.1. Характеристики послідовних контейнерів STL

| **Контейнер** | **Характеристика** | **Плюси/мінуси** |
|---|---|---|
| Звичайний масив | Фіксований розмір | Швидкісний випадковий доступ (по індексу)<br>Повільна вставка або вилучення даних з середини<br>Розмір не може змінюватись під час роботи |
| Вектор | Масив, що перерозподіляється та розширюється | Швидкісний випадковий доступ (по індексу)<br>Повільна вставка або вилучення даних з середини<br>Швидкісна вставка або вилучення даних з хвоста |
| Список | Аналогічно зв’язному списку | Швидкісна вставка або вилучення даних з будь-якого місця<br>Швидкий доступ до обох кінців<br>Повільний випадковий доступ |
| Черга з двостороннім доступ | Як вектор, але доступ з двох сторін | Швидкісний випадковий доступ |

## Реалізація контейнера STL

Для реалізації послідовних контейнерів STL в програму потрібно включити відповідний файл заголовку.

Наприклад:

```
# include <vector>        // вектор
# include <list>          // список
# include <deque>         // черга з двостороннім доступом
```

Передача інформації про те, які типи об’єктів будуть зберігатися у контейнері, здійснюється за допомогою аргумента шаблона.

Наприклад:

```
vector<int> aVect;              // створити вектор цілих чисел
vector<double> scores ;         // створити вектор дійсних чисел
list<Man> department;           // створити список типу Man
```

## Асоціативні контейнери

Асоціативний контейнер – це вже дещо інша організація даних. Дані тут розташовуються не послідовно. Ці контейнери побудовані на основі ***дерев***, що дозволяє здійснювати швидку вставку, видалення і пошук даних. Доступ до даних здійснюється за допомогою ***ключів*** – атрибутів, які використовують для впорядкування даних. Ключі – це просто номери рядків. Вони зазвичай використовуються для вибудовування елементів, що зберігаються, у певному порядку і модифікуються контейнерами автоматично. Прикладом асоціативного контейнера може служити звичайний орфографічний словник, де слова сортуються за алфавітом. Ви просто вводите потрібне слово (значення ключа), наприклад «кавун», а контейнер конвертує його в адресу цього елемента в пам’яті. Якщо відомий ключ, доступ до даних здійснюється дуже просто.

Одним з типів асоціативних контейнерів є ***множини (set)***. Наприклад, множина може зберігати об’єкти класу person, які упорядковані в алфавітному порядку. В якості ключа при цьому використовується значення name. При такій організації даних можна дуже швидко знайти потрібний об’єкт, здійснюючи пошук за ім’ям об’єкта (шукаємо об’єкт класу person за атрибутом name). Якщо

у множині зберігаються значення одного з базових типів, таких, як int, ключем є сам елемент.

У асоціативних контейнерах відображеннях і множинах даному значенню відповідає лише один ключ. Тобто елементи таких контейнерів є унікальними. Це має сенс у таких, наприклад, випадках, як зберігання списку працівників, де кожному імені зіставляється унікальний ідентифікаціонний номер.

В табл. 13.2 наведено характеристики асоціативних контейнерів STL.

Табл. 13.2. Характеристики асоціативних контейнерів STL

| **Контейнер** | **Характеристики** |
|---|---|
| Множина | Зберігає тільки ключові об'єкти. Кожному значенню відповідає один ключ |
| Мультимножина | Зберігає тільки ключові об'єкти. Одному значенню може відповідати декілька ключів |
| Відображення | Асоціює ключовий об'єкт з об'єктом, що зберігає значення (цільовим). Одному значенню відповідає один ключ |
| Мультивідображення | Асоціює ключовий об'єкт з об'єктом, що зберігає значення (цільовим). Одному значенню може відповідати декілька ключів |

### Реалізація асоціативних контейнерів STL

Для реалізації асоціативних контейнерів STL в програму потрібно включити відповідний файл заголовку.

Наприклад:

```
#include <map>    // для відображення та мультивідображення
#include <set>    // для множини та мультимножини
```

Створюються асоціативні контейнери приблизно так само, як і послідовні.

Наприклад:

```
set<int> intSet;    // створити множину значень типу int
```

### Адаптери контейнерів

Спеціалізовані контейнери можна створювати з базових (наведених вище) за допомогою конструкції, яку називають **адаптером контейнера**. Вони мають більш простий інтерфейс, ніж звичайні контейнери. Спеціалізовані контейнери, реалізовані в STL, – це *стеки*, *черги* та *пріоритетні черги*. Як уже зазначалося,

для ***стека*** характерний доступ до даних лише з одного кінця, його можна порівняти зі стопкою книг. ***Черга*** використовує для проштовхування даних один кінець, а для виштовхування – інший. У ***пріоритетній черзі*** дані проштовхуються спереду у довільному порядку, а виштовхуються у суворій відповідності до величини значення, що зберігається. Пріоритет мають дані з найбільшим значенням. Таким чином, пріоритетна черга автоматично сортує інформацію, яка в ній зберігається.

У табл. 13.3 наведено характеристики контейнерів, що реалізуються за допомогою адаптерів, і послідовні контейнери, що використовуються для їх реалізації.

Табл. 13.3. Характеристики контейнерів, що реалізуються за допомогою адаптерів, і послідовні контейнери, що використовуються для їх реалізації

| **Контейнер** | **Реалізація** | **Характеристики** |
|---|---|---|
| Стек | Реалізується як вектор, список або черга з двостороннім доступом | Проштовхування і виштовхування даних тільки з одного кінця |
| Черга | Реалізується як список або черга з двостороннім доступом | Проштовхування з одного кінця, виштовхування – з іншого |
| Пріоритетна черга | Реалізується як вектор або черга з двостороннім доступом | Проштовхування з одного кінця випадковим чином, виштовхування – впорядковане з іншого кінця |

## Реалізація адаптерів контейнерів STL

Для реалізації асоціативних контейнерів STL в програму потрібно включити відповідний файл заголовку.

Наприклад:

```
#include <stack>        // для стеку
#include <queue>        // для черги
```

Для практичного застосування цих класів необхідно використовувати як би шаблон в шаблоні.

Наприклад, нехай є об’єкт типу стек, що містить значення int, породжений класом «черга з двостороннім доступом» (deque):

stack< deque<int> > aStak;

Зверніть увагу! При описі цього формату необхідно ставити пробіли між двома кутовими дужками. Інакше компілятор інтерпретує >> як оператор.

## Алгоритми

**Алгоритми** – це функції, які виконують певні дії над елементами контейнера (контейнерів). Їхні можливості включають ініціалізацію, сортування, пошук, злиття, заміну і трансформацію вмісту контейнера. Багато алгоритмів оперують діапазонами елементів в контейнері.

Алгоритми були розроблені спеціально для контейнерів, але їхньою чудовою властивістю є те, що вони можуть бути застосовані й до звичайних масивів C++.

Як вже говорилося вище, алгоритми в STL не є методами класів і навіть не є дружніми функціями по відношенню до контейнерів. У нинішньому стандарті мови C++ алгоритми – це незалежні шаблонні функції. Їх можна використовувати при роботі як зі звичайними масивами C++, так і з користувацькими класами-контейнерами.

У таблиці 13.4 наведено деякі типові алгоритми STL.

Табл. 13.4. Деякі типові алгоритми STL

| Алгоритм | Призначення |
|---|---|
| find | Повертає перший елемент з вказаним значенням |
| count | Рахує кількість елементів, що мають вказане значення |
| equal | Порівнює вміст двох контейнерів і повертає true, якщо всі відповідні елементи еквівалентні |
| search | Шукає послідовність значень в одному контейнері, яка відповідає такій же послідовності в іншому |
| copy | Копіює послідовність значень з одного контейнера в інший (або в інше місце того ж контейнера) |
| swap | Обмінює значення, що зберігаються в різних місцях |
| iter_swap | Обмінює послідовність значень, що зберігаються в різних місцях |
| fill | Копіює значення в послідовність комірок |
| sort | Сортує значення у вказаному порядку |

| | |
|---|---|
| merge | Комбінує два відсортованих діапазони значень для отримання найбільшого діапазона |
| accumulate | Повертає суму елементів у вказаному діапазоні |
| for_each | Виконує вказану функцію для кожного елемента контейнера |

## Реалізація алгоритмів STL

Для реалізації алгоритмів STL в програму потрібно включити відповідний файл заголовку.

```
#include <algorithm>          // для алгоритмів
```

## Ітератори

**Ітератори** – є аналогом покажчика на елемент. Їх використовують для отримання доступу до окремих даних в контейнері, зокрема для послідовного просування контейнером від елемента до елемента (цей процес називають ***ітерацією***). Ітератори можна інкрементувати за допомогою звичайного оператора ++, після виконання якого ітератор пересунеться на наступний елемент. У STL ітератор є об’єктом класу **iterator**.

Для різних типів контейнерів використовують свої ітератори. Всього існує три основні класи ітераторів:

- прямі
- двоспрямовані
- з випадковим доступом.

***Прямий ітератор*** може проходити контейнером лише у прямому напрямку, що і зазначено в його назві. Прохід здійснюється поелементний. Працювати з ним можна, використовуючи інкремент ++. Такий ітератор не може пересуватися у зворотному напрямку і не може бути поставлений у довільне місце контейнера.

***Двоспрямований ітератор***, відповідно, може пересуватися в обох напрямках і реагує як на інкремент ++, так і на декремент --.

***Ітератор з випадковим доступом*** може рухатися в обох напрямках, а також перескакувати на довільний елемент контейнера. Можна наказати йому отримати доступ до позиції 27, наприклад.

Є два спеціалізованих види ітераторів:

- ***вхідний ітератор***, який може «вказувати» на пристрій введення (cin або навіть просто вхідний файл) та зчитувати послідовно елементи даних у контейнер;

- ***вихідний ітератор***, який, відповідно, вказує на пристрій виведення (cout) або вихідний файл і виводить елементи з контейнера.

У той час як значення прямих, двоспрямованих ітераторів та ітераторів з випадковим доступом можуть бути збережені, значення вхідних і вихідних ітераторів зберігатися не можуть. Це має сенс, бо перші три ітератори вказують на деяку адресу у пам'яті, а вхідні та вихідні ітератори вказують на пристрої введення/виведення, для яких зберігати «покажчик» неможливо.

У таблиці 13.5 наведено характеристики різних типів ітераторів STL.

Табл. 13.5. Характеристики різних типів ітераторів STL

| **Тип ітератора** | **Запис/Зчитування** | **Зберігання значення** | **Напрям** | **Доступ** |
|---|---|---|---|---|
| З випадковим доступом | Запис та зчитування | Можливо | Обидва напрями | Випадковий |
| Двонаправлений | Запис та зчитування | Можливо | Обидва напрями | Лінійний |
| Прямий | Запис та зчитування | Можливо | Тільки прямий | Лінійний |
| Вихідний | Тільки запис | Неможливо | Тільки прямий | Лінійний |
| Вхідний | Тільки зчитування | Неможливо | Тільки прямий | Лінійний |

Взагалі ітератор, що має більш широкі можливості доступу, може застосовуватися замість ітератора з меншими можливостями. Так, наприклад, прямий ітератор може бути використаний замість вхідного ітератора.

### Реалізація ітераторів STL

Для реалізації ітератора STL в програму потрібно включити відповідний файл заголовку.

```
#include <iterator>                    // для ітераторів
```

## Приклади програмної реалізації алгоритмів

**Приклад 13.3.** Алгоритм шукає перший елемент в контейнері, значення якого дорівнює вказаному.

```
// FIND, знайти перший об'єкт, значення якого дорівнює заданому
#include <iostream>
#include <algorithm>              // для find()
using namespace std;
int arr[] = { 11, 22, 33, 44, 55, 66, 77, 88 };
int main()
 {
 int* ptr;
 ptr = find(arr, arr+8, 33);      // знайти перше значення 33
 cout << "Перший об'єкт зі значенням 33 знайдений у позиції "
    << (ptr-arr) << endl;
 return 0;
 }
```

Результат виконання програми:

```
Перший об'єкт зі значенням 33 знайдений у позиції 2
```

***Приклад 13.4.*** Алгоритм підраховує, скільки елементів у контейнері мають задане значення.

```
// COUNT, підраховувати кількість об'єктів, що мають задане значення
#include <iostream>
#include <algorithm>              // для count()
using namespace std;
int arr[] = { 33, 22, 33, 44, 33, 55, 66, 77 };
int main()
{
int n = count(arr, arr+8, 33);    // рахує, скільки разів зустрічається 33
cout << "Число 33 зустрічається " << n << " рази в масиві." << endl;
```

```
return 0;
}
```

Результат виконання програми:

```
Число 33 зустрічається 3 рази в масиві
```

***Приклад 13.5.*** Алгоритм сортує елементи у контейнері

```
// SORT, сортування масиву цілих чисел
#include <iostream>
#include <algorithm>
using namespace std;
int arr[] = {45, 2, 22, -17, 0, -30, 25, 55};     // масив чисел
int main()
  {
  sort(arr, arr+8);            // сортування
for(int j=0; j<8; j++)         // виведення відсортованого масиву
    cout << arr[j] << ' ';
  cout << endl;
  return 0;
  }
```

Результат виконання програми:

```
-30  -17  0  2  22  25  45  55
```

В якості висновків зазначемо, що бібліотека STL мови C++ надає широкий набір шаблонів, що представляють контейнери, ітератори і алгоритми. **Контейнер** – це структура даних (подібна масиву), яка може зберігати кілька значень. Контейнери STL однорідні за структурою і зберігають тільки однотипні значення. **Алгоритми** використовують для виконання певних завдань, наприклад, сортування масиву або пошуку значення у списку. **Ітератори** – це об'єкти, що дозволяють переміщатися всередині контейнера подібно до того, як за допомогою покажчиків можна переміщатися масивом; вони є узагальненням ідеї покажчиків.

STL дозволяє створювати різні контейнери, що містять масиви, черги і списки, і здійснювати безліч операцій, включаючи пошук, сортування і тасування у випадковому порядку.

Використання контейнерів STL дозволяє значно підвищити надійність C++-програм, їх універсальність, а також зменшити час їх розробки.

## Контрольні запитання

1. Що таке узагальнене програмування (generic programming) і яку головну проблему вирішують шаблони (templates) у C++? Як це пов’язано з принципом уникнення дублювання коду (DRY – Don’t Repeat Yourself)?

2. Який синтаксис використовується для оголошення шаблону функції? Поясніть призначення ключових слів template та typename (або class) у кутових дужках.

3. Що таке «інстанціювання» (instantiation) шаблону? Хто і в який момент (під час компіляції чи під час виконання) генерує реальний машинний код функції для конкретного типу даних, наприклад, для int або std::string?

4. У чому полягає різниця між використанням шаблону функції та шаблону класу? Як саме потрібно вказувати типи даних під час створення об’єкта шаблонного класу (наприклад, MyClass<int> obj;) порівняно з викликом шаблонної функції?

5. Чи може шаблон приймати більше одного параметра типу? Наведіть приклад структури даних, де це логічно необхідно (наприклад, пара «ключ-значення»).

6. Що таке «спеціалізація шаблону» (template specialization)? Для чого розробнику може знадобитися написати окрему, унікальну реалізацію шаблонної функції чи класу спеціально для певного типу даних (наприклад, для вказівників char*)?

7. Що таке Standard Template Library (STL)? З яких трьох фундаментальних стовпів (компонентів) вона складається?

8. Як би ви визначили поняття «контейнер» у контексті STL? Які три основні категорії контейнерів існують у стандартній бібліотеці (послідовні, асоціативні, адаптери)?

9. Назвіть основні послідовні контейнери (sequence containers). У чому полягає ключова різниця у внутрішній організації пам’яті та швидкості базових операцій (вставка, пошук) між std::vector та std::list?

10. Чому std::vector вважається стандартом де-факто для більшості задач у C++? Які беззаперечні переваги він має над звичайними «сирими» динамічними масивами, створеними за допомогою оператора new[]?

11. Що таке асоціативні контейнери (associative containers)? Поясніть концепцію зберігання даних у std::map (словник) та std::set (множина). За яким принципом у них відбувається швидкий пошук елементів?

12. Яку роль відіграють ітератори при роботі з контейнерами STL? Чому стандартні алгоритми (наприклад, std::sort або std::find) спроєктовані так, щоб працювати саме через ітератори, а не через прямий доступ до методів конкретного контейнера?

## Лекція №14. Контейнери та ітератори

Хоча різні контейнери представляють собою різні структури для зберігання даних, всі вони розділяють **загальну функціональність STL** – відповідають набору загальних вимог незалежно від конкретної реалізації контейнера [7]. Таким чином, одні й ті самі загальні принципи можуть бути застосовні до будь-якого типу контейнера.

Базові прийоми використання контейнерів розглядатимемо на прикладі *послідовних контейнерів*. Продемонструємо як:

- створити послідовний контейнер;
- додати елементи в контейнер;
- визначити розмір контейнера;
- використовувати ітератор для виконання циклу контейнером;
- присвоїти один контейнер іншому;
- визначати еквівалентність одного контейнера іншому;
- видаляти елементи з контейнера;
- змінювати одні елементи в контейнері іншими;
- визначати ознаку порожнечі контейнера.

Мова піде про методи контейнерів.

### Вектори

Спочатку використаємо контейнерний **клас vector**, і в ньому ті методи, що є загальними для всіх послідовних контейнерів.

**Вектори** – це, так би мовити, розумні масиви. Вони займаються автоматичним розміщенням себе в пам’яті, розширенням і звуженням свого розміру в міру додавання або видалення даних. Вектори можна використовувати як масиви, звертаючись до елементів за допомогою звичного оператора індексації [ ]. Випадковий доступ до даних у векторах виконується дуже швидко. Також досить швидко здійснюється додавання (або *проштовхування*) нових даних у кінець вектора. Коли це відбувається, розмір вектора автоматично збільшується для того, щоб було куди помістити нове значення.

В класі vector визначено кілька типів **конструкторів**:

1) конструктор за замовчуванням – створює порожній контейнер.

Наприклад:

vector<int> a; // *порожній вектор, розмір якого = 0*

vector<int> a(5); // *порожній вектор розміром 5*

2) конструктор вектора з параметрами – створює контейнер з count-копіями елемента.

Наприклад:

vector<int> a(5, 8); // *8 8 8 8 8*

vector<int> a(5, '*'); // *42 42 42 42 42*

vector<char> a(5, '*'); // * * * * *

vector<string> words4(5, "Mo"); // *Mo Mo Mo Mo Mo*

3) конструктор копіювання. Створює контейнер-копію:

vector<int> b(a); // *b=a*

4) конструктор, що ініціалізує вектор діапазоном елементів, заданих парою ітераторів (iterator first та iterator last).

Наприклад:

vector a(arr, arr+4);

vector<string> words2(words1.begin(), words1.end());

В класі vector визначено близько двох десятків **функцій-членів** (методів), зокрема:

- Задати значення у контейнері можна за допомогою методу assign() та оператора присвоювання =.
- Доступ до окремих елементів контейнера здійснюється за допомогою методів та ітераторів:

front(), back(), at(), begin(), end(), rbegin(), rend() та оператора [].

- Зміна вмісту контейнера можлива за допомогою методів:

clear(), insert(), erase(), push_back(), pop_back(), swap().

- Місткість вектора визначається за допомогою методів:

empty(), size(), max_size(), capacity().

- Крім того, для вектора визначені операції відношень:

==, !=, <, >, <=, >=, які лексикографічно порівнюють значення у контейнері.

Зверніть увагу! Різні види контейнерів використовують методи з однаковими іменами та характеристиками.

У табл. 14.1 наведено методи класу vector, імена та призначення яких спільні для більшості класів контейнерів.

Таблиця 14.1. Методи класу vector

| Метод | В | Сп | Д/ч | Мн | М/м | Від | М/в | Ст | Ч | Пр/ч |
|---|---|---|---|---|---|---|---|---|---|---|
| **operator ==** | x | x | x | x | x | x | x | x | x | |
| **operator !=** | x | x | x | x | x | x | x | x | x | |
| **operator <** | x | x | x | x | x | x | x | x | x | |
| **operator >** | x | x | x | x | x | x | x | x | x | |
| **operator <=** | x | x | x | x | x | x | x | x | x | |
| **operator >=** | x | x | x | x | x | x | x | x | x | |
| **operator =** | x | x | x | | | | | | | |
| **operator []** | x | | x | | | x | | | | |
| **begin()** | x | x | x | x | x | x | x | | | |
| **end()** | x | x | x | x | x | x | x | | | |
| **rbegin()** | x | x | x | x | x | x | x | | | |
| **rend()** | x | x | x | x | x | x | x | | | |
| **empty()** | x | x | x | x | x | x | x | x | x | x |
| **size()** | x | x | x | x | x | x | x | x | x | x |
| **max_size()** | x | x | x | x | x | x | x | | | |
| **front()** | x | x | x | | | | | | x | |
| **back()** | x | x | x | | | | | | x | |
| **push_back()** | x | x | x | | | | | | | |
| **pop_back()** | x | x | x | | | | | | | |
| **swap()** | x | x | x | x | x | x | x | | | |
| **insert()** | x | x | x | x | x | x | x | | | |
| **erase()** | x | x | x | x | x | x | x | | | |
| **clear()** | x | x | x | x | x | x | x | | | |
| **capacity()** | x | | | | | | | | | |

**Огляд основних методів класу vector:**

**operator =** задає значення у контейнері.

Метод **assign()** задає значення у контейнері.

Метод **at()** надає доступ до вказаного елемента контейнера.

**operator []** надає доступ до вказаного елемента контейнера.

Метод **front()** надає доступ до першого елемента контейнера.

Метод **back()** надає доступ до останнього елемента контейнера.

Метод **begin()** повертає ітератор на перший елемент контейнера.

Метод **end()** повертає ітератор на елемент, що **слідує за** останнім елементом контейнера.

Метод **rbegin()** повертає зворотній ітератор на перший елемент контейнера (це останній елемент послідовності).

Метод **rend()** повертає зворотній ітератор на елемент, що **слідує за** останнім елементом контейнера (це елемент, що **передує** першому елементу послідовності).

Метод **empty()** перевіряє відсутність елементів у контейнері. Повертає true, якщо контейнер порожній.

Метод **size()** повертає кількість елементів в контейнері.

Метод **max_size ()** повертає максимальний розмір, до якого може розширитися контейнер. Це число залежить від типу даних, що зберігає контейнер (зрозуміло, що чим більше місця займає один елемент даного типу, тим менше елементів ми зможемо зберігати), типу контейнера та операційної системи.

Метод **capacity()** повертає кількість елементів, які можна розмістити у виділеній області пам’яті.

Метод **clear()** видаляє всі елементи з контейнера.

Метод **insert()** додає елемент у вказану позицію у контейнері.

Метод **erase()** видаляє вказаний елемент з контейнера.

Метод **push_back()** додає елемент у кінець контейнера.

Метод **pop_back()** видаляє останній елемент з контейнера.

Метод **swap()** обмінює дані одного контейнер на дані іншого, при цьому порядок слідування елементів не змінюється.

**Методи push_back(), size() та операція доступу за індексом [ ]**

Зупинимося докладніше на використанні найбільш загальних векторних операцій.

Наприклад:

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  vector<int> v;            // створити вектор типу int
  v.push_back(10);          // внести елемент в кінець вектора
  v.push_back(11);
  v.push_back(12);
  v.push_back(13);
  v[0] = 20;                // замінити новими значеннями
  v[3] = 23;
  for(int i=0; i<v.size(); i++)    // вивести вміст вектора
    cout << v[i] << " ";
  cout << endl;
  return 0;
}
```

Результат виконання програми:

```
20 11 12 23
```

Для створення вектора v використовується штатний конструктор вектора без параметрів. Як із будь-якими контейнерами STL, для визначення типу

змінних, що будуть зберігатися у векторі, використовується шаблонний формат (у даному випадку це тип int).

Ми не визначаємо розмір контейнера, тому спочатку він дорівнює 0.

Метод **push_back ()** вставляє значення свого аргумента в кінець вектора (кінець розташовується там, де знаходиться найбільший індекс).

Початок вектора (елемент з індексом 0), на відміну від списків і черг, не може використовуватися для вставки нових елементів.

У наведеному прикладі значення 10, 11, 12 і 13 проштовхуються таким чином, що v[0] містить 10, v[1] містить 11, v[2] містить 12, а v[3] містить 13.

Як тільки у векторі з'являються які-небудь дані, до них відразу можна отримати доступ за допомогою перевантаженого оператора індексації []. Цей оператор використаний у прикладі для заміни значення першого елемента з 10 на 20, а останнього – з 13 на 23.

Метод **size ()** повертає поточне число елементів, що містяться в контейнері.

Для наведеного прикладу число елементів – 4. Це значення використане у циклі for для виведення значень вектора на екран.

**Методи swap(), empty(), back() та pop_back()**

Наступний приклад демонструє ще кілька корисних методів для векторів [2].

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  double arr[] = { 1.1, 2.2, 3.3, 4.4 }; // масив типу double
  vector<double> v1(arr, arr+4);       // ініціалізація вектора масивом
  vector<double> v2(4);                // створити порожній вектор розміром 4
  v2 = { 0.1, 0.2, 0.3, 0.4 };         // ініціалізація вектора
  v1.swap(v2);                         // обміняти вміст векторів v1 та v2
  while( !v2.empty() )                 // поки вектор не буде порожній
```

```
    {
    cout << v2.back() << " ";          // вивести останній елемент
    v2.pop_back();                     // видалити останній елемент
    }
  cout << endl;
  for(int i=0; i<v1.size(); i++)       // вивести вміст вектора
    cout << v1[i] << " ";
  cout << endl;
  return 0;
}
```

Результат виконання програми:

```
4.4 3.3 2.2 1.1
0.1 0.2 0.3 0.4
```

У наведеному прикладі ми використали два нових конструктори векторів. Перший конструктор ініціалізує вектор v1 значеннями звичайного масиву C++, що переданий йому в якості аргумента. Аргументами цього конструктора є покажчики на перший елемент масиву і на елемент «після останнього».

У другому конструкторі вектор v2 «ініціалізується» установкою його розміру (у даному випадку це 4), але значення самого вектора для ініціалізації не передаються. Це робиться окремою інструкцією.

Метод **swap ()** обмінює дані одного вектора на дані іншого, при цьому порядок слідування елементів не змінюється.

Метод **back ()** повертає значення останнього елемента вектора.

Метод **pop_back ()** видаляє останній елемент вектора.

Таким чином, при кожному проходженні циклу while останній елемент вектора v2 буде мати різні значення. (Трохи дивно, що pop_back () тільки видаляє останній елемент, але не повертає його значення, як pop () при роботі зі стеком. Тому, в принципі, завжди потрібно використовувати методи pop_back () та back () у парі).

На рис. 14.1 проілюстровано роботу розглянутих методів для послідовного контейнера Вектор (vector):

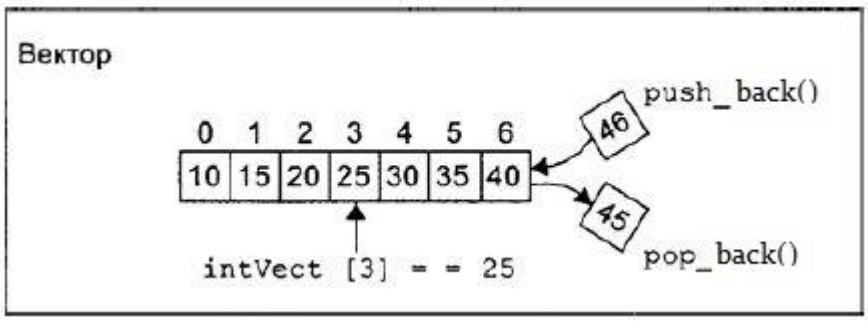



Рис. 14.1. Ілюстрація роботи методів послідовного контейнера vector

### Методи insert() та erase()

Не дивлячись на те, що методи вставки та видалення елемента з довільної позиції не дуже ефективно використовувати з векторами (оскільки в цьому випадку при кожній вставці або видаленні доводиться перекроювати всю структуру – розтискати, стискати вектор), ці методи визначені для контейнера vector. Тож використовувати їх з векторами і можна, і треба.

Наступний приклад показує, як це робити.

```
// вставка та видалення елементів з довільної позиції вектора
#include <iostream>
#include <vector>
using namespace std;
int main()
{
  int arr[] = { 10, 11, 12, 13 };          // масив типу int
  vector<int> v(arr, arr+4);               // ініціалізація вектора масивом
  cout << "\nПеред вставкою: ";
  for(int j=0; j<v.size(); j++)            // вивести всі елементи
    cout << v[j] << ' ';
  v.insert( v.begin()+2, 15);              // вставити 15 в позицію 2
  cout << "\nПісля вставки:  ";
  for(int j=0; j<v.size(); j++)            // вивести всі елементи
    cout << v[j] << ' ';
  v.erase( v.begin()+3 );                  // видалити елемент з позиції 3
```

```
  cout << "\nПісля видалення: ";
  for(int j=0; j<v.size(); j++)          // вивести всі елементи
    cout << v[j] << ' ';
  cout << endl;
  return 0;
  }
```

Результат роботи програми виглядає наступним чином:

```
Перед вставкою:  10 11 12 13
Після вставки:   10 11 15 12 13
Після видалення: 10 11 15 13
```

Метод **insert()** вставляє елемент у вказану позицію. Всі елементи від точки вставки до кінця контейнера зсуваються, щоб було місце для розміщення елемента, що вставляється. Розмір контейнера автоматично збільшується на одиницю.

Метод **erase()** видаляє елемент з вказаної позиції. Решта елементів зсуваються, розмір контейнера зменшується на одиницю.

### Визначення еквівалентності одного контейнера іншому

Визначення еквівалентності (ідентичності) одного контейнера іншому розглянемо на прикладі двох контейнерів vector.

```
// Визначення еквівалентності (ідентичності) двох векторів
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<char> v1;                  // порожній вектор типу char
   v1 = {'A', 'B', 'C', 'D', 'E', 'F'};
   vector<char> v2(v1);              // створити ідентичний вектор
   cout << " v1: ";
   for(int i=0; i<v1.size(); i++)    // вивести всі елементи v
```

```
    cout << v1[i] << ' ';
  cout << "\n v2: ";
  for(int i=0; i<v2.size(); i++)          // вивести всі елементи v2
    cout << v2[i] << ' ';
  if (v1 == v2) cout << "\n v1 та v2 еквівалентні" << endl;
  cout << endl;
  v1.pop_back();                          // видалити останній елемент v1
  v2.erase( v2.begin() );                 // видалити перший елемент v2
  v1.insert(v1.begin(), 'Z');             // додати Z на початок v1
  v2.push_back('X');                      // додати X в кінець v2
  cout << " v1: ";
  for(int i=0; i<v1.size(); i++)          // вивести всі елементи v
    cout << v1[i] << ' ';
  cout << "\n v2: ";
  for(int i=0; i<v2.size(); i++)          // вивести всі елементи v2
    cout << v2[i] << ' ';
    if(v1 < v2) cout << " \n v1 менший, ніж v2" << endl;
    if(v1 > v2) cout << " \n v1 більший, ніж v2" << endl;
  return 0;
}
```

Результат виконання програми:

```
v1: A B C D E F
v2: A B C D E F
v1 та v2 еквівалентні

v1: Z A B C D E
v2: B C D E F X
v1 більший, ніж v2
```

Як бачимо, **два контейнери порівнюються операціями** == та **<, >.** Для послідовних контейнерів порівняння елементів здійснюється в *лексикографічному порядку*. **Два контейнери вважаються еквівалентними**, якщо вони містять однакову кількість елементів, в однаковому порядку, і всі

відповідні елементи в двох контейнерах еквівалентні. В іншому випадку результат лексикографічного порівняння визначається порівнянням перших, не співпадаючих елементів.

**Методи front(), back(), max_size() та capacity()**

Застосування ще кількох базових операцій з послідовним контейнером vector демонструє наступний приклад.

```
#include <iostream>
#include <vector>
using namespace std;
int main()
{
   vector<char> letters {‘d’, ‘m’, ‘g’, ‘w’, ‘t’, ‘f’};
   if (!letters.empty())
   {
      cout << " Перший елемент: " << letters.front() << endl;
      cout << " Останній елемент: " << letters.back() <<endl ;
   }
   cout << " Кількість елементів в контейнері: " << letters.size() << endl;
   cout << " Максимальна кількість елементів в контейнері: "
           << letters.max_size() << endl;
   cout << " Доступна кількість елементів в контейнері: "
           << letters.capacity() << endl;
    return 0;
}
```

Результат виконання програми:

```
Перший елемент: d
Останній елемент: f
Кількість елементів в контейнері: 6
Максимальна кількість елементів в контейнері: 18446744073709551615
Доступна кількість елементів в контейнері: 6
```

### Сортування елементів контейнера

Наступна програма демонструє роботу алгоритму сортування елементів контейнера vector.

```
// сортування елементів контейнера vector
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
   vector<int> v;                    // порожній вектор
   v={7, 8, 2, 3, 27, 5, 1, 4, 10, 31};
   sort(v.begin(), v.end());
   for(int i=0; i<v.size(); i++)     // вивести всі елементи
      cout << v[i] << ' ';
   return 0;
}
```

Результат виконання програми:

```
1 2 3 4 5 7 8 10 27 31
```

Ще раз звертаємо увагу: у наведених прикладах використаний контейнер вектор, однак ті самі прийоми можуть бути застосовані до будь-якого послідовного контейнера.

### Списки

Контейнер STL під назвою **список** являє собою двозв'язаний список, в якому кожен елемент зберігає покажчик на сусіда зліва і справа [1, 3]. Контейнер містить адресу першого і останнього елементів, тому доступ до обох кінців списку здійснюється дуже швидко.

### Методи push_front(), front() та pop_front

Наступний приклад демонструє вставку, зчитування та видалення

даних зі списку.

```
// вставка, зчитування та видалення даних зі списку
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<int> ilist;                  // порожній список цілочисельних елементів
  ilist.push_back(30);              // вставка елементів у кінець
  ilist.push_back(40);
  ilist.push_front(20);             // вставка елементів на початок
  ilist.push_front(10);
  int size = ilist.size();           // число елементів
  for(int j=0; j<size; j++)
    {
    cout << ilist.front() << ' ';  // зчитати елемент з початку
    ilist.pop_front();             // видалити елемент з початку
    }
  cout << endl;
  return 0;
}
```

Результат виконання програми:

```
10 20 30 40
```

У наведеній програмі дані вставляються в кінець і на початок списку, а також видаляються з початку списку.

Методи push_front (), front () та pop_front () аналогічні методам push_back (), back () та pop_back (), які ми вже розглянули при роботі з векторами.

Списки також доцільно використовувати при частих операціях вставки та видалення з середини списку.

Пам’ятайте, що довільний доступ до елементів списку використовувати небажано, оскільки він здійснюється занадто повільно для нормальної обробки даних. Тому оператор [] навіть не визначений для списків.

На рис. 14.2 проілюстровано роботу розглянутих методів для послідовного контейнера Список (list).

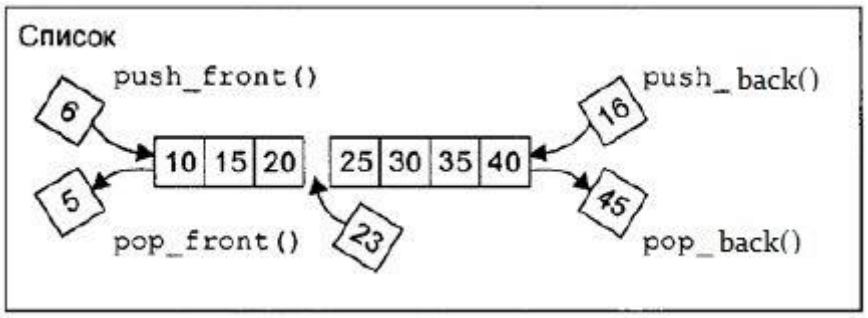



Рис. 14.2. Ілюстрація роботи методів послідовного контейнера list

### Методи reverse(), merge() и unique()

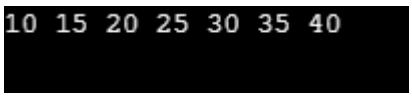
Деякі методи використовують лише зі списками. Інших контейнерів, для яких вони були б визначені, просто немає, хоча є, звичайно, алгоритми, які виконують практично ті самі функції.

У наступному прикладі розглянуто деякі з таких методів.

```
#include <iostream>
#include <list>
using namespace std;
int main()
{
  int j;
  list<int> list1, list2;
  int arr1[] = { 40, 30, 20, 10 };
  int arr2[] = { 15, 20, 25, 30, 35 };
  for(j=0; j<4; j++)
    list1.push_back( arr1[j] );          // list1: 40, 30, 20, 10
```

```
  for(j=0; j<5; j++)
    list2.push_back( arr2[j] );          // list2: 15, 20, 25, 30, 35
  list1.reverse();                       // перевернути list1: 10 20 30 40
  list1.merge(list2);                    // об'єднати list2 та list1
  list1.unique();                        // видалити елементи, що повторюються
  int size = list1.size();
  while( !list1.empty() )
    {
    cout << list1.front() << ' ';        // зчитати елемент з початку
    list1.pop_front();                   // видалити елемент з початку
    }
  cout << endl;
  return 0;
  }
```

Результат роботи програми виглядає наступним чином:




Спочатку списки ініціалізуються елементами масивів за допомогою методу **push_back():**

list1: 40, 30, 20, 10

list2: 15, 20, 25, 30, 35

Далі перший список перевертається за допомогою методу **reverse(),** щоб він був впорядкований за зростанням. Після цієї дії списки виглядають так:

list1: 10 20 30 40

list2: 15 20 25 30 35

Потім виконується метод **merge(),** який об'єднує два відсортовані списки в третій. З його допомогою об'єднуються списки list2 і list1, результат зберігається в list1. Його новий вміст:

list1: 10 15 20 20 25 30 30 35 40.

Нарешті, до списку list1 застосовується метод **unique(),** який знаходить сусідні елементи з однаковими значеннями і залишає у списку лише один з них.

Для виведення списку використовують методи **front()** та **pop_front().** З них складається цикл for, таким чином проходиться весь список. Кожен елемент по черзі від голови до хвоста виводиться на екран, а потім «виштовхується» зі списку. Таким чином на початку списку після кожного кроку опиняється новий елемент. В результаті виходить, що операція виведення на екран знищує список.

## Черги з двостороннім доступом

**Черга з двостороннім доступом** (deque) являє собою дещо схоже і на вектор, і на зв'язний список [2]. Як і вектор, цей тип контейнера підтримує довільний доступ (оператор []). Як і до списку, доступ може бути отриманий і на початку, і в кінці черги. В цілому, цей вид контейнера нагадує вектор з двома кінцями.

Однак, для векторів і двосторонніх черг пам'ять резервується по-різному. Вектор завжди займає суміжні комірки пам'яті. Тому при його розростанні резервується нова ділянка пам'яті, де контейнер може поміститися цілком. З іншого боку, черга з двостороннім доступом може розміщуватися в декількох сегментах пам'яті, не обов'язково суміжних. У цьому є і плюси, і мінуси. Черга практично завжди буде гарантовано розміщена в пам'яті, але доступ до сегментованих об'єктів завжди здійснюється з більш повільною швидкістю.

Наступний приклад демонструє використання розглянутих вище методів **push_back(), push_front (), size ()** та **оператора []** до черги з двостороннім доступом.

```
// черга з двостороннім доступом
#include <iostream>
#include <deque>
using namespace std;
int main()
{
  deque<int> d;
```

```
  d.push_back(30);              // додавання елементів в кінець черги
  d.push_back(40);
  d.push_back(50);
  d.push_front(20);             // додавання елементів на початок черги
  d.push_front(10);
  d[2] = 33;                    // зміна довільного елемента черги
  for(int j=0; j<d.size(); j++)
    cout << d[j] << ' ';        // виведення елементів
  cout << endl;
  return 0;
  }
```

Результат виконання програми:

```
10 20 33 40 50
```

На рис. 14.3 проілюстровано роботу розглянутих методів для послідовного контейнера Черга з двостороннім доступом (deque):

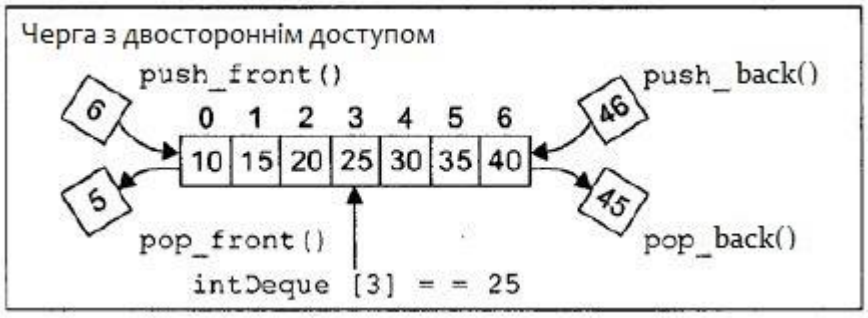



Рис. 14.3. Ілюстрація роботи методів послідовного контейнера deque

## Алгоритми STL

Алгоритми STL виконують різні операції над наборами даних контейнерів (і звичайних масивів C++). Кожен алгоритм реалізований у вигляді шаблону або набору шаблонів функції, тому може працювати з різними видами послідовностей і даними різних типів.

Нагадаємо, що оголошення стандартних алгоритмів STL знаходяться у **заголовку <algorithm>.**

Всі алгоритми STL можна поділити на чотири **категорії**:

• немодифікуючі операції з послідовностями;

• модифікуючі операції з послідовностями;

• операції сортування та відношень;

• операції над множинами.

Крім того, бібліотека містить узагальнені чисельні алгоритми, оголошення яких знаходяться у файлі **<numeric>**.

В якості параметрів алгоритмам передаються ітератори, що визначають початок і кінець послідовності, що обробляється. Вид ітераторів визначає типи контейнерів, для яких може використовуватися той чи інший алгоритм.

### Немодифікуючі операції з послідовностями

Алгоритми цієї категорії переглядають послідовність, не змінюючи її.

Вони використовуються для отримання інформації про послідовність чи для визначення положення елемента. У табл. 14.2 наведено приклади немодифікуючих операцій з послідовностями.

Табл. 14.2. Приклади немодифікуючих операцій з послідовностями

| **Операції, що не модифікують** | |
|---|---|
| **count()** | Працює з одним контейнером. Приймає чотири параметри.<br>Підраховує кількість входжень заданого значення у послідовність |
| **find()** | Працює з одним контейнером. Приймає три параметри.<br>Знаходить перше входження заданого значення у послідовність. |
| **search()** | Працює з двома контейнерами. Приймає чотири параметри.<br>Знаходить перше входження однієї послідовності в іншу. |
| **mismatch()** | Працює з двома контейнерами. Приймає три параметри.<br>Знаходить першу пару значень, що не співпадають, у двох послідовностях. |

### Модифікуючі операції з послідовностями

Алгоритми цієї категорії тим чи іншим чином змінюють послідовність, з якої вони працюють. Вони використовуються для копіювання, видалення, заміни і зміни порядку проходження елементів послідовності. У табл. 14.3 наведено приклади модифікуючих операцій з послідовностями.

Таблиця 14.3. Приклади модифікуючих операцій з послідовностями

| **Операції, що модифікують** | |
|---|---|
| **copy()** | Працює з двома контейнерами. Приймає три параметри.<br>Копіює елементи вказаного діапазону в іншу послідовність. |
| **swap()** | Міняє місцями два елементи |
| **replace()** | Замінює всі елементи з заданим значеннями на нові значення |
| **fill()** | Заміняє всі елементи діапазону заданим значенням |
| **reverse()** | Обертає послідовність елементів з діапазону |

### Алгоритми сортування та відношень

Алгоритми цієї категорії впорядковують послідовності, виконують пошук елементів, злиття послідовностей, пошук мінімуму і максимуму, лексикографічне порівняння, перестановки тощо. У табл. 14.4 наведено приклади модифікуючих операцій сортування та відношень.

Таблиця 14.4. Приклади модифікуючих операцій сортування та відношень

| **Операції сортування та відношень** | |
|---|---|
| **sort()** | Сортує елементи в діапазоні. |
| **merge()** | Об’єднує відсортовані діапазони 1 та 2 в цільовий діапазон 3. |
| **min()** | Знаходить менший з двох елементів |
| **max()** | Знаходить більший з двох |
| **min_element()** | Знаходить найменше значення у послідовності |
| **max_element()** | Знаходить найбільше значення у послідовності |

### Алгоритми роботи з множинами

Алгоритми цієї категорії виконують сортування множин. Операції об’єднання і перетину для множин мають таке ж саме значення, як в теорії множин. Алгоритми роботи з множинами не змінюють вхідні послідовності, а вихідні послідовності впорядковані. У табл. 14.5 наведено приклади модифікуючих операцій над множинами.

Таблиця 14.5. Приклади модифікуючих операцій над множинами

| **Операції над множинами** | |
|---|---|
| **ncludes()** | Визначає включення однієї множини в іншу |
| **set_intersection()** | Створення відсортованого перетину множин |
| **set_union()** | Створення відсортованого об’єднання множин |

## Ітератори STL

Ітератори – ключова частина всієї STL, оскільки саме вони забезпечують зв'язок між алгоритми та контейнерами [5].

Один з аспектів їхнього використання полягає в тому, що ітератори – це так би мовити «інтелектуальні покажчики» на елементи контейнера. У табл. 14.6 показано, які операції підтримуються якими ітераторами.

Таблиця 14.6. Які операції підтримуються якими ітераторами

| **Тип ітератора** | **«Крок вперед» ++** | **Зчитування** | **Запис *i = value** | **«Крок назад» --** | **Довільний доступ [n]** |
|---|---|---|---|---|---|
| Довільного доступу | X | X | X | X | X |
| Двонаправлений | X | X | X | X | |
| Прямий | X | X | X | | |
| Вхідний | X | | X | | |
| Вихідний | X | X | | | |

Як бачимо, всі ітератори підтримують оператор ++ для просування вперед контейнером.

Другий аспект використання ітераторів – це те, що ітератори визначають які алгоритми використовувати з якими контейнерами.

Справа в тім, що теоретично, звичайно, можна застосувати будь-який алгоритм до будь-якого контейнера. І, насправді, так можна і потрібно робити. Це одна з переваг STL. Але правил без винятків не буває. Так, деякі алгоритми дуже неефективні при роботі з певними типами контейнерів. Наприклад, алгоритму sort() потрібний довільний доступ до елементів контейнера, який він намагається сортувати. В іншому випадку доводиться проходити по всіх елементах до тих пір, поки не знайдеться потрібний, що, зрозуміло, займе чимало часу, якщо контейнер солідних розмірів. А для ефективної роботи алгоритму reverse() потрібно мати можливість зворотної та прямої ітерації, тобто проходження контейнера як у зворотному, так і в нормальному порядку.

**Ітератори дозволяють визначати відповідність алгоритмів контейнерам.** Як уже зазначалося, ітератори можна уявляти собі у вигляді

кабелю, що з'єднує, наприклад, комп'ютер і принтер. Один кінець «приєднаний» до контейнера, інший – до алгоритму. Проте, не всі кабелі можна під'єднати до будь-якого контейнера, і не всі кабелі можна під'єднати до будь-якого алгоритму.

### Ітератори і контейнери

Якщо обмежитися розглядом лише основних контейнерів STL, можна констатувати той факт, що достатньо лише двох типів ітераторів: ітератора довільного доступу та двоспрямованого ітератора. Як показано в табл. 14.7, вектори і черги з двостороннім доступом підтримують будь-який тип ітератора, а списки, множини, мультимножини, відображення і мультивідображення сприймають все, крім ітераторів довільного доступу.

Таблиця 14.7. Відповідність ітераторів та контейнерів

| **Тип ітератора** | **Вектор** | **Список** | **Deque** | **Множина** | **Мульти-множина** | **Відобра-ження** | **Мульти-відобра-ження** |
|---|---|---|---|---|---|---|---|
| Довільного доступу | X | | X | | | | |
| Дво-направлений | X | X | X | X | X | X | X |
| Прямий | X | X | X | X | X | X | X |
| Вхідний | X | X | X | X | X | X | X |
| Вихідний | X | X | X | X | X | X | X |

Процес підключення правильного ітератора до контейнера в STL автоматизований.

Зверніть увагу! При визначенні ітератора потрібно вказувати, для якого типу контейнера його слід використовувати.

Наприклад:

```
list<int> iList;                // список int
list<int>::iterator iter;       // ітератор для списку int
```

Після цього STL автоматично робить ітератор двоспрямованим, оскільки саме такий тип ітератора потрібний контейнеру типу список. А ітератор для

вектора або черги з двостороннім доступом автоматично робиться ітератором довільного доступу.

Наприклад:

```
vector<int> aVect;              // вектор цілих чисел
vector <int>::iterator iter;    // ітератор для вектора цілих чисел
```

Зверніть увагу! Вектори та черги з двостороннім доступом завжди «приєднані» до ітератора довільного доступу, а списки (й інші асоціативні контейнери) завжди «приєднані» до двоспрямованого ітератора.

Така автоматизація процесу досягається за рахунок того, що клас ітератора конкретного класу є нащадком класу більш загального ітератора.

Алгоритм завжди може використовувати ітератори з більш широкими можливостями, ніж йому потрібно. Наприклад, якщо потрібен прямий ітератор, абсолютно нормальним вважається використання двоспрямованого ітератора або ітератора з довільним доступом.

### Ітератори та алгоритми

Зрозуміло, кожному алгоритму, в залежності від того, чим він займається, потрібен свій тип ітератора. Якщо необхідно звертатися до довільних елементів контейнера, потрібний ітератор довільного доступу. Якщо ж потрібно черепашачим кроком пересуватися вперед від елемента до елемента, то підійде менш потужний прямий ітератор. У табл. 14.8 наведено приклади алгоритмів і необхідних для них ітераторів.

Таблиця 14.8. Приклади алгоритмів і необхідних для них ітераторів

| Алгоритм | Вхідний | Вихідний | Прямий | Двонаправлений | Довільного доступу |
|---|---|---|---|---|---|
| for_each | X | | | | |
| find | X | | | | |
| count | X | | | | |
| copy | X | X | | | |
| replace | | | X | | |
| unique | | | X | | |
| reverse | | | | X | |
| sort | | | | | X |

| | | | | | |
|---|---|---|---|---|---|
| nth_element | | | | | X |
| merge | X | X | | | |
| accumulate | X | | | | |

Очевидно, що, незважаючи на те, що кожному алгоритму потрібен певний рівень можливостей, більш потужні ітератори теж працюватимуть нормально. Алгоритму replace() потрібний прямий ітератор, але він буде працювати й з двоспрямованим ітератором, й з ітератором довільного доступу.

У той же час, не можна використовувати двоспрямовані ітератори з алгоритмами, яким потрібний довільний доступ. Таким чином, вектори і черги з двостороннім доступом, що використовують ітератори довільного доступу, можуть працювати з будь-яким алгоритмом, а списки та інші асоціативні контейнери з двоспрямованим ітератором – тільки з менш потужними алгоритмами.

### Алгоритми та контейнери

З наведених вище табл. 14.7 і 14.8 можна дізнатися, з якими алгоритмами буде працювати той чи інший контейнер. Наприклад, з табл. 14.8 видно, що алгоритму sort() потрібний ітератор довільного доступу. Табл. 14.6 показує, що єдиними контейнерами, які підтримують такі ітератори, є вектори і черги з двостороннім доступом. Марно навіть намагатися застосувати алгоритм sort() до списків, множин, відображень тощо.

Будь-який алгоритм, якому не потрібний ітератор довільного доступу, буде працювати з будь-яким типом контейнера STL, оскільки всі контейнери використовують двоспрямовані ітератори.

Як бачимо, відносно мала кількість алгоритмів реально потребують ітератор довільного доступу. Тож переважна більшість алгоритмів буде працювати з більшістю контейнерів.

### Робота з ітераторами як інтелектуальними покажчиками

Деякі приклади застосування ітераторів ми вже бачили, коли значення ітераторів поверталися методами begin() та end() контейнерів, але ми не звертали

на цей факт особливої уваги. Тепер розглянемо роботу з ітераторами більш детально.

**Доступ до даних**

У контейнерах, зв'язаних з ітераторами довільного доступу (векторах і чергах з двостороннім доступом), ітерація проводиться дуже просто за допомогою оператора [ ]. Такі контейнери, як списки, які не підтримують довільний доступ, вимагають особливого підходу.

Наступна програма демонструє виводення на екран вмісту контейнера.

```
// виводення на екран вмісту контейнера
#include <iostream>
#include <list>
using namespace std;
int main()
  {
  int arr[] = { 2, 4, 6, 8 };
  list<int> theList;                  // порожній список розміром 0
  for(int k=0; k<4; k++)              // список заповнюється елементами масиву
    theList.push_back( arr[k] );
  list<int>::iterator iter;           // ітератор для списку int
  for(iter = theList.begin(); iter != theList.end(); iter++)
    cout << *iter << ' ';             // виведення списку
  cout << endl;
  return 0;
  }
```

Результат виконання програми:

```
2 4 6 8
```

Ми визначаємо ітератор типу list <int>, що відповідає типу контейнера. Як і у випадку зі змінною-покажчиком, до початку використання ітератора потрібно задати його значення. У циклі for відбувається ініціалізація ітератора значенням методу theList.begin() – це покажчик на початок контейнера. Ітератор може бути

інкрементований оператором ++ для проходження вперед елементами контейнера, а оператором * можна отримати значення того елемента, на який він вказує. Можна навіть порівнювати його з чимось за допомогою оператора != та виходити з циклу при досягненні ітератором кінця контейнера (theList.end()).

Теж саме можна було б реалізувати за допомогою циклу while замість for:

```
iter = theList.begin();
while( iter != theList.end() )
cout << *iter++ << ‘ ‘;
```

**Заповнення контейнера даними. Вставка даних**

Наступний приклад демонструє вставку даних у вже існуючу послідовність (вектор чи чергу з двостороннім доступом) – тобто заповнення контейнера даними.

```
// заповнення контейнера даними
#include <iostream>
#include <list>
using namespace std;
int main()
{
  list<int> iList(5);              // порожній список для 5 значень
  list<int>::iterator it;          // ітератор
  int data = 0;
  for(it = iList.begin(); it != iList.end(); it++)      // заповнення списку даними
    *it = data += 2;
  for(it = iList.begin(); it != iList.end(); it++)      // виведення списку
    cout << *it << ‘ ‘;
  cout << endl;
  return 0;
}
```

Результат виконання програми:

```
2 4 6 8 10
```

Перший цикл заповнює контейнер цілими значеннями 2, 4, 6, 8, 10. Другий цикл призначений для виведення цих значень.

### Алгоритми та ітератори

Алгоритми використовують ітератори в якості параметрів (іноді ще в якості значень, що повертаються).

### Алгоритм find()

Наступний приклад демонструє використання *немодифікуючого* алгоритму **find()** до списку (це можливо, оскільки цьому алгоритму потрібний лише вхідний ітератор).

```
// Пошук числа 8 у списку
#include <iostream>
#include <algorithm>
#include <list>
using namespace std;
int main()
{
  list<int> theList(5);              // порожній список для 5 значень
  list<int>::iterator iter;          // ітератор
  int data = 0;
  for(iter = theList.begin(); iter != theList.end(); iter++)
    *iter = data += 2;               // 2, 4, 6, 8, 10
  iter = find(theList.begin(), theList.end(), 8);       // пошук числа 8
  if( iter != theList.end() )
    cout << "\n Знайдене число 8.\n";
  else
    cout << "\n Число 8 не знайдене. \n";
  return 0;
}
```

Результат роботи програми виглядає наступним чином:

```
Знайдене число 8.
```

**Алгоритм find()** повертає ітератор, який вказує на перший елемент, що дорівнює заданому значенню.

Алгоритм find () має три параметри. Перші два – це значення ітераторів, що визначають діапазон елементів, за якими може здійснюватися пошук. Третій – шукане значення. Якщо алгоритм повертає значення theList.end(), ми розуміємо, що досягнутий кінець контейнера, а шуканий елемент не знайдений. В іншому випадку повертається ітератор на перший елемент, значення якого дорівнює 8. Це означає, що елемент знайдений.

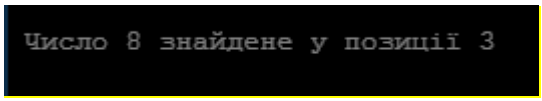
**Визначення позиції елемента в контейнері**

А чи можна за допомогою значення ітератора з'ясувати, де саме в контейнері розташоване число 8? Напевно, так. Можна знайти зміщення ітератора відносно початку (iter-theList.begin()). Але так можна робити лише з ітераторами випадкового доступу, наприклад для векторів і для черг з двостороннім доступом.

```
// пошук номера позиції числа 8 у векторі
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int main()
  {
  vector<int> v(5);                 // порожній вектор для 5 значень
  vector<int>::iterator iter;       // ітератор
  int data = 0;
  for(iter = v.begin(); iter != v.end(); iter++)
    *iter = data += 2;              // 2, 4, 6, 8, 10
  iter = find(v.begin(), v.end(), 8);      // пошук числа 8
  if( iter != v.end() )
```

```
    cout << "\n Число 8 знайдене у позиції "<< (iter - v.begin());
  else
    cout << "\n Число 8 не знайдене.\n";
  return 0;
}
```

Результат виконання програми:




**Алгоритм copy()**

Наступний приклад демонструє використання *модифікуючого* алгоритму copy() до вектора.

```
// копіювання елементів одного вектора в інший
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main()
{
  int first, last;
  int arr[] = { 11, 13, 15, 17, 19, 21, 23, 25, 27, 29 };
  vector<int> v1(arr, arr+10);   // ініціалізований масивом вектор
  vector<int> v2(10);            // порожній вектор для 10 значень
  cout << " Введіть діапазон копіювання (наприклад: 2 5): ";
  cin >> first >> last;
  vector<int>::iterator iter1 = v1.begin() + first;
  vector<int>::iterator iter2 = v1.begin() + last;
  vector<int>::iterator iter3;
  iter3 = copy( iter1, iter2, v2.begin() );       // копіювати діапазон з v1 у v2
  // iter3 -> елемент, що слідує за останнім скопійованим
  iter1 = v2.begin();            // ітерація вперед діапазоном
```

```
  while(iter1 != iter3)              // виведення значень з v2
    cout << *iter1++ << ' ';
  cout << endl;
  return 0;
}
```

Результат виконання програми:

```
 Введіть діапазон копіювання (наприклад: 2 5): 2 7
15 17 19 21 23
```

**Алгоритм copy()** копіює елементи вказаного діапазону в інший контейнер призначення, і повертає ітератор на елемент, що слідує за останнім з тих, які були скопійовані у контейнер призначення.

Використання стандартних алгоритмів, як і інших засобів стандартної бібліотеки, позбавляє програміста необхідності написання і налагодження циклів обробки послідовностей, що зменшує кількість помилок в програмі, знижує час її розробки і робить її більш читабельною і компактною.

## Контрольні запитання

1. Як влаштована внутрішня пам’ять контейнера std::vector? Що відбувається «під капотом», коли його місткість (capacity) вичерпується при спробі додати новий елемент?

2. У чому полягає фундаментальна різниця між вектором та двоспрямованим зв'язаним списком std::list з точки зору швидкості доступу до довільного елемента (за індексом) та швидкості вставки/видалення елементів у середині контейнера?

3. У яких типових задачах контейнер std::deque має перевагу над вектором? Завдяки якій сегментованій структурі пам'яті в ньому реалізовано швидке додавання елементів як у кінець, так і на початок (push_front)?

4. Основи ітераторів: Яке головне призначення ітераторів у бібліотеці STL? Чому вони є життєво необхідною сполучною ланкою між контейнерами даних та алгоритмами їх обробки?

5. Чому роботу з ітераторами розглядають як роботу з «інтелектуальними покажчиками»? Які базові операції (наприклад, розіменування *it, перехід до наступного елемента it++) вони запозичили у звичайних вказників мови С?

6. Що означає термін «інвалідація ітераторів» (iterator invalidation)? Наведіть практичний приклад, коли збережений ітератор на елемент вектора стає недійсним після додавання нових елементів, і спроба звернутися за ним призводить до краху програми.

7. Чому стандартні алгоритми (наприклад, std::find, std::count, std::replace) приймають як аргументи саме діапазон у вигляді пари ітераторів (початок і кінець), а не сам контейнер повністю?

8. Чи всі стандартні алгоритми можуть працювати з будь-якими контейнерами? Чому, наприклад, швидке сортування std::sort чудово працює з ітераторами вектора та черги з двостороннім доступом, але видасть величезну помилку компіляції при спробі застосувати його до ітераторів списку std::list?

## Лекція №15. Потокові класи. Файлові потоки.
## Потокове введення-виведення файлів. Потоки C++

Принциповим для розуміння C++-системи введення-виведення є те, що вона спирається на поняття *потоку* [2, 3].

У C++ **потік** являє собою об’єкт деякого класу.

1. **Потік (stream)** – це абстракція, – загальне поняття, що відноситься до будь-якого потоку даних від джерела до приймача як логічного пристрою.

Потік визначається як послідовність байтів і не залежить від конкретного пристрою, з яким проводиться обмін (оперативна пам’ять, файл на диску, клавіатура, екран або принтер). Тож потік можна назвати логічним інтерфейсом з пристроєм.

В операціях введення байти пересилаються від пристрою (наприклад, від клавіатури, дисководу чи з’єднання мережі) в оперативну пам’ять. При виведенні – з оперативної пам’яті на пристрої (наприклад, на екран дисплею, принтер, дисковод чи у з’єднання мережі).

За видом пристроїв, з якими працює потік, потоки можна розділити на стандартні, файлові та рядкові.

2. **Стандартні потоки** призначені для передачі даних від клавіатури на екран дисплея.

3. **Файлові потоки** призначені для обміну інформацією з файлами на зовнішніх носіях даних (наприклад, на магнітному диску).

4. **Рядкові потоки** призначені для роботи з масивами символів в оперативній пам’яті.

Характер поведінки всіх потоків однаковий, незважаючи на різні логічні пристрої, з якими вони зв’язуються. Оскільки потоки діють однаково, то практично до всіх типів пристроїв можна застосувати одні й ті самі функції та оператори введення-виведення. Наприклад, методи, які використовують для виведення даних на екран, також можна використовувати для виведення даних на принтер або для запису в дисковий файл.

Зчитування даних з потоку називають **вилученням**, виведення у потік – **додаванням**.

За напрямом обміну даними потоки можна розділити на **вхідні** (дані вводяться в пам'ять), **вихідні** (дані виводяться з пам'яті) та **двонаправлені** (допускають як вилучення, так і додавання). Так, наприклад, cin – це стандартний вхідний потік, а cout – стандартний вихідний потік.

Для збільшення швидкості передачі даних обмін з потоком проводиться, як правило, через спеціальну область оперативної пам'яті – ***буфер***.

Потоки C++ забезпечують надійну роботу як зі стандартними, так і з визначеними користувачем типами даних, оскільки реалізовані на основі шаблонних класів. Різні потоки призначені для подання різних видів даних.

C++ - система введення-виведення використовує **заголовок <iostream>**, в якому для підтримки операцій введення-виведення визначена доволі складна ієрархія класів.

### Класи потоків

Потокові класи мають досить складну ієрархічну структуру [1]. На рис. 15.1 показана організація найважливіших з них. Ця ієрархія побудована на основі двох різних (але пов'язаних) базових класів – ios та streambuf.

Клас **ios** містить загальні для введення та виведення поля і методи. Клас **streambuf** забезпечує буферизацію потоків, а також їхню взаємодію з логічними пристроями.

Від цих класів наслідується клас **istream** для вхідних потоків та **ostream** – для вихідних. Два останні класи є базовими для класу **iostream**, що реалізує двонаправлені потоки. Нижче в ієрархії класів розташовуються файлові і рядкові потоки.

Ми вже зустрічалися з використанням деяких потокових класів. Операція **витягання з потоку (>>)** є методом класу istream, операція **додавання в потік (<<)** – методом класу ostream. Обидва ці класи є нащадками класу ios.

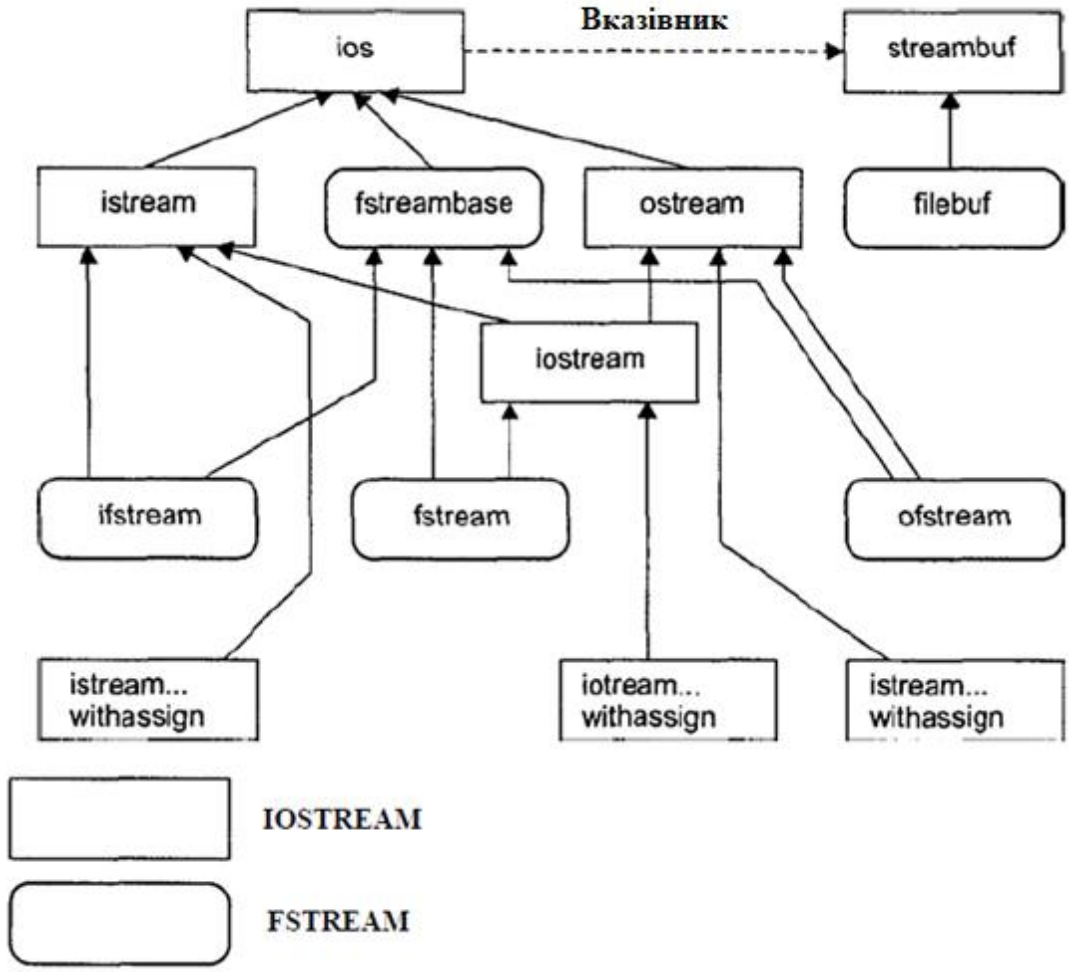



Рис. 15.1. Ієрархія потокових класів

## Файлові потоки

Переважній більшості програм потрібно зберігати дані у файлах на диску та зчитувати їх.

Під **файлом** зазвичай мається на увазі іменована інформація на зовнішньому носії, наприклад, на жорсткому або гнучкому магнітному диску. Логічно файл можна уявити як кінцеву кількість послідовних байтів, тому такі пристрої, як дисплей, клавіатуру і принтер також можна розглядати як окремі випадки файлів.

За способом доступу файли можна розділити на **послідовні** (читання і запис в яких відбувається з початку байт за байтом), та **файли з довільним доступом** (допускають читання і запис у зазначену позицію).

Робота з дисковими файлами передбачає використання спеціальних класів. Стандартна бібліотека містить три класи для роботи з файлами (табл. 15.1): **ifstream** для введення, **ofstream** для виведення, а також **fstream** – для введення та виведення.

Таблиця 15.1. Класи для роботи з файлами

| **Клас** | **Інстанцовано з шаблону** | **Базовий шаблонний клас** | **Призначення** |
|---|---|---|---|
| ifstream | basic_ifstream | basic_istream | Вхідний файловий потік |
| ofstream | basic_ofstream | basic_ostream | Вихідний файловий потік |
| fstream | basic_fstream | basic_iostream | Двонаправлений файловий потік |

Ці класи є похідними від класів istream, ostream та iostream відповідно, тому вони наслідують перевантажені операції << та >>, прапори форматування, маніпулятори, методи, стан потоків тощо. До того ж файлові класи використовують принцип множинного наслідування, будучи нащадками ще й класу fstreambase. Цей клас містить об’єкт класу filebuf, що є файловим буфером, а також асоційовані методи, наслідувані від більш загального класу streambuf.

**Файлові операції введення-виведення**

Файлові операції введення-виведення можна реалізувати після включення в програму **файлу заголовку <fstream>**, в якому визначені всі необхідні для цього класи.

Використання файлів у програмі передбачає наступні **операції**:

- створення потоку;
- відкриття потоку та зв’язування його з файлом;
- обмін з потоком (введення / виведення);
- знищення потоку;
- закриття файлу.

У C++ файл відкривається шляхом зв’язування його з потоком. Щоб відкрити вхідний потік, необхідно оголосити потоковий об’єкт типу ifstream. Для відкриття вихідного потоку потрібно оголосити потік класу ofstream. Потік, який

передбачається використовувати для операцій як введення, так і виведення, повинен бути оголошений як об’єкт класу fstream.

Кожен клас файлових потоків містить кілька конструкторів, які дозволяють варіювати способи створення потокових об’єктів.

• ***Конструктори з параметрами*** створюють об’єкт відповідного класу, відкривають файл з вказаним ім’ям і пов’язують файл з об’єктом.

Наприклад:

```
ifstream (const char *filename, int mode = ios::in);
ofstream (const char *filename, int mode = ios::out | ios::trunc);
fstream (const char *filename, int mode = ios::in | ios::out);
```

Тут перший параметр filename означає ім’я файлу. Другий параметр mode визначає спосіб відкриття файлу. Він повинен приймати одне або кілька значень **бітових масок**/режимів (об’єднаних операцією |), які визначені в класі ios:

**in** // *відкрити файл для введення / для читання*

**out** // *відкрити файл для виведення / для запису*

**ate** // *читання, починаючи з кінця файлу*

**app** // *запис, починаючи з кінця файлу*

**trunc** // *якщо файл існує, обрізати його до нульової довжини*

**binary** // *відкрити у двійковому режимі*

**nocreate** // *якщо файл не існує, видати помилку*

**noreplace** // *якщо файл існує, видати помилку*

Зверніть увагу! За замовчуванням для ifstream встановлюється in, а для ofstream – out.

• ***Конструктори без параметрів*** створюють об’єкт відповідного класу, не зв’язуючи його з файлом.

Наприклад:

```
ifstream in;          // вхідний поток
ofstream out;         // вихідний поток
fstream mystream;     // потік введення-виведення
```

У цьому випадку зв’язок потоку з конкретним файлом здійснюється пізніше – викликом методу open(), який має параметри, аналогічні параметрам розглянутих вище конструкторів.

Нижче розглянемо приклади створення потокових об’єктів і зв’язування їх з конкретними файлами:

```
// Відкриття вихідних файлів - Файли для виведення
ofstream mystream("myfile");
ofstream fout1, fout2;
fout1.open("test1", ios::app);
fout2.open("test2", ios::binary);
// Файл для введення
ifstream fin1("data.txt");
// Файл для введення та виведення
fstream myfile;
myfile.open("mf.dat");
```

Якщо в якості параметра filename задано коротке ім’я файлу (без специфікатора шляху), то мається на увазі, що файл відкривається в поточному каталозі. В іншому випадку потрібно задавати повне ім’я файлу.

Наприклад:

```
ifstream fin1 ("D: \\ VCwork \\ Task1 \\ data.txt");
```

**Обробка помилок файлового введення-виведення**

Перш ніж робити спробу отримати доступ до файлу, слід завжди перевіряти результат виклику **функції open()**. Це можна зробити за допомогою наступної if-інструкції:

```
if(!mystream)
{
cout << "Не вдалося відкрити файл. \n";    // обробка помилки
}
```

Або використати булеву **функцію is_open()**, яка є членом класів fstream, ifstream та ofstream:

```
if(!mystream.is_open())
```

```
{
cout << " Не вдалося відкрити файл. \n";  // обробка помилки
}
```

### Закриття файлів

Зазвичай потреби закривати файли вручну немає – відповідні об'єкти запускають деструктори і закривають асоційовані файли. Якщо ж така потреба виникає, то для закриття файлу використовують **функцію close()**.

Наприклад:

```
mystream.close( ) ;                    // закриття файлу
```

Зверніть увагу! Функція close() не має параметрів і не повертає ніякого значення.

### Запис текстових файлів

При форматованому введенні/виведенні числа зберігаються на диску у вигляді серії символів. Наприклад, число 6,02 замість того, щоб зберігатися у вигляді чотирибайтового значення типу float або восьмибайтового double, зберігається у вигляді послідовності символів '6', ',', '0', '2'. Це дивно з точки зору економії пам'яті, проте легко застосовується на практиці.

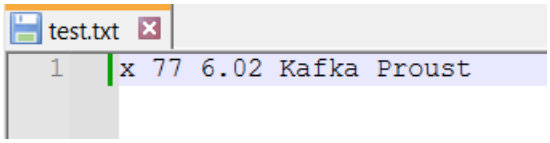
Найпростіше зчитувати дані з текстового файлу чи записувати їх у файл за допомогою **операторів "<<"** та **">>"**.

Наступний приклад демонструє **запис у файл** символу, цілого числа, числа з плаваючою крапкою та двох рядків без пробілів. Виведення на екран не відбувається.

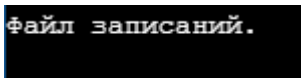
```
// запис у файл
#include <iostream>
#include <fstream>                    // для файлового введення/ виведення
#include <string>
using namespace std;
int main()
{
   char ch = 'x';
```

```
    int j = 77;
    double d = 6.02;
    string str1 = "Kafka";          // рядки без пробілів
    string str2 = "Proust";
  ofstream outfile("test.txt");     // створити об'єкт ofstream,
                                    // відкрити його для запису
  if(!outfile)
    {
    cout << "Не вдається відкрити файл. \n";       // обробка помилки
    return 1;
    }
// вставити (записати) дані у файл
  outfile << ch << ' ' << j << ' ' << d << ' ' << str1 << ' ' << str2;
   cout << "Файл записаний. \n";
   return 0;
}
```

Подивитися на результат роботи програми можна, відкривши файл test.txt за допомогою програми Notepad++.




На екран в результаті роботи програми буде виведене повідомлення:




Тут ми визначили об'єкт outfile як компонент класу ofstream. У той же час ми ініціалізувати його файлом test.txt. Ініціалізація резервує для дискового файлу із заданим ім'ям різні ресурси та отримує доступ (або *відкриває файл*) до нього. Якщо файлу не існує, він створюється. Якщо файл вже існує, він переписується – нові дані в ньому замінюють старі. Об'єкт outfile поводиться подібно cout з попередніх програм, тому можна використовувати операцію вставки (<<) для виведення у файл змінних будь-якого зі стандартних типів. Це чудово працює

оскільки оператор вставки перевантажений в класі ostream, який є базовим для ofstream.

Коли програма завершується, об’єкт outfile викликає свій деструктор, який закриває файл, тож нам не доводиться робити це явно за допомогою функції outfile.close().

Існує кілька проблем щодо форматованого виведення у дискові файли. По-перше, необхідно розділяти числа (77 і 6,02 у прикладі) нечисловими символами. Оскільки числа зберігаються у вигляді послідовності символів, а не у вигляді полів фіксованої довжини, це єдиний шанс дізнатися при *витяганні*, де закінчується одне і починається інше число.

По-друге, роздільники мають бути між рядками – з тих самих причин. Це передбачає, що всередині рядка не може бути пробілів. У цьому прикладі для розділення даних ми використовували пробіл в обох випадках.

### Зчитування даних з текстових файлів

Прочитати текстовий файл можна з використанням об’єкта типу **ifstream**, ініціалізованого ім’ям файлу. Файл автоматично відкривається при створенні об’єкта. Потім дані з нього можна зчитувати за допомогою **оператора вилучення (>>).**

В наступній програмі виконаємо зчитування зі створеного у попередньому прикладі файлу.

```
// зчитування з файлу
#include <iostream>
#include <fstream>
#include <string>
using namespace std;
int main()
{
  char ch;
  int j;
  double d;
```

```
  string str1;
  string str2;
  ifstream infile("test.txt");        // створити об’єкт ifstream
// відкрити його для читання
  if(!infile)
    {
    cout << "Не вдається відкрити файл. \n";
    return 1;
    }
  infile >> ch >> j>> d >> str1 >> str2;   // прочитати дані з файлу
  cout << ch << endl                  // вивести дані
       << j << endl
       << d << endl
       << str1 << endl
       << str2 << endl;
  return 0;
}
```

Об’єкт infile типу ifstream діє приблизно так само, як cin в попередніх програмах. Якщо коректно вставляли дані в файл, то витягти їх не складе ніяких проблем. Ми просто зберігаємо їх у відповідних змінних і виводимо на екран. Звісно, числа приводяться назад до свого бінарного подання, щоб з ними можна було працювати в програмі.

В результаті виконання програми на екран буде виведена наступна інформація:

```
x
77
6.02
Kafka
Proust
```

Зверніть увагу! При використанні оператора ">>" для зчитування даних з текстових файлів відбувається перетворення деяких символів. Наприклад,

символи пробілів опускаються. Якщо необхідно запобігти будь яким перетворенням символів, відкрийте файл у двійковому режимі доступу. Крім того, пам’ятайте, що при використанні оператора ">>" для зчитування рядка введення припиняється при виявленні першого символу «пробілу».

**Запис у файл рядків з пробілами**

Підхід, продемонстрований в останньому прикладі, не дозволяє обробляти рядки з char *, що містять пробіли. В наступному прикладі розглянемо, як можна записувати у файл рядки з пробілами.

```
// запис у файл рядків з пробілами
#include <fstream>
using namespace std;
int main()
{  ofstream outfile("test.txt");            // створити вихідний файл
// відкрити його для запису
   outfile << "Хай сонце – прабог всіх релігій - \n";
   outfile << "золотопере й життєсійне, \n";
   outfile << "благословить мій дім крилатий. \n";
   outfile << "Накреслю взір його неземний, \n";
   outfile << "святий, арійський знак таємний, \n";
   outfile << "накреслю свастику на хаті, \n";
   outfile << "і буду спати вже спокійний. \n";
   return 0;
}
```

В результаті виконання програми рядки вірша Богдана-Ігоря Антонича «Молитва» будуть записані у файл test.txt:

test.txt

```
1  Хай сонце – прабог всіх релігій -
2  золотопере й життєсійне,
3  благословить мій дім крилатий.
4  Накреслю взір його неземний,
5  святий, арійський знак таємний,
6  накреслю свастику на хаті,
7  і буду спати вже спокійний.
8
```

Зауважте, що кожен рядок закінчується спеціальним обмежувальним символом – **символом розділення рядків (‘\n’)**.

Зверніть увагу! В наведеній програмі рядки типу char*, а не об’єкти класу string. Багато потокових операцій легше виконувати з цим типом даних.

### Зчитування з файлу рядків з пробілами

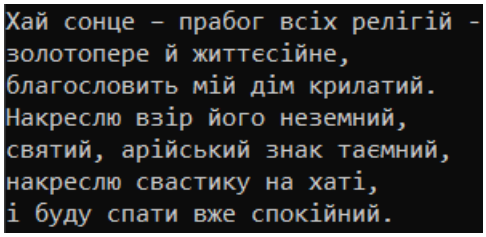
Для того, щоб витягти рядки з файлу, треба створити **ifstream** і читати рядок за рядком функцією **getline()**, яка є методом класу istream [5]. Ця функція зчитує всі символи, включаючи роздільники, поки не дійде до спеціального символу розділення рядків ‘\n’. Потім вона розміщує результат читання у буфер, переданий в якості аргумента. Максимальний розмір буфера передається за допомогою другого аргумента функції getline(). Вміст буфера виводиться на екран після зчитування кожного рядка.

В наступній програмі продемонструємо зчитування з файлу рядків з пробілами – виведемо на екран строфу з вірша, яку ми записали у файл test.txt у попередньому прикладі:

```
// зчитування з файлу рядків з пробілами
#include <fstream>                  // для файлових функцій
#include <iostream>
using namespace std;
int main()
{
   const int MAX = 80;              // розмір буфера
   char buffer[MAX];                // буфер символів
   ifstream infile("test.txt");     // створити вхідний файл
   while( !infile.eof() )           // цикл поки не EOF
   {
      infile.getline(buffer, MAX);  // зчитує рядок тексту
      cout << buffer << endl;       // та виводить його
   }
   return 0;
```

```
}
```

Результат виконання програми:




Програма не знає, скільки всього рядків у тексті, тому вона продовжує зчитування до тих пір, поки не зустріне **ознаки закінчення файлу (EOF).** Сигнал EOF надсилається в програму операційною системою, коли більше немає даних для зчитування.

Зверніть увагу! Цю програму не можна застосовувати для читання довільних файлів, – кожен рядок тексту повинен закінчуватися символом розділення рядків ‘\n’. При спробі прочитати файл іншої структури програма зависне.

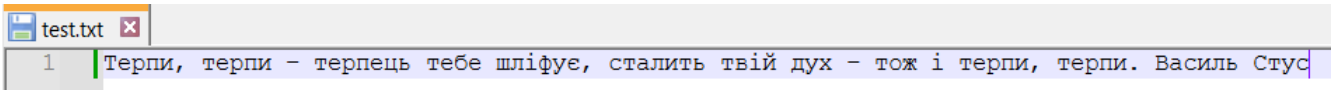
### Посимвольне файлове введення-виведення

Наступний приклад демонструє посимвольний запис рядка у файл за допомогою **функції put().**

```
// посимвольне файлове введення
#include <fstream>                    // для файловых функций
#include <iostream>
#include <string>
using namespace std;
int main()
{
   string str = "Терпи, терпи – терпець тебе шліфує, "
                " сталить твій дух – тож і терпи, терпи. Василь Стус";
   ofstream outfile("test.txt");      // створити вихідний файл
   for(int j=0; j<str.size(); j++)    // кожний символ записати у файл
      outfile.put( str[j] );
   cout << " Файл записаний \n";
```

```
    return 0;
}
```

В результаті роботи програми рядки вірша відомого українського письмнника, Василя Стуса, буде записано у файл test.txt:




У цій програмі створюється об’єкт **ofstream**. Довжина об’єкта класу string на ім’я str знаходиться за допомогою методу **size()**, а символи записуються у файл в циклі for функцією **put()**.

**Посимвольне зчитування рядка з файлу та виведення його на екран** демонструє наступна програма:

```
// посимвольне файлове виведення
#include <fstream>
#include <iostream>
using namespace std;
int main()
{
   char ch;                          // символ для зчитування
   ifstream infile("test.txt");      // створити вхідний файл
   while( infile )                   // читати до EOF чи помилки
   {
      infile.get(ch);                // зчитати символ
      cout << ch;                    // та вивести його
   }
   cout << endl;
   return 0;
}
```

У цій програмі використана функція **get()**. Зчитування проводиться до ознаки закінчення файлу EOF (або виникнення помилки). Кожен прочитаний символ виводиться на екран за допомогою **cout**.

В результаті виконання програми на екран буде виведений весь текст.

```
Терпи, терпи – терпець тебе шліфує, сталить твій дух – тож і терпи, терпи. Василь Стус
```

## Контрольні запитання

1. Що таке потоки C++ і яка їхня головна абстракція? Як вони допомагають уніфікувати роботу з різними пристроями (клавіатурою, екраном, файлами)?

2. Які основні класи потоків існують у стандартній бібліотеці? Назвіть три базові класи, які використовуються спеціально для роботи з файлами (наприклад, ifstream, ofstream, fstream).

3. Чим файлові потоки концептуально відрізняються від стандартних консольних потоків (cin, cout)? Які додаткові кроки (відкриття/закриття) необхідні для роботи з ними?

4. Як виконуються базові файлові операції введення-виведення? Які режими відкриття файлу (наприклад, для додавання ios::app або перезапису) ви знаєте?

5. Як повинна здійснюватися обробка помилок файлового введення-виведення? Які методи (наприклад, is_open(), fail()) дозволяють перевірити, чи успішно відкрито файл і чи не виникло помилок під час читання?

6. Чому закриття файлів після завершення роботи з ними є обов’язковим кроком? Що відбувається з даними в буфері при виклику методу close() і до яких наслідків може призвести ігнорування цього правила?

7. Як відбувається стандартний запис текстових файлів та зчитування даних з текстових файлів за допомогою операторів << та >>? Як ці оператори реагують на розділювачі (пробіли, табуляції, переноси рядків)?

8. У чому полягає складність, коли потрібен запис у файл рядків з пробілами або зчитування з файлу рядків з пробілами? Чому стандартний оператор >> тут не працює і як функція getline() вирішує цю проблему?

9. Для яких задач використовується посимвольне файлове введення-виведення? За допомогою яких методів (наприклад, get(), put()) можна обробляти файл символ за символом без ігнорування пробілів чи символів нового рядка?

## Список використаної літератури

1. Lafore R. Object-Oriented Programming in C++. 4th ed. Indianapolis : Sams Publishing, 2002. 1012 p.

2. Stroustrup, Bjarne. The C++ Programming Language. 4th ed., Addison-Wesley, 2013.

3. Langr, Jeff. Modern C++ Programming with Test-Driven Development: Code Better, Sleep Better. Dallas, TX: Pragmatic Bookshelf, 2013.

4. Barr, Michael. Programming Embedded Systems in C and C++. O'Reilly Media, 1999.

5. Modern C++ Programming Cookbook: Recipes to explore data structure, multithreading, and networking in C++17 , ISBN 978-1-78646-518-4, 2017. – 559 p.

6. Amini K. Extreme C: Taking You To The Limit In Concurrency, OOP, And The Most Advanced Capabilities Of C, New York: Packt Publishing, 2019. – 823 p.

7. Я. М. Глинський, В. Є. Анохін, В. А. Ряжська, C++ і C++ Builder, навчальний посіб. 5-те вид. – Львів: [СПД Глинський], 2011. – 192 с.

8. О. Васильєв Програмування C++ у прикладах і задачах. – Ліра-К, 2017. – 382 с.

9. О. Трофименко C++ Основи програмування. Теорія і практика. – Одеса: Фенікс, 2010. – 544 с.