**Міністерство надзвичайних ситуацій України**
**Львівський державний університет безпеки життєдіяльності**

**Юрій ГРИЦЮК,**
**Тарас РАК**

# ПРОГРАМУВАННЯ МОВОЮ С++

**Навчальний посібник**

**Львів**
**Вид-во ЛДУ БЖД**
**2011**

ББК 32.97я73
Г85
УДК 681.322[075.8]

А в т о р и
Ю.І. ГРИЦЮК – д-р техн. наук, доцент, завідувач кафедри управління інформаційною безпекою Львівського ДУ БЖД;
Т.Є. РАК – канд. техн. наук, доцент, підполковник сл. цив. захисту, начальник інституту цивільного захисту Львівського ДУ БЖД

Р е ц е н з е н т и:
А.Д. КУЗИК – канд. фіз.-мат. наук, доцент, доцент кафедри фундаментальних дисциплін Львівського ДУ БЖД (м. Львів);
В.М. СЕНЬКІВСЬКИЙ – д-р техн. наук, професор, завідувач кафедри електронних видань Української академії друкарства (м. Львів),
Д.Д. ПЕЛЕШКО – д-р техн. наук, доцент, професор кафедри автоматизованих систем управління НУ "Львівська політехніка" (м. Львів)

Відповідальний редактор В.В. ДУДОК

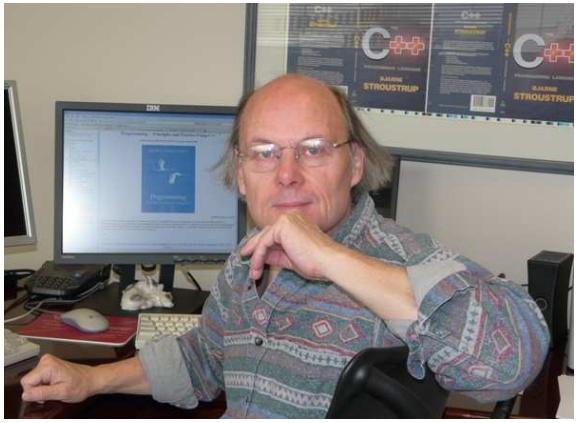
*Затверджено до друку Вченою радою*
*Львівського державного університету безпеки життєдіяльності*
*(прот. № 8 від 19.05.2011 р.)*

**ISBN 978-966-3466-85-9**

# ЗМІСТ

# ПЕРЕДМОВА

Процес оволодіння навиками програмування мовою C++ має багато спільного з науково-дослідною діяльністю. Пов'язано це з тим, що ця мова поєднує декілька технологій програмування – традиційну, тобто структурне програмування (представлене мовою C), об'єктно-орієнтоване програмування (представлене таким поняттям як клас, який підвищує потужність мови C++ порівняно з мовою C) і узагальнене програмування (програмування за допомогою шаблонів мови C++). Розроблена Б'ярном Страуструпом (англ. Bjarne Stroustrup[1]) мова C++ постійно перебуває в стані розвитку, позаяк відбувається її доповнення новими функціональними можливостями, однак на даний час, з прийняттям ще у 1998 р. стандарту ISO/ANSI C++, специфікація та синтаксис мови стабілізувалися. Сучасні компілятори підтримують практично усі функції, дозволені цим стандартом, і більшість програмістів мали достатньо часу, щоб встигнути звикнути до них.



**Б'ярн Страуструп**

У цьому навчальному посібнику огляд особливостей програмування мовою C++ дається в поєднанні з функціональними можливостями мови C, що робить його самодостатнім. Це означає, що студент вчиться не тільки виконувати дії з новими компонентами мови C++, але й опановує її основу – мову C. Вивчення починається з тих елементів мови, які є спільними для C і C++. Окрім цього, тут особливо акцентовано увагу на тих поняттях мови, важливість яких стане очевидною в процесі подальшого вивчення матеріалу, а також на відмінностях між мовами C і C++. Після того, як студент твердо засвоїть основи програмування мовою C++, він може приступати до вивчення її надбудови – об'єктно-орієнтованого програмування. Вже там він зможе

[1] http://www2.research.att.com/~bs/





скласти цілісне уявлення про те, що таке об'єкти, класи, шаблони і як вони реалізуються стандартом мови C++. У процесі викладення матеріалу опис тих чи інших положень мови C++ продемонстровано невеликими навчальними програмами, які студенту нескладно скопіювати і самостійно випробувати на власному комп'ютері та проаналізувати отримані результати.

Матеріал цього навчального посібника повністю відповідає робочій навчальній програмі такої дисципліни як "Програмування мовою C++", яка викладається у Львівському державному університеті безпеки життєдіяльності для курсантів і студентів напрямку підготовки "Управління інформаційною безпекою" та вивчають комп'ютерні методи захисту інформації.

Ми (автори, рецензенти і літературний редактор) приклали багато зусиль, щоби зробити матеріал цього навчального посібника конкретним, простим для розуміння, засвоєння та, що найважливіше, цікавим. Наша мета полягала у тому, щоби, внаслідок вивчення того чи іншого матеріалу, студенти могли створювати свої власні навчальні програми і, водночас, знаходити в цьому як користь, так і задоволення.

**Подяки**

Автори вдячні усім тим, хто так чи інакше сприяв написанню цього навчального посібника. Особлива подяка доктору технічних наук, професору Я.І. Соколовському, завідувачу кафедри ОТіМТП НЛТУ України за цінні вказівки і безпосередню підтримку.

Автори висловлюють вдячність рецензентам доценту Андрію Даниловичу Кузику, професору Всеволоду Миколайовичу Сеньківському і професору Дмитру Дмитровичу Пелешку, а також літературному редактору В.В. Дудку за цінні зауваження та уточнення, які сприяли значному покращенню рукопису.

Автори з вдячністю приймуть будь-які конструктивні зауваження стосовно викладеного у посібнику матеріалу, які можна надсилати за адресою:

*Кафедра управління інформаційною безпекою*

*Львівський ДУ БЖД, вул. Клепарівська 35, м. Львів, 79007*

***Тел. моба*** *067-944-11-15.* ***E-mail****: gryciuk.yura@gmail.com*

# ВСТУП

Основне завдання цього навчального посібника – навчити студентів розробляти програми мовою С++ з використанням технології структурного програмування. Не секрет, що протягом декількох останніх десятиліть на ринку програмного забезпечення відбулися значні зміни, які є очевидними навіть для не професіонала. Зокрема, мова С++ стала універсальною мовою програмування[1] високого рівня з підтримкою декількох сучасних парадигм програмування: об'єктно-орієнтованої, узагальненої та процедурної. Вона була розроблена Б'ярном Страуструпом (англ. Bjarne Stroustrup) в AT&T Bell Laboratories (м. Мюррей-Хілл, шт. Нью-Джерсі) у 1979 р. під назвою "С з класами" (C with Classes). Згодом, у 1983 р. її перейменували на мову С++.

Назву мові "С++" придумав Рік Масситті (Rick Mascitti), яка походить від оператора мови С інкремента "++" (збільшення значення змінної на одиницю). У цей час також був поширений спосіб присвоєння нових імен комп'ютерним програмам, що полягав у додаванні до назви символу "+" для позначення її нових модифікацій. Згідно з задумом Б'ярна Страуструпа, "ця назва вказує на еволюційну природу змін мови С". Виразом "С+" називали попередню мову програмування, зовсім не пов'язану з мовою С++. Багато зі фахівців можуть зауважити, що введення назви мови С++ не змінює самої мови програмування С, тому найточнішим іменем мало б бути "С+1".

У 1990-х роках мова програмування С++ стала однією з найуживаніших мов програмування загального призначення. При створенні мови С++ прагнули зберегти її синтаксис сумісний з мовою С. Більшість програм, написаних мовою С, справно працюють і з компілятором мови С++. Нововведеннями мови програмування С++ порівняно з мовою С було: підтримка об'єктно-орієнтованого програмування через класи і об'єкти; підтримка узагальненого програмування через шаблони; доповнення до стандартної бібліотеки; додаткові типи даних; оброблення винятків; простори імен; вбудовані функції; перевизначення операторів та імен функцій; посилання та оператори управління вільно розподіленою пам'яттю. У 1998 р. ратифіковано міжнародний стандарт мови С++: ISO/IEC 14882 "Standard for the C++ Programming Language". Подальша версія цього стандарту мала назву – ISO/IEC 14882:2003.

### Базові знання, необхідні студентам для роботи з цим навчальним посібником

Матеріал, викладений у цьому навчальному посібнику, доступний навіть тим студентам, хто вивчає програмування "з нуля". Проте наявність досвіду програмування такими мовами, як Pascal чи Visual Basic, не стоятиме на заваді при засвоєнні матеріалу, викладеному у цьому посібнику.

[1] http://uk.wikipedia.org/wiki/C%2B%2B





Багато підручників з описом мови програмування С++ розраховано на те, що студенти вже знайомі з мовою С. Наш навчальний посібник починає вивчення мови програмування С++ "з чистого листа", тому відсутність знань мови С ніяк не позначиться на ефективності засвоєння процесу розроблення працездатної програми. Бажано також, щоб студенти, вивчаючи матеріал з цього навчального посібника, мали уявлення про основні операції Microsoft Windows, такі як запуск програм на виконання, копіювання файлів тощо.

У цьому посібнику вивчення технології програмування починається з найпростіших прикладів розроблення програм, поступово ускладнюючи їх, і завершуючи повноцінними структурними програмами. Інтенсивність подання нового матеріалу дає змогу студентам без зайвого поспіху його вивчити і, при потребі, повторити пройдений. Для полегшення процесу його засвоєння у кожному розділі наведено достатню кількість наочних прикладів з програмною їх реалізацією, а також детальним аналізом їх програмних блоків. Зверніть увагу також на те, що у цьому посібнику містяться навчальні приклади програм, початкові коди яких є консольними, тобто такими, що виконуються в текстовому вікні компілятора. Це зроблено для того, щоби уникнути труднощів, які виникають під час роботи з повноцінними графічними додатками Windows.

Найбільш популярним середовищем для розроблення програм мовою С++ є програмний продукт, розроблений спільно з компаніями Microsoft і Borland, призначений для роботи під управлінням операційної системи Microsoft Windows. Наведені у посібнику приклади кодів програм розроблено так, що вони підтримуються як компіляторами Microsoft, так і компіляторами Borland. Детальну інформацію про компілятори можна знайти у відповідних документаціях до "Microsoft Visual C++" і до "Borland C++ Builder". Інші компілятори, розраховані на стандарт мови програмування С++, безпомилково сприйматимуть більшість кодів програм у тому вигляді, у якому їх наведено у цьому посібнику.

Часто серед студентів побутує думка, згідно з якою мову програмування С++ важко вивчити. Насправді вона має багато спільного з такими мовами програмування як Pascal чи Visual Basic, за винятком декількох вдало реалізованих нових ідей. Забігаючи наперед, зазначимо, що ці ідеї є цікавими та повчальними, тому їх вивчення навряд чи виявиться для Вас зайвим. Окрім цього, про зміст цих ідеї необхідно мати уявлення хоча би тому, що вони є не тільки основою сучасних технологій програмування і створення програмних продуктів, але й становлять значну частину "культури програмування".

Будемо сподіватися на те, що цей навчальний посібник допоможе студентам розібратися як зі специфікою технології структурного програмування мовою С++, так і у загальних концепціях і принципах створення програмних продуктів. Запропонований матеріал буде корисним як студентам, що вивчають програмування, так і аспірантам, а також усім тим, хто проявляє інтерес до даної області знань.

# Розділ 1. ІСТОРІЯ ВИНИКНЕННЯ МОВИ ПРОГРАМУВАННЯ С++

Мова програмування С++ – одна з найдосконаліших мов, яку має засвоїти потенційний програміст-початківець. Ця заява може видатися дуже серйозною, але вона – не перебільшення. Мова С++ – центр притягання, навколо якого "обертається" все сучасне програмування. Її синтаксис і принципи розроблення програм визначають суть об'єктно-орієнтованого програмування. Понад це, мова С++ вторувала шлях для розроблення багатьох інших мов майбутнього. Наприклад, мови Java та C# – прямі "нащадки" мови С++. Її також можна назвати універсальною мовою програмування, оскільки вона дає змогу фаховим програмістам обмінюватися різноманітними ідеями. Сьогодні бути професійним програмістом високого класу означає бути компетентним у тонкощах мови С++, позаяк це ключ до оволодіння сучасними перспективними технологіями програмування.

Приступаючи до вивчення мови програмування С++, важливо знати, як вона вписується в історичний контекст мов програмування. Розуміючи, що привело до її створення, які принципи розроблення вона представляє і що вона успадкувала від своїх попередників, студентам буде легше оцінити суть новаторства і унікальність засобів мови програмування С++. Саме тому у цьому розділі спробуємо зробити короткий екскурс в історію створення мови програмування С++, заглянути в її витоки, проаналізувати її взаємини з безпосереднім попередником, тобто мовою С, розглянути її області застосування та принципи програмування, які вона підтримує. Тут також студент дізнається, яке місце займає мова С++ серед інших мов програмування.

## 1.1. Витоки мови програмування С++

Історія розроблення мови програмування С++ починається з мови С, тобто її побудовано на фундаменті та синтаксисі мови С. Мова С++ насправді є надбудовою мови С, тобто усі компілятори С++-програм можна використовувати для компілювання С-програм. Мову С++ можна назвати розширеною та поліпшеною версією мови С, у якій реалізовано технологію об'єктно-орієнтованого, узагальненого та процедурного програмування. Вона також містить ряд інших удосконалень мови С, наприклад, розширений набір бібліотечних функцій. Щоб до кінця зрозуміти і оцінити переваги мови програмування С++, необхідно зрозуміти все "як" і "чому" відносно мови С.

### 1.1.1. Причини створення мови програмування С

Поява мови С потрясла комп'ютерний світ. Її вплив не можна було недооцінити, оскільки вона докорінно змінила підхід до методології програмування і його сприйняття. Мова С стала вважатися першою сучасною "мовою програміста", оскільки до її винаходу комп'ютерні мови в основному розроблялися або як навчальні вправи, або як результат діяльності науково-дослідних структур. З появою мови програмування С все пішло інакше. Вона була задумана і розроблена реальними програмістами-практиками, яка відображала їх підхід до методології програмування. Її синтаксис і засоби були багато разів продумані, відточені та протестовані користувачами, які дійсно працювали з нею. Як наслідок, появилася мова, яку сприйняли багато програмістів-практиків. Вона швидко знайшла багато прихильників, які мали про неї найкращі враження, що сприяло широкому її використанню різними програмістами. Загалом мову програмування С було розроблено фаховими програмістами-практиками для програмістів-аматорів. Саме це і зумовило її шалений успіх як серед професійних програмістів, так і серед аматорів.

Мову програмування С винайшов Деніс Рітчи (Dennis Ritchie) для комп'ютера PDP-11 (розробка компанії DEC – Digital Equipment Corporation), який працював під управлянням операційної системи (ОС) UNIX. Мова С – результат тривалого робочого процесу, який спочатку був пов'язаний з іншою мовою – BCPL, створеною Мартіном Річардсом (Martin Richards). Мова BCPL індукувала появу мови, що отримала початкову назву B (її автор – Кен Томпсон (Ken Thompson)), після чого на початку 70-х років вона привела до розроблення мови С.

Впродовж багатьох років стандартом для мови програмування С де-факто слугувала мова, яку підтримувала ОС UNIX і описана в книзі Брайана Кернігана (Brian Kernighan) і Деніса Рітчи *The C Programming Language* (Prentice-Hall, 1978). Проте формальна відсутність стандарту стала причиною розбіжностей між різними реалізаціями мови С. Щоб змінити ситуацію, на початку літа 1983 р. було започатковано комітет із створення ANSI-стандарту, покликаного – раз і назавжди визначити мову програмування С. Кінцеву версію цього стандарту було прийнято у грудні 1989 р., а його перше видання стало доступним для програмістів на початку 1990 р. Ця версія мови програмування С отримала назву С89, тому саме вона стала фундаментом для розроблення мови С++.

Мову програмування С часто називають комп'ютерною мовою середнього рівня. Однак це визначення не має негативного відтінку, оскільки воно зовсім не означає, що мова С є менш потужною та розвиненою (порівняно з іншими мовами "високого рівня") або її складно використовувати (подібно до мови Assembler[1]). Мова С належить до середнього рівня тільки тому, що вона поєднує елементи мов високого рівня, таких як Pascal, Modula-2 або Visual Basic з функціональними можливостями мови Assembler.

[1] Assembler – система програмування, яка містить саму мову і транслятор цієї мови. Вона є мовою програмування низького рівня. Чим нижчий рівень мови програмування, тим ближча специфіка роботи програми до самого процесора, для якого вона й була написана. Вважається, що мови низького рівня складніші від мов високого рівня й потребують більш вузької спеціалізації програміста, оскільки програма, написана мовою Assembler, для одного типу процесорів може виявитися не завжди придатною для роботи з іншими процесорами. З іншого боку програми, написані мовою Assembler, компактні та мають високу швидкодію, що теж є немаловажним при експлуатації різних програм.

З погляду теорії програмування, у *мову високого рівня* закладено прагнення створити програмісту максимум зручностей у вигляді вбудованих засобів. *Мова низького рівня* не забезпечує програміста нічим, окрім доступу до реальних машинних команд. *Мова середнього рівня* надає програмісту певний (невеликий) набір інструментів, які дають змогу йому самому розробляти конструкції програм вищого рівня. Іншими словами, мова середнього рівня пропонує програмісту вбудовані можливості в поєднанні з її гнучкістю.

Будучи мовою середнього рівня, мова С дає змогу маніпулювати бітами, байтами і адресами, тобто базовими елементами, з якими працює комп'ютер. Таким чином, у мові С не передбачено спроби відокремити апаратні засоби комп'ютера від програмних. Наприклад, розмір цілочисельного значення у мові С безпосередньо пов'язаний з розміром слова центрального процесора. У більшості мов високого рівня існують вбудовані настанови, призначені для зчитування і запису дискових файлів. У мові С всі ці процедури виконуються за допомогою виклику бібліотечних функцій, а не за допомогою ключових слів, визначених у самій мові. Такий підхід підвищує функціональну гнучкість мови програмування С.

Мова С дає змогу програмісту (правильніше сказати, стимулює його) визначати підпрограми для виконання операцій високого рівня. Ці підпрограми називаються *функціями*, які є "будівельними" блоками мови С. Програміст може без особливих зусиль створити бібліотеку функцій, призначених для виконання різних задач, які має використовувати його програма. У цьому сенсі програміст може персоніфікувати мову С відповідно до своїх потреб.

Необхідно згадати ще про один аспект мови С, дуже важливий і для мови програмування С++: С – *структурована мова*, найхарактернішою особливістю якої є використання блоків. *Блок* – набір настанов, які логічно пов'язані між собою. Наприклад, уявіть собі настанову **if**, яка, при успішній перевірці свого виразу, має виконати п'ять окремих настанов. А якщо ці настанови спеціально згрупувати і звертатися до них як до єдиного цілого, то така група утворює блок.

Структурована мова підтримує концепцію функцій і локальних змінних. *Локальна змінна* – звичайна змінна, яка відома тільки функції, у якій вона визначена. Структурована мова також підтримує ряд таких циклічних конструкцій, як **while**, **do-while** і **for**. Проте використання настанови **goto** або забороняється, або не рекомендується, а також не несе того змістового навантаження щодо передачі керування, яке властиве їй у таких мовах програмування, як Basic або Fortran. Структурована мова дає змогу формалізувати код програми, тобто робити відступи під час написання настанов, і не вимагає строгої прив'язки до полів, як це реалізовано в ранніх версіях мови Fortran.

Нарешті (і це, можливо, найважливіше), мова С не несе відповідальності за некоректні дії програміста. Основний принцип використання мови С полягає у тому, щоб дати змогу програмісту робити все те, що він захоче, але за наслідки роботи програми (нехай навіть дуже незвичайні або зовсім підозрілі) відповідають не засоби мови, а програміст. Мова С надає програмісту практично повну владу над розробленою ним програмою та комп'ютером,





який її виконує, але за наслідки такого володарювання він несе персональну відповідальність.

### 1.1.2. Передумови виникнення мови програмування С++

Наведена вище характеристика мови С може викликати справедливий подив: навіщо ж тоді, мовляв, було винайдено мову С++? Якщо мова програмування С – така хороша і корисна, то чому виникла потреба в чомусь ще новому? Виявляється, вся справа в складності написання досконалих кодів програм. Впродовж всієї історії розвитку програмування ускладнення кодів програм спонукало програмістів шукати шляхи, які б дали змогу справитися з тими труднощами, що виникали. Закладені можливості у мову програмування С++ можна вважати одним із способів їх подолання. Спробуємо краще розкрити цей взаємозв'язок.

Ставлення до процесу програмування різко змінилося з моменту винаходу персонального комп'ютера. Основна причина – прагнення "приборкати" все зростаючу складність написання кодів програм. Наприклад, програмування для перших обчислювальних машин полягало в перемиканні тумблерів на передній панелі так, щоб їх положення відповідали двійковим кодам машинних команд. Доки величини програм не перевищували декількох сотень команд, такий метод ще мав право на існування. Але у міру їх подальшого зростання було винайдено мову низького рівня Assembler, яка дала змогу програмістам використовувати символічне представлення машинних команд. Оскільки розміри програм продовжували зростати, то бажання справитися з вищим рівнем їх складності викликало згодом появу мов програмування високого рівня, розроблення та використання яких дало змогу програмістам значно більше інструментів – нових і різних.

Першою широко поширеною мовою програмування була, звичайно ж, мова Fortran. Незважаючи на те, що це був дуже значний крок на шляху прогресу у області програмування, Fortran все ж таки важко назвати тією мовою, яка сприяла написанню яскравих і простих для розуміння програм. Шістдесяті роки XX ст. вважаються періодом появи технології структурного програмування. Саме її і було реалізовано у мові С та багатьох інших мовах. За допомогою структурованих мов можна було розробляти програми середньої складності, до того ж без особливих зусиль з боку програміста. Але, якщо програмний проект досягав певного розміру, то навіть з використанням згаданих структурних методів його складність різко зростала і виявлялася непосильною для можливостей навіть професійних програмістів. Коли (до кінця 1970-х) до "критичної" точки підійшло достатньо багато проектів, почали появлятися нові технології програмування, одна з яких отримала назву *об'єктно-орієнтованого програмування*[1] (ООП). Опанувавши методи ООП, прог-

[1] Об'єктно-орієнтоване програмування – одна з парадигм програмування, яка розглядає програму як множину "об'єктів", що взаємодіють між собою. В ній використано декілька технологій від попередніх парадигм, зокрема успадкування, модульність, поліморфізм та інкапсуляцію. Не зважаючи на те, що ця парадигма з'явилася ще в 1960-их роках, вона не мала широкого застосування до 1990-их років. На сьогодні багато із мов програмування (зокрема, Java, C#, C++, Python, PHP, Ruby та Objective-C, ActionScript 3) підтримують ООП.

раміст міг справлятися з програмами набагато більшого розміру, ніж раніше. Але мова програмування С не підтримувала технології ООП. Прагнення отримати об'єктно-орієнтовану версію мови врешті-решт і привело до розроблення мови програмування С++.

Незважаючи на те, що мова С була однією з найулюбленіших і поширеніших професійних мов програмування, настав час, коли її можливості з написання складних програм досягли своєї межі. Бажання подолати цей бар'єр і допомогти програмісту легко справлятися зі ще складнішими структурами програмам – ось що стало основною причиною розроблення мови С++.

### 1.1.3. Поява мови програмування С++

Отже, мова програмування С++ з'явилася як відповідь на потребу подолати всезростаючу складність написання кодів програм складної структури. Вона була створена Б'ярном Страуструпом (Bjarne Stroustrup) в 1979 р. в компанії AT&T Bell Laboratories (м. Муррей-Хилл, шт. Нью-Джерси). Спочатку вона мала ім'я "С з класами" (C with Classes), але в 1983 р. її було перейменовано під назвою С++. У 1998 р. вона була стандартизована Міжнародною організацією стандартизації (МОС) під номером 14882:1998 – мова програмування С++. Згодом робоча група МОС почала працювати над новою версією стандарту під кодовою назвою С++09 (раніше відомою як С++0X), який мав вийти в 2009 р. під загальною назвою "ISO/IEC 14883(2009): Programming Language C++".

Стандарт мови програмування С++ станом на 1998 рік складався з двох основних частин: ядра мови і стандартної бібліотеки. Стандартна бібліотека С++ увібрала в себе бібліотеку шаблонів STL (Standard Template Library), що розроблялася одночасно із стандартом. Зараз назва STL офіційно не вживається, проте в колах програмістів, які програмують мовою С++, ця назва використовується для позначення частини стандартної бібліотеки, що містить визначення шаблонів, контейнерів, ітераторів[1], алгоритмів і функторів[2].

Стандарт мови програмування С++ містить нормативне посилання на стандарт мови С від 1990 р. і не визначає самостійно ті функції стандартної

ООП сягає своїм корінням до створення мови програмування Simula ще в 1960-их роках, одночасно з посиленням дискусій про кризу програмного забезпечення. Разом із тим, як з роками ускладнювалось апаратне та програмне забезпечення, було дуже важко зберегти якість написаних програм. ООП частково розв'язало цю проблему шляхом наголошення на модульності програми.

На відміну від традиційних поглядів, коли програму розглядали як набір підпрограм, або як перелік настанов комп'ютеру, об'єктно-орієнтований підхід до розроблення програми можна вважати сукупністю об'єктів. Відповідно до парадигми ООП, кожний об'єкт здатний отримувати повідомлення, обробляти дані, та надсилати повідомлення іншим об'єктам. Тобто об'єкт – своєрідний незалежний автомат з окремим призначенням та відповідальністю.

[1] Ітератор (англ. Iterator) – шаблон проектування, належить до класу шаблонів поведінки. Надає спосіб послідовного доступу до всіх елементів складеного об'єкта, не розкриваючи його внутрішнього улаштування.

[2] Функтор – збиральний термін, який означає всі види сутностей у мові програмування С++, до яких застосовується оператор виклику функції – '()'. Функтори діляться на два підкласи: 1) звичайні функції (покажчики на звичайні функції), до яких застосовується вбудований оператор '()'; 2) функціональні об'єкти – об'єкти класів з перевизначеним оператором '()'. Інколи під терміном "функтор" розуміють тільки функціональні об'єкти. Це, строго кажучи, некоректне вживання цього терміну. Звичайні функції та покажчики на них теж є окремими випадками функторів.





бібліотеки, які було запозичено із стандартної бібліотеки мови С. Поза цим, існує величезна кількість бібліотек мови С++, котрі не входять в її стандарт, тобто можна успішно використовувати різні бібліотеки і з мови С.

Стандартизація визначила специфікацію та синтаксис мови програмування С++, проте за цією назвою можуть ховатися також неповні, обмежені до-стандартні варіанти мови. Спочатку мова розвивалася поза формальними рамками, спонтанно, у міру надходження тих завдань, що ставилися перед нею. Розвиток мови супроводив розвиток крос-компілятора Cfront. Нововведення в мові відбивалися в зміні номера версії крос-компілятора, які розповсюджувалися і на саму мову. Проте але, стосовно теперішнього часу, то нумерація версії мови програмування С++ не ведеться.

Мова програмування С++ багато в чому є надбудовою мови С. Мова С++ містить такі нові можливості як: оголошення у вигляді виразів, перетворення типів у вигляді функцій, динамічні оператори **new** і **delete**, тип **bool**, посилання, розширене поняття константності та змінності, вбудовані функції, аргументи за замовчуванням, перевизначення функцій, простори імен, класи (включаючи і всі пов'язані з класами можливості, такі як успадкування, функції-члени (методи), віртуальні функції, абстрактні класи і конструктори), перевизначення операторів, шаблони, оператор '::', оброблення винятків, динамічну ідентифікацію і багато що інше. Мова програмування С++ є також мовою строгого типування даних і накладає більше вимог щодо дотримання типів даних, порівняно з мовою С.

У мові програмування С++ з'явилися коментарі у вигляді подвійної косої риски ("//"), які були в попереднику мови С – мові BCPL. Деякі особливості мова програмування С++ пізніше були перенесені в мову С, наприклад ключові слова **const** та **inline**, оголошення в циклах **for** і коментарі в стилі С++ ("//"). У пізніших реалізаціях мови програмування С також були представлені можливості, яких немає в мові С++, наприклад макроси vararg і покращена робота з масивами-параметрами.

Отож, мова програмування С++ повністю містить всі особливості мови С, всі її засоби і атрибути, володіє всіма її перевагами. Для неї також залишається у силі принцип функціонування мови С, згідно з яким програміст, а не мова, несе власну відповідальність за результати роботи своєї програми. Саме цей момент дає змогу зрозуміти, що процес розроблення мови С++ не був спробою створити нову мову програмування. Це було мабуть удосконалення вже наявної (і при цьому дуже успішної) мови.

Більшість нововведень, якими Б'ярн Страуструп збагатив мову С, було використано для підтримки технології ООП. По суті мова С++ стала об'єктно-орієнтованою версією мови С. Взявши мову С за основу, Б'ярн Страуструп підготував плавний перехід до ООП. Тепер, замість того, щоби вивчати абсолютно нову мову, С-програмісту достатньо було засвоїти тільки ряд нових засобів, і він міг пожинати плоди використання технології ООП. Проте в основу мови програмування С++ лягла не тільки мова С. Б'ярн Страуструп стверджує, що деякі об'єктно-орієнтовані засоби були інспіровані іншою об'єктно-орієнтованою мовою, а саме Simula67. Таким чином, мова С++ є симбіозом двох могутніх технологій програмування.

Створюючи мову програмування С++, Б'ярн Страуструп розумів, наскільки важливо, зберігши початкову суть мови С, тобто її ефективність, функціональна гнучкість і принципи розроблення програм, внести в неї підтримку технології ООП. На щастя, ця мета була досягнута. Мова програмування С++, як і раніше, надає програмісту свободу дій і владу над комп'ютером (які були властиві мові С), розширюючи при цьому його (програміста) можливості за рахунок використання об'єктів.

Хоча мова програмування С++ спочатку була спрямована на підтримку дуже великих програм, проте тільки цим її використання не обмежилося. Виявляється об'єктно-орієнтовані засобу мови програмування С++ можна ефективно застосовувати практично до будь-якого завдання програмування. Не дивно, що мову програмування С++ часто використовують для створення компіляторів, редакторів, комп'ютерних ігор і програм мережевого обслуговування. Оскільки мова С++ володіє ефективністю мови С, то програмне забезпечення багатьох високоефективних систем побудоване з використанням мови С++. Окрім цього, мова програмування С++ – мова, яка найчастіше вибирається для Windows-програмування.

***Вартоознати!*** *Оскільки мова С++ є надбудовою мови С, то, навчившись програмувати мовою С++, студент зможе також програмувати і мовою С. Таким чином, доклавши деякі зусилля до вивчення тільки однієї мови програмування, студент насправді вивчить відразу дві мови.*

### 1.1.4. Етапи вдосконалення мови програмування С++

Історія розвитку мови програмування С++ містить такі ключові події:

- квітень 1979 – початок роботи над мовою С з класами (C with Classes);
- жовтень 1979 – робоча версія мови С з класами (Cpre);
- серпень 1983 – назва мови програмування С++ вперше використовується в компанії AT&T Bell Laboratories;
- 1984 – затверджена офіційна назва мови програмування С++;
- лютий 1985 – перший зовнішній випуск мови програмування С++ – Cfront Relese E (Educational – випуск для навчальних закладів);
- жовтень 1985 – перший комерційний випуск – Cfront 1.0;
- лютий 1987 – другий комерційний випуск – Cfront 1.2;
- грудень 1987 – перший випуск GNU C++ (1.13);
- 1988 – перші випуски Oregon Software C++ і Zortech C++;
- червень 1989 – комерційний випуск Cfront 2.0;
- 1989 – вийшла у світ книга "The Annotated C++ Reference Manual" (ARM); засновано комітет ANSI C++;
- 1990 – перша технічна зустріч комітету ANSI C++; прийнято шаблони (**templates**), виняткові ситуації (**exceptions**); перший випуск Borland C++;
- 1991 – перша зустріч ISO; комерційний випуск Cfront 3.0 (з шаблонами); книга "The C++ Programming Language" (2-га редакція);
- 1992 – перші випуски IBM, DEC, Microsoft C++;
- 1993 – RTTI (Run-time type identification – визначення типу даних під час виконання) прийнято; простори назв (**namespaces**) і **string** (шаблонний за символьним типом) прийнято;





- 1994 – прийнято STL – Стандартна Бібліотека Шаблонів;
- 1996 – прийнято **export**;
- 1997 – остаточне голосування комітету за завершений стандарт С++;
- 1998 – ратифіковано стандарт ISO C++;
- 2003 – технічні поправки до стандарту; початок роботи над С++0x;
- 2005 – перше голосування за можливості стандарту С++0x; auto, static_assert, rvalue references прийняті в загальному використанні;
- 2006 – перше офіційне голосування за стандарт С++0x.

З моменту винаходу мова програмування С++ зазнала три значних вдосконалення, тобто щоразу вона як доповнювалася новими засобами, так і в чомусь змінювалася. Першій ревізії мова С++ була піддана в 1985 р., а другої вона зазнала в 1990 р. Третя ревізія відбулася в процесі її стандартизації, яка активізувалася на початку 1990-х. Спеціально для цього був сформований об'єднаний ANSI/ISO-комітет, який 25 січня 1994 р. прийняв перший проект запропонованого на розгляд стандарту. У цей проект були внесені всі засоби, вперше визначені Б'ярном Страуструпом, і додано нові. Але в цілому він відображав стан мови програмування С++ на той момент часу.

Незабаром, після завершення роботи над першим проектом стандарту мови програмування С++, відбулася подія, яка примусила значно розширити наявний стандарт. Йдеться про створення Стандартної Бібліотеки Шаблонів (Standard Template Library – бібліотека STL). Як буде сказано пізніше, бібліотека STL – набір узагальнених функцій, які можна використовувати для оброблення даних. Вона достатньо велика за розміром. Комітет ANSI/ISO проголосував за внесення бібліотеки STL у специфікацію мови програмування С++. Додавання бібліотеки STL розширило сферу застосування засобів мови С++ далеко за межами початкового її визначення.

Необхідно зазначити, що стандартна бібліотека мови програмування С++ містить стандартну бібліотеку мови С з невеликими змінами, які роблять її більш відповідною мові С++. Інша велика частина бібліотеки С++ базується на бібліотеці STL. Вона надає такі важливі інструменти, як контейнери (наприклад, вектори і списки) і ітератори (узагальнені покажчики), що надають доступ до цих контейнерів як до масивів. Окрім цього, бібліотека STL дає змогу аналогічно працювати і з іншими типами контейнерів, наприклад, асоціативними списками, стеками, чергами. Використовуючи шаблони, можна писати узагальнені алгоритми, здатні працювати з будь-якими контейнерами або послідовностями, доступ до членів яких забезпечують ітератори. Так само, як і в мові програмування С, можливості бібліотек активізуються використанням директиви #include для включення стандартних файлів. Всього в стандарті мови програмування С++ визначено 50 таких файлів.

Бібліотека STL до внесення її в стандарт мови програмування С++ була сторонньою розробкою, на початку – фірми HP, а потім SGI. Стандарт мови не називає її "STL", оскільки ця бібліотека стала невід'ємною частиною мови, проте багато програмістів до цих пір використовують цю назву, щоб відрізняти її від решти частини стандартної бібліотеки (потоки введення/виведення (Iostream), підрозділ мови С тощо). Проект під назвою STLport, який базуєть-

ся на SGI STL, здійснює постійне оновлення STL, IOstream і рядкових класів. Деякі інші проекти також займаються розробленням приватних додатків стандартної бібліотеки для різних конструкторських завдань. Кожен виробник компіляторів мови програмування C++ обов'язково поставляє якусь реалізацію цієї бібліотеки, оскільки вона є дуже важливою частиною стандарту і широко використовується програмування.

Причиною успіху бібліотеки STL, зокрема її вхід до стандартної бібліотеки мови програмування C++, була направленість на широке коло виконуваних завдань і узагальнена структура. В цьому сенсі, близькою за духом бібліотеки STL, на сьогодні є бібліотека Boost, яка теж є бібліотекою загального застосування і теж впливає на формування стандартної бібліотеки мови C++.

Проте внесення бібліотеки STL, окрім іншого, уповільнило процес стандартизації мови програмування C++, причому достатньо істотно. Окрім бібліотеки STL, в саму мову програмування було додано декілька нових засобів і внесено багато дрібних змін. Тому версія мови C++ після розгляду комітетом із стандартизації стала набагато більша і складніша порівняно з початковим варіантом Б'ярна Страуструпа. Кінцевий результат роботи комітету датується 14 листопада 1997 р., а реально ANSI/ISO-стандарт мови C++ побачив світ у 1998 р. Саме ця специфікація мови програмування C++ зазвичай називається *Standard C++*, саме вона описана у цьому посібнику. Ця версія мови програмування C++ підтримується всіма основними C++-компіляторами, в т.ч. Visual C++ (Microsoft) і C++ Builder (Borland). Тому коди програм, наведені у цьому посібнику, повністю придатні для застосування в усіх сучасних C++-середовищах.

Мова програмування C++ продовжує розвиватися, щоб відповідати сучасним вимогам. Одна з науково-дослідних груп, що займається розвитком мови C++ в її сучасному вигляді і яка направляє комітету зі стандартизації поради щодо її поліпшення, – група Boost. Наприклад, один з напрямів діяльності цієї групи – вдосконалення можливостей мови шляхом залучення до неї особливостей метапрограмування[1] – одна з сучасних мало використовуваних технологій програмування, яка полягає в написанні міні-програм, що генерують програми або частини програм.

[1] Метапрограмування – технологія створення міні-програм, які генерують код, має численні застосування:
1) при написанні програм, які будуть заздалегідь генерувати таблиці даних для використання їх під час виконання інших програми. Наприклад, при потребі виконувати швидкий пошук у таблиці синусів для всіх 8-бітових цілих чисел, то можна або обчислювати кожен синус і закодувати це так, щоб програма будувала таблицю при її запуску, або написати програму для створення спеціалізованого коду для таблиці синусів перед її компілюванням. Хоча для такого маленького набору чисел може мати сенс створення таблиці синусів під час виконання програми, інші подібні завдання можуть значно уповільнити тривалість її запуску. У таких випадках написання програми побудови статичних даних зазвичай є найкращим варіантом.
2) при розробленні великого додатку, у якому множина функцій містить довгий стереотипний фрагмент коду, то можна розробити міні-код, який буде генерувати стереотипний код замість Вас і дасть Вам змогу кодувати тільки важливі частини програми. Тут, за змогою, найкраще абстрагувати стереотипні фрагменти у просту функцію. Але часто ці фрагменти не настільки приємні. Можливо є список змінних, які потрібно оголосити в кожному його примірнику, можливо є потреба зареєструвати обробники помилок, можливо існує декілька стереотипних фрагментів, які мають містити код за певних обставин. Все це робить створення простої функції неможливим. Тому в таких ситуаціях гарною ідеєю є створення міні-коду, який дає змогу Вам працювати з ним більш простим способом. Цей міні-код потім конвертується в початковий код, написаний звичайною мовою програмування перед її компілюванням.
3) багато мов програмування змушують Вас писати багатослівні оператори для виконання простих завдань. Програми, які генерують код, дають змогу Вам скоротити ці оператори і зберегти багато часу, витраченого на набір коду, що також охороняє від багатьох помилок через зменшення шансу їх виникнення.





Група Boost має помітну спрямованість на дослідження та розширення як технології метапрограмування, так і узагальненого програмування з активним використанням шаблонів. Завдяки ретельному підбору і контролю якості стандартні бібліотеки шаблонів, внесені в бібліотеку Boost, мають високу надійність та продуктивність роботи. Думки щодо їх ефективного використання серед різних програмістів мають майже протилежні спрямування. Деякі вважають їх стандартом де-факто і необхідним доповненням до бібліотеки STL. Деякі, навпаки, уникають всякого використання бібліотеки шаблонів у проектах, оскільки це зайва залежність в проекті. Окрім цього, для успішного використання цих бібліотек програмістові необхідно добре знати мову програмування C++, оскільки деякі частини бібліотеки Boost вимагають достатньо хорошої його підготовки і є вельми складними у використанні.

Стандарт мови програмування C++ не описує способи іменування об'єктів, деякі деталі оброблення винятків і інші можливості, пов'язані з деталями реалізації, що робить несумісним об'єктний код, створений різними компіляторами. Проте для цього третіми особами створено багато стандартів для конкретної архітектури і операційних систем. Однак серед компіляторів мови програмування C++ все ще продовжується битва за повну реалізацію стандарту мови C++, особливо в області шаблонів – частини мови, яку зовсім недавно було повністю розроблено комітетом стандартизації.

Одним із каменів спотикання у цьому питанні є ключове слово **export**, яке використовується також і для розділення оголошення і визначення шаблонів. Першим компілятором, що підтримав слово **export** в шаблонах, став Comeau C++ на початку 2003 р. (тобто п'ять років після виходу стандарту). У 2004 р. бета-версія компілятора Borland C++ Builder X також почала його підтримку. Обидва цих компілятора базуються на зовнішньому інтерфейсі EDG. Інші компілятори, такі як Microsoft Visual C++ або GCC, взагалі цього не підтримують. Ерб Саттер (Herb Sutter), секретар комітету із стандартизації мови програмування C++, рекомендував прибрати слово **export** з майбутніх версіях стандарту унаслідок серйозних складнощів у повноцінній реалізації, проте згодом остаточним рішенням було вирішено його залишити.

Із переліку інших проблем, пов'язаних з шаблонами, можна навести питання конструкцій часткової спеціалізації шаблонів, які погано підтримувалися протягом багатьох років після виходу стандарту C++.

## 1.2. Поняття про технологію структурного програмування

Для полегшення роботи зі складними завданнями процес підготовки задачі для її розв'язання можна розділити на два етапи: створення укрупненого

У міру набуття деякою мовою додаткових можливостей, що генерують код програми, вони стають менш привабливими. Те, що є доступним як стандартна можливість в одній мові, може бути доступною в іншій мові тільки при використанні програми, які генерують код. Однак, неадекватний дизайн мови є не єдиною причиною потреби в програмах, що генерують код. Важливе значення тут має ще й більш легке її обслуговування.

алгоритму (вимоги до початкових даних і очікуваний результат, постановка задачі, опис точної схеми розв'язання задачі з вказанням усіх особливих ситуацій) і написання програми. Розробляючи програми для розв'язання складних задач, на різних етапах розвитку програмісти застосовували різні технології програмування, наприклад, низхідне програмування ("зверху вниз"), висхідне програмування ("знизу вгору"), структурне та об'єктно-орієнтоване програмування, узагальнене та процедурне програмування.

Опускаючи низхідне та висхідне програмування, дещо детальніше зупинимося на структурному підході до розроблення складних програмних продуктів, яке нам знадобиться при вивченні мови програмування C++.

### 1.2.1. Структурний підхід до проектування програм

За часів стихійного програмування хорошими програмістами вважалися ті, хто створював достатньо хитромудрі програми, які займали мінімальний об'єм пам'яті та швидко виконувалися. Це було цілком природно, враховуючи тодішні можливості обчислювальної техніки. Результатом такого програмування появлялися програми, які було важко (якщо взагалі можливо) зрозуміти іншим фахівцям. Інколи навіть автори таких програм з плином часу з трудом розуміли власне творіння. Внесення необхідних змін у таку програму робило ситуацію ще більш заплутаною, ніж вона була до цього. Подібні проекти отримали назву BS-програм – абревіатура від "bowl of spaghetti" – блюдо спагеті, бо саме так виглядала готова програма при спробі зобразити всі переходи між її операторами.

Винахідник структурного програмування Е. Дейкстра[1] навіть проголосив, що "кваліфікація програміста обернено пропорційна кількості операторів безумовного переходу в його програмах". Структурне програмування ще тоді називали "програмуванням без **goto**", хоча для багатьох це була екстремальна точка зору. Насправді йдеться про те, щоб не використовувати оператори переходу без особливої потреби. Насамперед структурне програмування мало своєю метою позбавитись від поганої структури самої програми. Ще однією метою було створення таких програм, які були б легко зрозумілими навіть без їх авторів, адже "програми пишуться для людей – комп'ютером вони лише обробляються". Зміст цієї фрази полягає у тому, що трансляція та виконання програми будь-якої структури за допомогою комп'ютера дійсно не викликає ніяких труднощів. А от трудомісткий процес перевірки правильності роботи програми, внесення відповідних виправлень і логічних змін доводиться виконувати людині.

[1] Едсгер Вібе Дейкстра (нідерл. Edsger Wybe Dijkstra, народився 11 травня 1930 р., м. Роттердам, Нідерланди – помер 6 серпня 2002, м. Нуєнен, Нідерланди) – нідерландський учений, ідеї якого зробили вплив на розвиток комп'ютерної індустрії. Популярність Дейкстри принесли його роботи в області застосування математичної логіки при розробленні комп'ютерних програм. Він активно брав участь в розробленні мови програмування Алгол і написав перший компілятор Алгол-60. Будучи одним з авторів концепції структурного програмування, він "проповідував" відмову від використання настанови **goto**. Також йому належить ідея застосування "семафорів" для синхронізації процесів у багатозадачних системах і алгоритм знаходження найкоротший шлях на орієнтованому графові з не негативними вагами ребер, відомий як Алгоритм Дейкстри. У 1972 р. Дейкстра став лауреатом премії Т'юринга.





Отож, структурне програмування – методологія й технологія розроблення серйозних програмних проектів, яка об'єднує способи розроблення структури програми, зручної для читання та розуміння людиною, стеження за логікою її роботи, внесення до неї виправлень та інших змін. Згідно з думкою Н. Вірта[1], "структуризація є принциповим інструментом, яка допомагає програмісту систематично аналізувати і синтезувати складні програми, зберігаючи про них повне уявлення". Ідеї структурного програмування з'явилися на початку 1970-х роках у компанії IBM, у її розробленні брали участь відомі вчені Е. Дейкстра, Х. Милі, Є. Батіг, С. Хоор.

Структурне програмування базується на таких основних принципах:

- програмування має здійснюватися зверху-униз;
- вся програма розв'язання складної задачі має бути поділена на модулі з одним входом і одним виходом (оптимальний розмір модуля – кількість рядків, що поміщається на екрані дисплея);
- логіка алгоритму й програми має допускати тільки три основні структури: послідовне виконання, розгалуження й повторення. Недопустимий оператор передачі керування в будь-яку точку програми;
- при розробленні коду програми супровідна документація має створюватися одночасно із програмуванням, у вигляді коментарів до неї.

З точки зору структурного програмування, правильна програма – програма, структура якої містить тільки базові елементи, і жоден з них не є недоступним і не допускає так зване зациклювання. Правильна програма має тільки один вхід і тільки один вихід. У правильній програмі не має бути таких частин, які ніколи не виконуються.

### 1.2.2. Принцип поділу програми на окремі модулі

Принцип поділу програми на окремі модулі (у колах програмістів його називають принципом модульності програми) полягає у тому, що будь-яку складну програму доцільно розділити на логічно незалежні частини (модулі), дотримуючись при цьому певних зв'язків між ними. Історично поняття модульної програми виникло раніше, ніж були сформульовані принципи структурного програмування, проте ця ідея виявилася просто необхідною складовою нової технології програмування разом з аналітичним проектуванням.

Модуль – послідовність логічно пов'язаних команд, який оформлено у вигляді окремої програми. Ці допоміжні програми можна розробляти й аналізувати окремо та незалежно одну від іншої, використовуючи їх потім у основній програмі або інших допоміжних програмах. Структурний підхід до розроблення програм і принцип її модульності також привів до ідеї розподілу робіт серед розробників програм.

Поняття модуля цілком логічно з'явилося на відповідному етапі аналітичного програмування: модуль – частина програми, яка розв'язує порівняно

[1] Ніклаус Вірт (нім. Niklaus Wirth, народився 15 лютого 1934 р.) – швейцарський учений, фахівець в області інформатики, один з відомих теоретиків у області розроблення мов програмування Pascal, Modula-2, Oberon, професор комп'ютерних наук (ETH), Лауреат премії Т'юринга 1984 р.

нескладну задачу, логічно незалежну від інших задач. Здебільшого, модулі – підпрограми (процедури або функції), які мають певні властивості:

1) єдиний вхід, єдиний вихід (деякі мови дають змогу існування декількох входів або виходів);
2) окреме компілювання кожного модуля;
3) кожний модуль доступний за своїм ідентифікатором;
4) модуль може викликати інший модуль;
5) модуль не має зберігати історію своїх викликів (інакше може виникати так званий побічний ефект);
6) модуль має бути невеликим порівняно зі всією програмою;
7) кожен модуль відповідає за виконання тільки однієї задачі;
8) незалежність функціонування (заміна модуля на аналогічний не впливає на роботу всієї програми).

З часом, коли принцип модульності став підтримуватись різними мовами програмування, на перший план висунулась вимога логічної та програмної незалежності модулів. Необхідно зауважити, що повної незалежності між модулями бути не може. Залежність між модулями існує тоді, коли:

- використовуються спільні списки параметрів;
- вони користуються спільними (глобальними) змінними;
- модулі програми залежать від структури даних цієї програми;
- модулі програми залежать від логіки функціонування програми (від того фактору, що модуль може викликатися іншими модулями).

Отже, модулі мають бути незалежні в межах інтерфейсу програми і структури даних. Практика програмування показала, що чим вищий ступінь незалежності модулів, тим простіше розібратись в окремих модулях і в самій програмі загалом; тим менша ймовірність появи нових помилок при виправленні старих, або внесенні змін у програму, тобто менша ймовірність так званого хвильового ефекту. Із сказаного вище випливає, що не варто без крайньої на те потреби використовувати в модулях глобальні змінні. Всі зв'язки між модулями мають підтримуватися через списки параметрів.

### 1.2.3. Методологія покрокової деталізації програми

При створенні програми розв'язання складної задачі застосовується методологія покрокової її деталізації ("проектування зверху вниз"). При цьому складна задача ділиться на декілька простих, які, водночас, можуть ділитися на серію ще простіших і т.д. Процес покрокової деталізації вважається завершеним, коли задачі чергового рівня стають достатньо простими для незалежного їх розв'язання. Потім результати програмування простих задач компонуються в загальний алгоритм.

Технологія структурного програмування часто вимагає розв'язання різних додаткових задач. Програму для виконання основної задачі назвемо основною програмою. Тоді програма для розв'язання допоміжної задачі, виділеної в окрему структуру, називатиметься допоміжною програмою. Вона призначена для досягнення деякої допоміжної мети на шляху розв'язання основної задачі. Одна і та сама допоміжна програма може бути використана і в





основній, і в іншій допоміжній програмі. Кожна допоміжна програма має мати своє унікальне ім'я, за яким її розпізнають серед інших програм і за цим іменем вона викликається з будь-якої іншої програми.

Виконання допоміжної програми з потрібними початковими даними та/або передавання результатів розв'язання в основну програму здійснюється за допомогою параметрів програми. Параметри, які необхідно вказати перед початком роботи допоміжної програми, називаються аргументами програми. Ці параметри дають змогу змінити вхідні дані перед початком роботи допоміжної програми. Параметри, значення яких визначаються внаслідок роботи програми, називаються результатами роботи програми. Допоміжна програма може зовсім не мати параметрів або мати якийсь один тип параметрів. Параметри, які використовуються для опису вхідних і вихідних даних програми, називаються формальними. При конкретному виконанні допоміжної програми формальні параметри замінюються на фактичні, тобто під час виклику допоміжної програми потрібно вказати її фактичні параметри.

Результатом виконання деякої допоміжної програми може бути одна або декілька результуючих величин, які містяться в списку параметрів, або деяка дія (наприклад виведення повідомлення чи відтворення звуку) без результуючої величини. Такі допоміжні програми називаються програмами-процедурами. Програми-функції – допоміжні програми, які виконують деяку послідовність різних дії, результатом виконання яких є один-єдиний результат, що, зазвичай, присвоюється безпосередньо імені функції. Ім'я функції використовується в командах як ім'я звичайної змінної або деякої константи.

Програми-процедури і програми-функції забезпечують практичну реалізацію методології структурного програмування. Виклик допоміжної програми має різний вигляд: виклик допоміжної програми-процедури здійснюється за допомогою спеціальної команди; виклик програми-функції відбувається безпосереднім вказанням імені функції з фактичними аргументами в деякому арифметичному чи логічному виразі.

Виконання цих команд відбувається так:

- формальні аргументи допоміжної програми замінюються фактичними значеннями, вказаними в команді виклику програми в списку фактичних параметрів;
- якщо в списку фактичних аргументів присутні математичні чи логічні вирази – то вони спочатку обчислюються;
- виконуються всі команди допоміжної програми з використанням фактичних аргументів;
- отримані результати присвоюються фактичним іменам результатів (іменам фактичних змінних, які використовуються як фактичні результати в програмах-процедурах або імені самої програми-функції).

Після виконання допоміжної програми виконується настанова основної програми, записана після настанови виклику допоміжної програми.

## 1.3. Основні ознаки об'єктно-орієнтованого програмування

На думку Алана Кея[1], розробника мови програмування Smalltalk[2], якого вважають одним з "батьків-засновників" ООП, об'єктно-орієнтований підхід до розроблення сучасних програмних продуктів полягає в такому наборі основних принципів:

- все можна представити у вигляді об'єктів;
- всі дії та розрахунки виконуються шляхом взаємодії (обміну даними) між об'єктами, при якій один об'єкт потребує, щоб інший об'єкт виконав деяку дію. Об'єкти взаємодіють між собою, надсилаючи і отримуючи повідомлення. Повідомлення – запит на виконання дії, доповнений набором аргументів, які можуть знадобитися при її виконанні;
- кожен об'єкт має незалежну пам'ять, яка складається з інших об'єктів;
- кожен об'єкт є представником (екземпляром, примірником) класу, який виражає загальні властивості об'єктів;
- у класі задається поведінка (функціональність) об'єкта, тобто усі об'єкти, які є екземплярами одного класу, можуть виконувати одні й ті ж самі дії;
- класи можуть бути організовані у єдину деревоподібну структуру з загальним корінням, яка називається ієрархією успадкування;
- стан пам'яті та поведінка об'єкта, зв'язані з екземплярами деякого класу, стають доступні будь-якому класу, розташованому нижче в ієрархічному дереві.

Таким чином, програма складається з набору об'єктів, кожен з яких характеризується станом пам'яті та поведінкою. Об'єкти взаємодіють між собою, використовуючи для цього повідомлення. Будується ієрархія об'єктів: програма загалом – об'єкт; для виконання своїх функцій вона звертається до інших об'єктів, що містяться у ньому, які водночас виконують запити шляхом звернення до інших об'єктів програми. Звісно, щоб уникнути безкінечної рекурсії у зверненнях, на якомусь етапі об'єкт трансформує запит у повідомлення до стандартних системних об'єктів, що даються мовою та середовищем програмування. Стійкість та керованість системи забезпечуються за рахунок чіткого розподілу відповідальності об'єктів (за кожну дію відповідає певний об'єкт), однозначного визначення інтерфейсів міжоб'єктної взаємодії та повної ізольованості внутрішньої структури об'єкта від зовнішнього середовища (інкапсуляції).

[1] Алан Кьортіс Кей – американський фахівець з інформаційних технологій, відомий завдяки своїм новаторським роботам у галузі об'єктно-орієнтованого програмування, проектуванню віконного графічного інтерфейсу користувача, а також завдяки відомій фразі: "Найкращий спосіб спрогнозувати майбутнє – винайти його". Президент дослідницького інституту Вест-Поінта, ад'юнкт-професор інформатики в Каліфорнійському університеті. Також був радником у TTI/Vanguard. До середини 2005 р. був головним співробітником в HP Labs, читав лекції в Кіотському університеті та був ад'юнкт-професором у Масачусетському технологічному інституті (англ. Massachusetts Institute of Technology або MIT).

[2] Smalltalk (вимовляється [смо́лток]) – об'єктно-орієнтована мова програмування з динамічною типізацією, розроблена в Xerox PARC Аланом Кеєм, Деном Інгаллсом, Тедом Кеглером, Адель Голдбергом і іншими в 1970-х роках. Мова була представлена як Smalltalk-80 і з тих пір широко використовується різними програмними компаніями. Вона ще й сьогодні продовжує активно розвиватися і збирає навколо себе співтовариство користувачів. Мова Smalltalk істотно вплинула на розвиток багатьох інших мов, таких як: OBJECTIVE-C, Actor, Java та Ruby. Багато ідей 1980-х і 1990-х із написання програм з'явилися в співтоваристві Smalltalk. До них можна віднести рефакторінг, шаблони проектування (стосовно ПЗ), карти Класс-Обов'язки-Взаємодія і екстремальне програмування загалом. Засновник концепції Wiki, Вард Каннінгам, також входить в співтовариство Smalltalk.





Оскільки саме принципи ООП були основоположними при розробленні мови C++, то важливо точно визначити, що вони собою представляють. Отож ООП об'єднало кращі ідеї структурного програмування з рядом потужних концепцій, які сприяють ефективній організації програм. *Об'єктно-орієнтований підхід до програмування* дає змогу розкласти задачу на складові частини так, що кожна частина стане самостійним об'єктом, який містить власні дані та методи для роботи на ними. При такому підході істотно знижується загальний рівень складності написання кодів програм, що дає змогу програмісту справлятися зі значно складнішими програмами, ніж це можна було робити раніше (тобто при використання структурного програмування).

Всі мови ООП характеризуються трьома основними ознаками: інкапсуляцією, поліморфізмом і успадкуванням. Розглянемо стисло кожну з них.

### 1.3.1. Поняття про механізм реалізації програм методом інкапсуляція

Як відомо, усі програми, як правило, складаються з двох основних елементів: методів (кодів програм) і даних. *Код* – частина програми, яка здійснює дії, а *дані* є інформацією, на яку спрямовані ці дії. *Інкапсуляція* – такий механізм програмування, який пов'язує воєдино програмні коди і дані, які вони обробляють, щоб забезпечити їх як від зовнішнього втручання, так і від неправильного використання.

У об'єктно-орієнтованій мові програмування коди і дані можуть пов'язуватися між собою так, що створюється самостійний *чорний ящик*. У цьому "ящику" містяться всі необхідні (для забезпечення самостійності) дані та коди. При такому взаємозв'язку кодів програми і даних створюється об'єкт, тобто *об'єкт* – конструкція, яка підтримує інкапсуляцію.

Усередині об'єкта програмні коди, дані або обидві ці складові можуть бути не тільки відкритими (**public**) або закритими (**private**), але й захищеним (**protected**). *Закритий* програмний код (або дані) відомий і доступний тільки іншим частинам того ж об'єкта, тобто, до нього не може отримати доступ та частина програми, яка існує поза цим об'єктом. *Відкритий* програмний код (або дані) доступний будь-яким іншим частинам програми, навіть якщо вони визначені в інших об'єктах. *Захищений* програмний код (або дані) буде доступним для інших елементів програми, які не є членами даного об'єкта, за винятком того, що доступ до захищеного члена є ідентичним доступу до закритого члена, тобто до нього можуть звертатися тільки інші члени того ж самого об'єкта. Зазвичай відкриті частини об'єкта використовують для надання керованого інтерфейсу із закритими елементами об'єкта.

### 1.3.2. Поняття про властивість поліморфізму

*Поліморфізм* (від грецького слова *polymorphism*, що означає "багато форм") – властивість, яка дає змогу використовувати один інтерфейс для цілого класу дій. Конкретна дія визначається характерними ознаками ситуації. Як простий приклад поліморфізму можна навести кермо автомобіля. Для кер-

ма (тобто інтерфейсу) байдуже, який тип рульового механізму використовується в автомобілі. Тобто, кермо працює однаково, незалежно від того, чи оснащений автомобіль рульовим керуванням прямої дії (без підсилювача), рульовим керуванням з підсилювачем або механізмом рейкової передачі. Якщо Ви знаєте, як поводитися з кермом, то Ви зможете вести автомобіль будь-якого типу. Цей же принцип можна застосувати до програмування. Розглянемо, наприклад, стек, або список, додавання елементів до якого чи видалення з нього здійснюється за принципом "останній прибув – перший обслужений". У Вас може бути програма, у якій використовуються три різних типи стека. Один стек призначений для цілочисельних значень, другий – для значень з плинною крапкою і третій – для символів. Алгоритм реалізації всіх стеків – один і той самий, хоча у них зберігаються дані різних типів. У необ'єктно-орієнтованій мові програмісту довелося б створити три різні набори підпрограм обслуговування елементів стека, причому підпрограми мали б мати різні імена, а кожен набір – власний інтерфейс. Але завдяки поліморфізму у мові C++ можна створити один загальний набір підпрограм, який підходить для всіх трьох конкретних ситуацій. Таким чином, знаючи, як використовувати один стек, програміст може використовувати й усі інші.

У більш загальному вигляді концепція поліморфізму виражається фразою "один інтерфейс – багато методів". Це означає, що для групи взаємопов'язаних дій можна використовувати один узагальнений інтерфейс. Поліморфізм дає змогу знизити рівень складності за рахунок можливості застосування одного і того ж інтерфейсу для виконання цілого класу дій. Вибір же *конкретної дії* (тобто функції) стосовно тієї або іншої ситуації лягає "на плечі" компілятора. Вам, як програмісту, не потрібно робити цей вибір вручну. Ваше завдання – використовувати загальний інтерфейс.

Перші мови ООП були реалізовані у вигляді інтерпретаторів, тому поліморфізм підтримувався у процесі виконання програм. Проте мова програмування C++ – трансльована мова (на відміну від того, що інтерпретується). Отже, у мові C++ поліморфізм підтримується на рівні як компілювання програми, так і її виконання.

### 1.3.3. Поняття про використання процесу успадкування

*Успадкування* – процес, завдяки якому один об'єкт може набувати властивості іншого. Завдяки успадкуванню підтримується концепція ієрархічної класифікації. У вигляді керованої ієрархічної (низхідної) класифікації організовуються більшість областей знань. Наприклад, яблука *Джонатан* (*йони*) є частиною класифікації *яблука*, яка сама є частиною класу *фрукти*, а той – частиною ще більшого класу *їжа*. Таким чином, клас *їжа* володіє певними якостями (їстівність, поживність і ін.), які застосовуються і до підкласу *фрукти*. Окрім цих якостей, клас *фрукти* має специфічні характеристики (соковитість, солодкість і ін.), які відрізняють їх від інших харчових продуктів. У класі *яблука* визначаються якості, специфічні для них (ростуть на деревах, не тропічні й ін.). Клас *Джонатан* успадковує якості всіх попередніх класів і при цьому визначає якості, які є унікальними для цього сорту яблук.





Якщо не використовувати ієрархічне представлення ознак, для кожного об'єкта довелося б в явній формі визначити всі властиві йому характеристики. Але завдяки процесу успадкування об'єкту потрібно додатково визначити тільки ті якості, які роблять його унікальним усередині його класу, оскільки він (об'єкт) успадковує загальні атрибути свого батька. Отже, саме механізм успадкування дає змогу одному об'єкту представляти конкретний примірник більш загального класу.

## 1.4. Зв'язок мови програмування C++ з мовами Java і C#

Ймовірно, багато студентів знають про існування таких мов програмування як Java[1] і C#[2]. Мову Java розроблено в компанії Sun Microsystems[3], а C# – у корпорації Microsoft[4]. Оскільки іноді виникає плутанина відносно того, яке відношення ці дві мови мають до мови програмування C++, спробуємо внести деяку ясність в неї та розставити крапки над "і".

Мова програмування C++ є батьківською мовою для Java і C#. І хоча розробники цих мов багато додали до першоджерела, видалили з нього або модифікували різні засоби, в цілому синтаксис цих трьох мов практично однаковий. Понад це, об'єктна модель, яку використано у мові C++, подібна до об'єктних моделей мов Java і C#. Нарешті, дуже схоже загальне враження та відчуття від використання всіх цих мов. Це означає, що, знаючи мову C++, Ви можете легко вивчити мову Java або C#. Схожість синтаксисів і об'єктних моделей – одна з причин швидкого засвоєння (і схвалення) цих двох мов багатьма досвідченими C++-програмістами. Зворотна ситуація також можлива: якщо Ви знаєте мову Java або C#, то вивчення мови програмування C++ не завдасть Вам труднощів.

Основна відмінність між мовами C++, Java і C# полягає в типі обчислювального середовища, для якого розроблялася кожна з цих мов. Мова програмування C++ створювалася з метою написання ефективних кодів програм, призначених для виконання під управлянням певної операційної системи і з розрахунку на цифрові перетворювачі конкретного типу. Наприклад, якщо виникає бажання написати високоефективну програму для виконання на процесорі Intel Pentium під управлянням операційної системи Windows, найкраще використовувати для цього мову C++.

---

[1] Java – об'єктно-орієнтована мова програмування, розроблена компанією Sun Microsystems. Додатки, написані нею, зазвичай компілюються в спеціальний байт-код, тому вони можуть працювати на будь-якій віртуальній Java-машині (JVM) незалежно від комп'ютерної архітектури. Дата офіційного випуску – 23 травня 1995 р.

[2] C# (читається сі-шарп, інколи перекладають сі-дієз) – об'єктно-орієнтована мова програмування. Розроблена в 1998-2001 р. групою інженерів під керівництвом Андерса Хейлсберга в компанії Microsoft як основна мова розроблення додатків для платформи Microsoft.NET Framework і згодом була стандартизована як ECMA-334 і ISO/IEC 23270. Компілятор мови C# входить в стандартну установку .NET Framework.

[3] Sun Microsystems, Inc. (зараз входить до складу Oracle Corporation) – американська компанія, виробник програмного і апаратного забезпечення; штаб-квартира компанії розташована в Санта-Кларі (англ. Santa Clara), Каліфорнія, в Кремнієвій долині.

[4] Корпорація Майкрософт (Microsoft Corporation) – багатонаціональна корпорація комп'ютерних технологій (виручка в 2010 фінансовому році – $62,5 млрд, операційний прибуток – $24,1 млрд, чистий прибуток – $18,8 млрд.) з 89 тис. працівниками у 102 країнах (2010 р.), є найбільшою в світі компанією – виробником програмного забезпечення. Головний офіс розташований на корпоративному кампусі в м. Редмонді, штат Вашингтон.

Мови Java і C# були розроблені у відповідь на унікальні потреби сильно розподіленого мережевого середовища, яке може слугувати типовим прикладом сучасних обчислювальних середовищ. Мова Java дає змогу створювати міжплатформений (сумісний з декількома операційними середовищами) переносний програмний код для мережі Internet. Використовуючи мову Java, можна написати програму, яка виконуватиметься в різних обчислювальних середовищах, тобто в широкому діапазоні операційних систем і типів цифрових перетворювачів. Таким чином, Java-програма може вільно почуватися у мережі Internet. Мова C# розроблена для середовища .NET Framework (Microsoft .NET), яка підтримує *багатомовне програмування* (mixed-language programming) і компонентно-орієнтований код, який виконується в мережевому середовищі.

Microsoft.NET (читається дот-нет) – програмна технологія, запропонована фірмою Microsoft як платформа для створення як звичайних програм, так і web-програм. Вона багато в чому є продовженням ідей та принципів, закладених у технологію програмування мовою Java. Одною з ідей .NET технології є сумісність служб, написаних різними мовами. Хоча ця можливість рекламується Microsoft як перевага .NET, платформа Java має таку саму можливість.

Кожна бібліотека (збірка) в середовищі .NET засвідчує свою програмну версію, що дає змогу усунути можливі конфлікти між різними їх версіями:

- .NET Framework 1.0 – випущений у 2002 р.;
- .NET Framework 1.1 – випущений у 2003 р.;
- .NET Framework 2.0 – випущений 27 жовтня 2005 р.;
- .NET Framework 3.0 (кодове ім'я WinFX) – випущений 6 листопада 2006 р. Містить в собі CLR і компілятори від .NET Framework 2.0, плюс низка нових API: Windows Presentation Foundation (WPF, кодове ім'я Avalon), Windows Communication Foundation (WCF, кодове ім'я Indigo), Windows Workflow Foundation (WF) і Windows CardSpace (WCS, кодове ім'я InfoCard). Входить до складу операційної системи Windows Vista;
- .NET Framework 3.5 – випущений 11 січня 2008 р. – є розширенням .NET Framework 3.0, додатково реалізуючи інтеграцію з LINQ, підтримку ASP.NET AJAX, підтримку нових протоколів для Web, таких як AJAX, JSON, REST, POX, RSS, ATOM тощо;
- .NET Framework 4.0 – випущений 12 травня 2010 р.

.NET – крос-платформена технологія, на даний момент існує реалізація для платформи Microsoft Windows, FreeBSD (від Microsoft) і варіант технології для ОС Linux у проекті Mono (в рамках угоди між Microsoft з Novell), DotGNU . Захист авторських прав відноситься до створення середовищ виконання (CLR – Common Language Runtime) для програм .NET. Компілятори для .NET вільно випускаються багатьма фірмами для різних мов.

Програмна технологія .NET поділяється на дві основні частини – середовище виконання (по суті віртуальна машина) та інструментарій розроблення. Середовища розроблення .NET-програм: Visual Studio.NET (C++, C#, J#), SharpDevelop, Borland Developer Studio (Delphi, C#) і т.д. Середовище Eclipse має додаток для розроблення .NET-програм. Застосовувані при цьому програми також можна розробляти в текстовому редакторі та використовувати консольний компілятор.





Як і технологія Java, так і середовище розроблення .NET створює байт-код, призначений для виконання віртуальною машиною. Вхідна мова цієї машини в .NET називається CIL (Common Intermediate Language), також відома як MSIL (Microsoft Intermediate Language), або просто IL. Застосування байт-кода дає змогу отримати крос-платформеність на рівні скомпільованого проекту (в термінах .NET: збірка), а не на рівні початкового тексту, як, наприклад, в мові C++. Перед запуском збірки в середовищі виконання (CLR) байт-код перетворюється вбудованим в середовище JIT-компілятором (just in time, компіляція на льоту) в машинні коди цільового процесора.

Необхідно зазначити, що один з перших JIT-компіляторів для технології Java був також розроблений фірмою Microsoft (на даний момент у середовищі Java використовується більш досконала багаторівнева компіляція – Sun HotSpot). Сучасна технологія динамічної компіляції дає змогу досягнути аналогічного рівня швидкодії з традиційними "статичними" компіляторами (наприклад, C++) і питання швидкодії часто залежить від якості того чи іншого компілятора.

Окрім повної версії .NET, компанією Microsoft також випускається так званий ".NET Compact Framework" .NET Compact Framework є обрізаною версією повного фреймворка і несумісний з ним на рівні виконання (програми, написані для Compact Framework, не можуть виконуватись виконавчим середовищем від повної версії фреймворка, для їх виконання необхідно встановити виконавче середовище саме від Compact Framework). Внутрішньо Compact Framework працює дещо інакше, ніж повний фреймворк, наприклад "збирач сміття" працює значно агресивніше, не розділяючи об'єкти на покоління. Відмінності здебільшого обумовлені особливостями роботи компактних пристроїв: меншими розрахунковими можливостями, значно вищими вимогами до низьких енергозатрат, обмеженими графічними можливостями.

З наведено вище можна зробити висновок про те, що мови Java і C# дають змогу створювати переносний програмний код, який працює в спільно розподіленому середовищі, а ціна цієї переносності – ефективність роботи. Java-програми виконуються значно повільніше, ніж C++-програми. Те саме стосується і мови C#. Тому, якщо виникає бажання створювати високоефективні програмні продукти, використовують мову програмування C++. Якщо ж потрібно переносні програми, то використовують мову Java або C#.

***Вартоазнати!*** *Мови програмування C++, Java і C# призначені для вирішення різноманітних питань та розв'язання різних класів задач. Тому запитання "яка мова є кращою?" поставлене некоректно. Доречніше було б сформулювати запитання дещо по-іншому: "яка мова найбільш підходить для розв'язання конкретної задачі?".*

# Розділ 2. ОСНОВНІ ЕЛЕМЕНТИ МОВИ ПРОГРАМУВАННЯ С++

Найважливішим у вивченні мови програмування С++, безумовно, є те, що жоден її елемент не існує ізольовано від інших, тобто компоненти мови працюють разом. Такий тісний взаємозв'язок ускладнює розгляд одного аспекту мови програмування С++ без перегляду інших. Часто обговорення одного засобу програмування передбачає попереднє знайомство з іншим. Для подолання подібних труднощів у цьому розділі наведено короткий опис таких основних елементів мови програмування С++, як загальний формат С++-програми, основні настанови керування та оператори. При цьому ми не заглиблюватимемося в деталі, а зосередимося на загальних концепціях створення С++-програми. Більшість тільки дещо розглянутих тут елементів мови програмування С++ детальніше розглядаються в наступних розділах цього навчального посібника.

Оскільки вивчати мову програмування найкраще шляхом безпосереднього програмування, то рекомендуємо читачу власноручно виконувати наведені у посібнику приклади на своєму комп'ютері, аналізувати їх і вносити можливі корективи.

## 2.1. Розроблення найпростішої С++-програми

Перш ніж заглибитись детально в теорію програмування, розглянемо спочатку роботу найпростішої С++-програми. Почнемо з введення коду програми, а потім перейдемо до її компілювання та виконання.

**Код програми 2.1. Демонстрація механізму роботи першої програми**

```
// Програма №1 – перша С++-програма.
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()                   // Початок виконання програми.
{
      cout << "Це моя перша С++-програма.";

      getch(); return 0;
}        // Завершення роботи програми.
```

Отже, програміст повинен виконати такі дії: ввести код програми; скомпілювати її; запустити на виконання.

Перш ніж приступати до виконання цих дій, необхідно визначити два терміни: початковий код і об'єктний код. *Початковий код* – версія програми, яку може читати людина. Наведений вище код програми – приклад початкового коду програми. Виконувана версія програми називається *об'єктним*, або *виконуваним кодом*.





### 2.1.1. Особливості введення коду програми

Коди програм, які представлено у цьому навчальному посібнику, у разі потреби необхідно ввести вручну. У цьому випадку бажано використовувати будь-який текстовий редактор (наприклад, WordPad), а не текстовий процесор (Word processor). Йдеться про те, що під час введення коду програми мають бути створені виключно текстові файли, а не файли, у яких разом з текстом зберігається інформація про його форматування. Необхідно також пам'ятати, що зайва інформація про форматування тексту значно перешкоджає роботі С++-компілятора, часто видає помилку компілювання.

Ім'я файлу, який міститиме початковий код програми, формально може бути будь-яким. Але С++-програми зазвичай зберігаються у файлах з розширенням *.cpp. Тому називайте свої С++-програми будь-якими іменами, але як розширення використовуйте *.cpp. Наприклад, назвіть свою першу програму Savchuk_Prog.cpp (це ім'я вживатиметься в подальших настановах), а для інших програм (якщо не буде спеціальних вказівок) вибирайте імена на свій власний розсуд.

### 2.1.2. Компілювання програми

Спосіб компілювання програми Savchuk_Prog.cpp залежить від використовуваного компілятора і вибраних опцій. Понад це, багато компіляторів, наприклад Visual C++ (Microsoft) і C++ Builder (Borland), надають два різні способи компілювання програм: за допомогою компілятора командного рядка та інтегрованого середовища розробки (Integrated Development Environment – IDE). Тому для компілювання С++-програм неможливо дати універсальні настанови, які підійдуть для всіх компіляторів. Це означає, що програміст повинен дотримуватися настанов, наведених у супровідній документації, що додається до Вашого компілятора.

Але, як було зазначено вище, найпопулярнішими компіляторами є Visual C++ і C++ Builder, тому для зручності роботи читачів, які їх використовують, ми наведемо тут настанови з компілювання програм, які відповідають цим компіляторам. Найпростіше в обох випадках компілювати і виконувати програми, наведені у цьому посібнику, з використанням компіляторів командного рядка. Так ми і вчинимо.

Щоб скомпілювати програму Savchuk_Prog.cpp, використовуючи Visual C++, введіть такий командний рядок:

```
C:\...>cl -GX Savchuk_Prog.cpp
```

Опція -GX призначена для підвищення якості компілювання. Щоб використовувати компілятор командного рядка Visual C++, необхідно виконати пакетний файл VCVARS32.bat, який входить до складу Visual C++.

Щоб скомпілювати програму Savchuk_Prog.cpp, використовуючи C++ Builder, введіть такий командний рядок:

```
C:\...>bcc32 Savchuk_Prog.cpp
```

Внаслідок роботи С++-компілятора утворюється об'єктний код, який може виконати комп'ютер після завершення роботи компілятора. Для Windows-середовища виконуваний файл матиме те саме ім'я, що і початковий, але з іншим розширенням, а саме розширенням *.exe. Отже, виконувана версія програми Savchuk_Prog.cpp зберігатиметься в окремому файлі під назвою Savchuk_Prog.exe.

***Вартоазнати!*** *Якщо під час спроби компілювання першої програми було отримано повідомлення про помилку, але Ви упевнені у тому, що ввели її текст коректно і правильно без помилок, то, можливо, Ви використовуєте стару версію С++-компілятора, який був створений ще до прийняття С++-стандарту ANSI/ISO. У цьому випадку зверніться до відповідної літератури за настановами з використання старих компіляторів.*

### 2.1.3. Виконання програми

Скомпільована програма готова до виконання. Оскільки результатом роботи С++-компілятора є виконуваний об'єктний код, то для запуску програми як команду достатньо ввести її ім'я. Наприклад, щоб виконати програму Savchuk_Prog.exe, потрібно ввести такий запис у командному рядку:

```
C:\...>Savchuk_Prog.cpp
```

Результатом виконання цієї програми є такий:

```
Це моя перш С++-програма.
```

Якщо Ви використовуєте інтегроване середовище розробки, то виконати програму можна шляхом вибору з меню команди Run (Виконати). Безумовно, точніші настанови наведено в супровідній документації, що додається до Вашого компілятора. Але, як згадувалося вище, найпростіше компілювати і виконувати наведені у цьому посібнику програми за допомогою програмного середовища С++ Builder.

Необхідно відзначити, що всі ці програми є консольними додатками, а не додатками, які базуються на застосуванні вікон, тобто вони виконуються в сеансі запрошення на введення команди. При цьому Вам, мабуть, відомо, що мова програмування С++ не просто підходить для Windows-програмування, вона – основна мова, що використовують для розроблення Windows-додатків. Проте жодна з програм, представлених у цьому посібнику, не використовує графічного інтерфейсу користувача (GUI – graphics user interface). Йдеться про те, що Windows – достатньо складне середовище для написання програм, що містить багато другорядних тем, не пов'язаних безпосередньо з мовою програмування С++. Водночас консольні додатки є набагато коротшими від графічних і краще надається для вивчення процесу програмування. Засвоївши мову програмування С++, Ви зможете без перешкод застосувати свої знання у сфері створення Windows-додатків.

### 2.1.4. Аналіз рядків коду програми

Після успішного компілювання та виконання першого прикладу програми настав час з'ясувати, як вона працює і чи можна її якось ще вдосконалити. Тому ми детально проаналізуємо кожен її рядок. Отже, наша програма починається з таких рядків.

```
// Програма №1 – перша С++-програма.
```

Це – *коментар*. Подібно до більшості інших мов програмування, мова С++ дає змогу вводити в початковий код програми коментарі, зміст яких компілятор ігнорує. За допомогою коментарів описуються або пояснюються дії, які виконуються у програмі, і ці пояснення призначаються для тих, хто читатиме початковий код. У цьому випадку коментар просто ідентифікує програму і нагадує, що з нею потрібно зробити. Зазвичай, в реальних програмних продуктах коментарі використовуються для пояснення особливостей роботи окремих частин програми або конкретних дій програмних засобів. Іншими словами, Ви можете використовувати коментарі для детального опису всіх (або деяких) її рядків.

***Коментар*** *– текст повідомлення, зауваження чи пояснення, що записується у код програми для кращого розуміння її роботи чи сприйняття.*

У мові програмування С++ підтримується два типи коментарів. Перший, показаний на початку даної програми, називається *багаторядковим*. Коментар цього типу повинен починатися символами /* і закінчуватися ними ж, але переставленими в зворотному порядку (*/). Все, що знаходиться між цими парами символів, компілятор ігнорує. Коментар цього типу, як випливає з його назви, може займати декілька рядків. Другий тип коментарів ми розглянемо трохи нижче.

Наведемо тут наступний рядок програми:

```
#include <iostream>          // Потокове введення-виведення
```

У мові мови програмування С++ визначено ряд *заголовків* (header), які зазвичай містять інформацію, необхідну для програми. У нашу програму внесено заголовок <**iostream**> (він використовують для підтримки С++-системи введення-виведення), який є зовнішнім початковим файлом, що поміщається компілятором в початок програми за допомогою директиви **#include**. Нижче у цьому посібнику ми ближче познайомимося із заголовками і дізнаємося, чому вони такі важливі.

Для розуміння сказаного розглянемо такий рядок програми:

```
using namespace std;          // Використання стандартного простору імен
```

Цей рядок означає, що компілятор повинен використовувати простір імен **std**. Простори імен – відносно недавнє доповнення до мови програмування С++. Детальніше розглянемо їх пізніше, а поки що обмежимося їх коротким визначенням. *Простір імен* (namespace) створює декларативну область, у якій можуть розміщуватися різні елементи програми. Простір імен дає змогу зберігати одну множину імен окремо від іншої. Іншими словами, імена, оголошені в одному просторі, не конфліктуватимуть з іменами, оголошеними в іншому. Простори імен дають змогу спростити організацію великих програм. Ключове слово **return** інформує компілятор про використання заявленого простору імен (у цьому випадку **std**). Саме у просторі імен **std**

оголошена вся бібліотека стандарту мови програмування C++. Таким чином, використовуючи простір імен **std**, Ви спрощуєте доступ до стандартної бібліотеки мови.

Черговий рядок в нашій програмі є *однорядковим коментарем:*

```
// main() – початок виконання програми.
```

Так виглядає коментар другого типу, підтримуваний мовою програмування C++. Однорядковий коментар починається з пари символів // і закінчується в кінці рядка. Як правило, програмісти використовують багаторядкові коментарі для детальних і більше розлогих пояснень, а однорядкові – для коротких (порядкових) описів настанов або призначення використовуваних змінних. Взагалі-то, характер використання коментарів – особиста справа програміста.

Перейдемо до такого рядка:

```
int main()
```

Як повідомлено у щойно розглянутому коментарі, саме з цього рядка і починається виконання програми.

---

*З функції **main**() починається виконання будь-якої C++-програми.*

Всі C++-програми складаються з однієї або декількох функцій[1]. Кожна C++-функція має ім'я, і тільки одна з них (її повинна містити кожна C++-програма) називається **main**(). Виконання C++-програми починається та закінчується (здебільшого) виконанням функції **main**()[2]. Відкрита фігурна дужка в наступному (після **int main**()) рядку вказує на початок коду функції **main**(). Ключове слово **int** (скорочення від слова **int**eger), що знаходиться перед іменем **main**(), означає тип даних для значення, що повертається функцією **main**(). Як Ви згодом дізнаєтеся, мова програмування C++ підтримує декілька вбудованих типів даних, і тип даних **int** – один з них.

Розглянемо черговий рядок програми:

```
cout << "Це моя перш C++-програма.";
```

Це настанова вказує на виведення даних на консоль. Під час її виконання на екрані монітора з'явиться повідомлення: Це моя перша C++-програма. У цій настанові використовують оператор виведення даних "<<". Він забезпечує виведення виразу, що знаходиться з правого боку, на пристрій, який було вказано зліва. Слово **cout** є вбудованим ідентифікатором (складений з частин слів ***c**onsole **out**put*), який здебільшого означає екран комп'ютера. Отже, ця настанова забезпечує виведення заданого повідомлення на екран. Зверніть увагу на те, що ця настанова завершується крапкою з комою. Насправді всі виконувані C++-настанови завершуються крапкою з комою.

Повідомлення "Це моя перша C++-програма." є рядком. У мові програмування C++ під рядком розуміють послідовність символів, поміщених у подвійні лапки. Як Ви побачите пізніше, рядок у мові C++ – один з часто використовуваних елементів мови.

[1] Під функцією ми розуміємо підпрограму.

[2] Точніше, C++-програма починається з виклику функції **main**() і зазвичай закінчується поверненням з неї.





А цим рядком завершується функція **main**():

```
getch(); return 0;
```

Під час її виконання функція **main**() повертає викликуваному процесу (ним зазвичай виступає операційна система) значення 0. Для більшості операційних систем нульове значення, яке повертає ця функція, свідчить про нормальне завершення роботи програми. Інші значення можуть означати завершення роботи програми у зв'язку з якою-небудь помилкою. Слово **return** належить до ключових і використовують для повернення значення з функції. При нормальному завершенні (тобто без помилок) всі Ваші програми повинні повертати значення 0.

Закрита фігурна дужка в кінці програми формально завершує її. Хоча фігурна дужка насправді не є частиною об'єктного коду програми, її "виконання" (тобто обробку закритої фігурної дужки функції **main**()) подумки можна вважати кінцем C++-програми. І справді, якщо у наведеному прикладі коду програми настанова **return** була б відсутня, то програма автоматично завершилася б після досягнення цієї закритої фігурної дужки.

### 2.1.5. Оброблення синтаксичних помилок

Кожному програмісту відомо, наскільки легко під час введення коду програми в комп'ютер вносяться випадкові помилки (друкарські огріхи). На щастя, під час компілювання такої програми компілятор "сигналить" повідомленням про наявність *синтаксичних помилок*. Більшість C++-компіляторів спробують "побачити" сенс в початковому коді програми, незалежно від того, що Ви ввели. Тому повідомлення про помилку не завжди відображає дійсну причину проблеми. Наприклад, якщо в попередній програмі випадково опустити відкриту фігурну дужку після імені функції **main**(), компілятор вкаже як джерело помилки настанову **cout**. Тому під час отримання повідомлення про помилку прогляньте два-три рядки коду програми, які безпосередньо передують рядку з "виявленою" помилкою. Адже іноді трапляється так, що компілятор починає виявляти збої програми тільки через декілька рядків після реальної помилки.

Багато C++-компіляторів видають як результати своєї роботи не тільки повідомлення про помилки, але і попередження (warning). У мову програмування C++ "від народження" закладено доброзичливе ставлення до програміста, тобто вона дає змогу програмісту чинити практично все, що коректно з погляду синтаксису. Проте навіть "всепрощаючим" C++-компіляторам деякі синтаксично правильні речі можуть видатися підозрілими. У таких ситуаціях і видається попередження. Тоді програміст сам повинен оцінити, наскільки справедливі підозри компілятора. Деякі компілятори дуже пильні, тобто інколи виводять попередження з приводу набору абсолютно коректних настанов. Окрім цього, компілятори дають змогу використовувати різні опції, які можуть інформувати про деякі речі, що можуть цікавити Вас. Іноді така інформація має форму застережливого повідомлення навіть незважаючи на відсутність "складу" попередження. Програми, наведені у посібнику, написа-

ні відповідно до стандарту мови програмування C++ і при коректному введенні не повинні генерувати ніяких застережливих повідомлень.

***Вартоазнати!а****Більшість C++-компіляторів пропонують декілька рівнів повідомлень (і попереджень) про помилки. У загальному випадку можна вибрати тип помилок, про наявність яких Ви хотіли б отримувати повідомлення. Наприклад, більшість компіляторів за бажанням програміста можуть інформувати про використання неефективних конструкцій або застарілих засобів. Для прикладів цього навчального посібника достатньо використовувати звичайне налаштування компілятора. Але програмісту все ж таки доцільно заглянути в документацію, що додається до компілятора, і поцікавитися, які можливості керування процесом компілювання є у його розпорядженні. Багато компіляторів достатньо "інтелектуальні" і можуть допомогти у виявленні неочевидних помилок ще до того, як вони спричинять великі проблеми. Знання принципів, що використовуються компілятором під час складання звіту про помилки, вимагає часу і зусиль, які потрібно програмісту на їх засвоєння.*

## 2.2. Розроблення навчальної програми

Будь-яка навчальна програма має передбачати виконання таких дій:

- присвоєнням значень змінним у самій програмі;
- введення даних у програму за допомогою клавіатури;
- виконання розрахунків;
- деякі можливості виведення даних;
- введення нового типу даних.

### 2.2.1. Присвоєнням значень змінним

Можливо, найважливішою конструкцією в будь-якій мові програмування є присвоєння змінній деякого значення. *Змінна* – іменована область пам'яті, у якій можуть зберігатися різні значення. При цьому значення змінної у процесі виконання програми можна змінити один або кілька разів. Іншими словами, вміст змінної є змінним, а не фіксованим.

У наведеному нижче коді програми створюється цілочисельна змінна з іменем x, якій присвоюється значення 1023, а потім на екрані монітора з'являється таке повідомлення:

```
Ця програма виводить значення змінної x: 1023.
```

**Код програми 2.2. Демонстрація механізму розроблення другої програми**

```
// Програма №2 – використання змінної.
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int x;                   // Тут визначається тип змінної
    x = 1023;                // Тут змінній x присвоюється число 1023.
    cout << "Ця програма виводить значення змінної x: ";
```





```
    cout << x;           // Відображення числа 1023.
    getch(); return 0;
}
```

Що ж нового у цьому коді програми? По-перше, настанова:

```
int x;                   // Тут визначається тип змінної.
```

оголошує змінну з іменем x цілого типу. У мові програмування C++ всі змінні мають бути оголошені до їх використання. У визначенні змінної, окрім її імені, треба вказати, значення якого типу вона може зберігати. Тим самим оголошується *тип* змінної. У цьому випадку змінна x може зберігати цілочисельні значення, тобто цілі числа, що знаходяться в діапазоні -32 768 ÷ +32 767. У мові програмування C++ для оголошення змінної цілого типу достатньо поставити перед її іменем ключове слово **int**. Таким чином, настанова **int x;** оголошує змінну x типу **int**. Нижче дізнаємося, що мова програмування C++ підтримує широкий діапазон вбудованих типів змінних[1].

По-друге, у процесі виконання такої настанови змінній присвоюється конкретне значення:

```
x = 1023;            // Тут змінній x присвоюється число 1023.
```

У мові програмування C++ *оператор присвоєння* позначають одиночним знаком рівності (=). Його дія полягає в копіюванні значення, розташованого праворуч від оператора, у змінну, вказану зліва від нього. Після виконання цієї настанови присвоєння змінна x міститиме число 1023.

Результати, що були видані цією програмою, відображаються на екрані за допомогою двох настанов **cout**. Зверніть увагу на використання такої настанови для виведення значення змінної x:

```
cout << x;           // Відображення числа 1023.
```

У загальному випадку для відображення значення змінної достатньо в настанові **cout** помістити її ім'я праворуч від оператора "<<":

```
cout << "Ця програма виводить значення змінної x: " << x;
```

Оскільки у цьому конкретному випадку змінна x містить число 1023, то його і буде відображено на екрані.

Перед останнім оператором **return** в секції **main**() знаходиться функція **getch**(), за допомогою якої Ви зможете побачити результати роботи програми на екрані.

Перш ніж переходити до наступного розділу, спробуйте присвоїти змінній x інші значення (у початковому коді програми) і подивіться на результати виконання цієї програми після внесення відповідних змін.

### 2.2.2. Введення з клавіатури даних у програму

Перші дві програми, окрім демонстрації декількох важливих засобів мови програмування C++, не робили нічого корисного. У наведеному нижче коді програми розв'язується практична задача перетворення галонів у літри. Тут також показано один із способів введення даних у програму.

[1] Понад це, мова програмування C++ дає змогу програмісту визначати власні типи даних.

**Код програми 2.3. Демонстрація механізму роботи програми, яка перетворює галони у літри**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int gallons, liters;

    cout << "Введіть кількість галонів: ";
    cin >> gallons;          // Введення даних від користувача.
    liters = gallons * 4;    // Перетворення в літри.
    cout << "Літрів: " << liters;

    getch(); return 0;
}
```

Ця програма спочатку відображає на екрані повідомлення, що пропонує користувачу ввести число для перетворення галонів у літри, а потім чекає доти, доки його не буде введено[1]. Потім програма відобразить значення, що приблизно дорівнює еквівалентному об'єму, вираженому в літрах. Насправді для отримання точного результату необхідно використовувати коефіцієнт 3,7854 (тобто в одному галоні поміщається 3,7854 літра), але оскільки у наведеному прикладі ми працюємо з цілочисельними змінними, то коефіцієнт перетворення заокруглений до 4.

Зверніть увагу на те, що дві змінні gallons і liters оголошуються після ключового слова **int у** формі списку, елементи якого розділяються між собою комами. У загальному випадку можна оголосити будь-яку кількість змінних одного типу, розділивши їх комами[2].

Для прийняття значення, яке вводить користувач використовують таку настанову:

```
cin >> gallons;          // Введення даних від користувача.
```

У цьому записі застосовується ще один вбудований ідентифікатор – **cin**, що надається C++-компілятором. Він складений з частин слів ***c**onsole **i**nput* і, здебільшого, означає введення даних з клавіатури. Як оператор введення даних використовується символ ">>". У процесі виконання цієї настанови значення, введене користувачем (яке у цьому випадку повинно бути цілочисельним), поміщається в змінну, вказану з правого боку від оператора ">>" (у цьому випадку це змінна gallons).

У цьому коді програми заслуговує уваги і ця настанова:

```
cout << "Літрів: " << liters;
```

У цьому записі цікавим є те, що в одній настанові використано відразу два оператори виведення даних "<<". Під час виконання цієї настанови спочатку буде виведено рядок "Літрів: ", а за ним – значення змінної liters. У загальному випадку в одній настанові можна поєднувати будь-яку кількість операторів виведення даних, кожен елемент виведення якої передує "своїм" оператором "<<".

[1] Необхідно пам'ятати, що Ви повинні ввести ціле число галонів, тобто число, що не містить дробової частини.

[2] Як альтернативний варіант можна використовувати декілька декларативних int-настанов – результат буде той самий.





### 2.2.3. Деякі можливості виведення даних

Дотепер у нас не було потреби під час виведення даних забезпечувати перехід на наступний рядок. Проте така необхідність може виявитися дуже скоро. У мові програмування C++ послідовність символів "повернення каретки/переклад рядка" генерується за допомогою *символу нового рядка*. Для виведення цього символу використовують код \n (символ зворотної косої риски "\" і малої літери "n"). Продемонструємо використання послідовності символів "повернення каретки/переклад рядка" на прикладі такої програми.

**Код програми 2.4. Демонстрація механізму роботи \n-послідовностей, які забезпечують перехід на новий рядок**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    cout << "один\n";
    cout << "два\n";
    cout << "три\n";
    cout << "чотири\n";

    getch(); return 0;
}
```

У процесі виконання програма відображає на екрані такі результати:

```
один
два
три
чотири
```

Символ переходу на новий рядок (\n) можна помістити в будь-якому місці рядка, а не тільки в його кінці. "Погайте" з символом нового рядка, щоб переконатися у тому, що Ви правильно розумієте його призначення.

### 2.2.4. Введення нового типу даних

Попри те, що для виконання приблизних підрахунків розглянута вище програма перетворення галонів у літри цілком придатна, проте для отримання дещо точніших результатів її необхідно переробити. Як зазначено вище, за допомогою цілочисельних типів даних неможливо представити значення з дробовою частиною. Для них потрібно використовувати один з типів даних з плинною крапкою, наприклад **double** (подвійної точності). Дані цього типу зазвичай знаходяться в діапазоні 1,7E-308÷1,7E+308. Операції, що виконуються над числами з плинною крапкою, зберігають будь-яку дробову частину результату і, отже, забезпечують точніше перетворення.

У наведеному нижче коді програми перетворення галонів у літри використовуються для значення з плинною крапкою.

**Код програми 2.5. Демонстрація механізму перетворення галонів у літри за допомогою чисел з плинною крапкою**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    double gallons, liters;

    cout << "Введіть кількість галонів: ";
    cin >> gallons;                  // Введення даних від користувача.
    liters = gallons * 3.7854;       // Перетворення в літри.
    cout << "Літрів: " << liters;

    getch(); return 0;
}
```

Для отримання цього варіанта програми в її попередню версію було внесено дві зміни. По-перше, змінні gallons і liters оголошені цього разу з використанням типу **double**. По-друге, коефіцієнт перетворення заданий у вигляді числа 3.7854, що дає змогу отримати точніші результати. Якщо C++-компілятор потрапляє на число, що містить десяткову крапку, він автоматично сприймає його як константу з плинною крапкою. Зверніть також увагу на те, що настанови **cout** і **cin** залишилися здебільшого такими, як у попередньому варіанті програми, у якій використовувалися змінні типу **int**. Це дуже важливий момент: C++-система введення-виведення автономно налаштовується на тип даних, який було вказано у програмі.

Скомпілюйте і виконайте цю програму. На пропозицію вказати кількість галонів, введіть число 1. Як результат виконання програма має відобразити на екрані 3,7854 літра. Спробуйте після цього ввести інші числові значення, внаслідок чого отримаєте й відповідні результати розрахунку.

Отже, підведемо підсумок найважливіше з того, що уже пройдено у попередньому матеріалі, а саме:

1. Кожна C++-програма має основну функцію **main**(), яка означає початок виконання програми.
2. Всі змінні мають бути оголошені до їх використання.
3. Мова програмування C++ підтримує різні типи даних, в т.ч. цілочисельні та з плинною крапкою.
4. Оператор виведення даних позначається символом "<<", а під час використання в настанові **cout** він забезпечує відображення інформації на екрані монітора.
5. Оператор введення даних позначається символом ">>", а під час використання в настанові **cin** він зчитує інформацію з клавіатури.
6. Виконання програми завершується після завершення роботи основної функції **main**().





## 2.3. Функції – "будівельні блоки" C++-програми

***Функція*** – у програмуванні – один з видів підпрограми. Особливість, що відрізняє її від іншого виду підпрограм – процедури, полягає в тому, що функція повертає значення, а її виклик може використатися в програмі як вираз.

***Підпрограма*** – частина програми, яка реалізує певний алгоритм і дає змогу звернення до неї з різних частин загальної (головної) програми. Підпрограма часто використовується для скорочення розмірів програм у тих завданнях, у процесі розв'язання яких необхідно виконати декілька разів однаковий алгоритм при різних значеннях параметрів. Оператори (команди), які реалізують відповідну підпрограму, записують один раз, а в необхідних місцях розміщують оператори передачі управління на цю підпрограму.

З погляду теорії систем, функція в програмуванні – окрема система (підсистема, підпрограма), на вхід якої надходять керуючі впливи у вигляді значень аргументів. На виході системи одержуємо результат виконання програми, що може бути як скалярною величиною, так і векторним значенням. За ходом виконання функції можуть виконуватися також деякі зміни в керованій системі, причому як оборотні, так і необоротні.

У деяких мовах програмування (наприклад, у мові Pascal) функції існують поряд із процедурами (підпрограмами, що не повертають значення), в інші, наприклад, у мові C++, є єдиним реалізованим видом підпрограми (тобто всі підпрограми є функціями й можуть повертати значення).

Побічним ефектом функції називається будь-яка зміна функцією стану програмного середовища, крім повернення результату (зміна значень глобальних змінних, виділення й звільнення пам'яті, введення-виведення і так далі). Теоретично найбільш правильним є використання функцій, що не мають побічного ефекту (тобто таких, внаслідок виклику яких повертається обчислене значення і тільки), хоча на практиці доводиться використати функції з побічним ефектом, хоча б для забезпечення введення-виведення й відображення результатів роботи програми. Існує специфічна парадигма програмування – функціональне програмування, у якій будь-яка програма є набір вкладених викликів функцій, що не викликають побічних ефектів. Найбільш відома мова програмування, що реалізує цю парадигму – Лісп[1]. У ньому будь-яка операція, будь-яка конструкція мови, будь-який вираз, окрім константи, є викликами функцій.

### 2.3.1. Основні поняття про функції

Будь-яка C++-програма складається з "будівельних блоків", що називаються *функціями*. Функція – підпрограма, яка містить одну або декілька C++-настанов і здійснює одну або декілька задач. Хороший стиль програмування

[1] Лісп (LISP) – мова програмування загального призначення з підтримкою парадигм функціонального та процедурного програмування. Вихідна інформація записується у вигляді списків. Мову програмування Лісп було розроблено в кінці 1950-тих у Масачусетському Технологічному Інституті для дослідження проблем штучного інтелекту. Але, через потужність закладених принципів, мова програмування Лісп також придатна для багатьох інших застосувань.

мовою C++ передбачає, що кожна функція виконує тільки одну задачу, наприклад, окремо введення та виведення даних.

Кожна функція має ім'я, яке використовують для її виклику. Своїм функціям програміст може давати будь-які імена за винятком імені **main**(), зарезервованого для функції, з якої починається виконання програми.

У мові програмування C++ жодна функція не може бути вбудована в іншу. На відміну від таких мов програмування, як Pascal, Modula-2 і деяких інших, які дають змогу використовувати вкладені функції, у мові програмування C++ всі функції розглядаються як окремі компоненти[1].

Під час позначення функцій у цьому посібнику використовують домовленість (зазвичай її дотримується в літературі, присвяченій мові програмування C++), згідно з якою ім'я функції завершується парою круглих дужок. Наприклад, якщо функція має ім'я getVal, то її згадка в тексті позначиться як Put(). Дотримання цієї домовленості дасть змогу легко відрізняти імена змінних від імен функцій.

У вже розглянутих прикладах програм функція **main**() була єдиною. Як було зазначено вище, функція **main**() – перша функція, яка виконується під час запуску програми. Її повинна містити кожна C++-програма. Взагалі, функції, які Вам належить використовувати, бувають двох типів. До першого типу належать функції, написані програмістом (**main**() – приклад функції такого типу). Функції іншого типу знаходяться в *стандартній бібліотеці* C++-компілятора[2]. Як правило, C++-програми містять як функції, написані програмістом, так і функції, які надає компілятор.

Оскільки функції утворюють фундамент мови програмування C++, займемося ними ґрунтовніше. Наведена вище програма містить дві функції: **main**() і FunC(). Ще до виконання цієї програми (або читання подальшого опису) уважно вивчіть її текст і спробуйте передбачити, що вона повинна відобразити на екрані.

**Код програми 2.6. Демонстрація механізму реалізації структури програми, що містить дві функції: main() і FunC()**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void FunC();                 // Попереднє оголошення прототипу функції FunC()

int main()
{
      cout << "У функції main().";
      FunC();                // Викликаємо функцію FunC().
      cout << "Знову у функції main()."

      getch(); return 0;
}
```

[1] Безумовно, одна функція може викликати іншу.

[2] Стандартну бібліотеку буде розглянуто нижче, а поки помітимо, що вона є колекцією вбудованих функцій.





```
void FunC()                  // Визначення функції FunC()
{
      cout << " У функції FunC(). ";
}
```

Програма працює таким чином. Спочатку викликається функція **main**() і здійснюється її перша **cout**-настанова. Потім з функції **main**() викликається функція FunC(). Зверніть увагу на те, як цей виклик реалізується у програмі: вказується ім'я функції FunC, за яким стоїть пара круглих дужок і крапка з комою. Виклик будь-якої функції є C++-настановою і тому повинен завершуватися крапкою з комою. Потім функція FunC() здійснює свою єдину **cout**-настанову і передає керування назад функції **main**(), причому тому рядку коду програми, який розташований безпосередньо за викликом функції. Нарешті, функція **main**() здійснює свою другу **cout**-настанову, яка завершує всю програму. Отже, на екрані ми повинні побачити такі результати:

У функції **main**(). У функції FunC(). Знову у функції **main**().

У цьому коді програми необхідно розглянути таку настанову:

```
void FunC();                 // Попереднє оголошення прототипу функції FunC()
```

Як зазначено в коментарі, це – *прототип оголошення функції* FunC(). Хоча детальніше прототипи буде розглянуто нижче, все ж таки без коротких пояснень тут не обійтися. Прототип функції оголошує функцію до її визначення. Прототип дає змогу компіляторові дізнатися тип значення, що повертається цією функцією, а також кількість і тип параметрів, які вона може мати. Компіляторові потрібно знати цю інформацію до першого виклику функції. Тому прототип розташовується ще до функції **main**(). Єдиною функцією, яка не вимагає прототипу, є **main**(), оскільки вона є вбудованою у мові програмування C++.

Як бачимо, функція FunC() не містить настанови **return**. Ключове слово **void**, яке передує як прототипу, так і визначенню функції FunC(), формально заявляє про те, що функція FunC() не повертає ніякого значення. У мові програмування C++ функції, які не повертають значень, оголошуються з використанням ключового слова **void**.

### 2.3.2. Загальний формат визначення C++-функцій

У попередніх прикладах було показано конкретний тип функції. Проте всі C++-функції мають такий загальний формат їх визначення:

```
тип_поверненого_значення ім'я_функції(перелік_параметрів)
{
      .
      .  // тіло методу
      .
}
```

Розглянемо детально всі елементи, з яких складається функція.

За допомогою елемента *тип_поверненого_значення* вказується тип значення, що повертається функцією. Як буде показано далі, це може бути практично будь-який тип, в т.ч. тип, що створюється безпосередньо програмістом.

Якщо функція не повертає ніякого значення, треба вказати тип **void**. Якщо функція дійсно повертає значення, воно повинно мати тип, сумісний з вказаним у визначенні функції.

Кожна функція має ім'я. Воно, як неважко здогадатися, задається елементом *ім'я функції*. Після імені функції поміж круглих дужок вказують перелік параметрів, який є послідовністю пар (складаються з типу даних і імені), розділених між собою комами. Якщо функція не має параметрів, елемент *перелік_параметрів* відсутній, тобто круглі дужки залишаються порожніми.

У фігурні дужки поміщено тіло функції. Тіло функції становлять С++-настанови, які визначають конкретні дії функції. Функція завершується (і керування передається процедурі, яка її викликає), досягши закритої фігурної дужки або настанови **return**.

### 2.3.3. Передавання аргументів функції

Функції можна передати одне або декілька значень. Значення, що передається функції, називають *аргументом*. Хоча у програмах, які ми розглядали дотепер, жодна з функцій (ні **main**(), ні FunC()) не отримувала ніяких значень, функції у мові програмування С++ можуть приймати один або декілька аргументів. Верхня межа кількості аргументів, що приймаються, визначається конкретним компілятором. Згідно зі стандартом мови програмування С++, він дорівнює 256.

***Аргумент*** *– значення, що передається функції під час її виклику.*

Розглянемо коротку програму, яка для відображення абсолютного значення числа використовує стандартну бібліотечну (тобто вбудовану) функцію **abs**(). Ця функція приймає один аргумент, перетворює його в абсолютне значення і повертає результат.

**Код програми 2.7. Демонстрація механізму використання функції abs()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен

int main()
{
      cout << abs(-10);

      getch(); return 0;
}
```

У цьому коді програми функції **abs**() як аргументу передається число -10. Функція **abs**() приймає цей аргумент під час виклику і повертає його абсолютне значення, яке у цьому випадку дорівнює числу 10. Важливо запам'ятати: якщо функція приймає аргумент, то його вказують усередині круглих дужок, розташованих відразу після імені функції.

Значення, що повертається функцією **abs**(), використовується настановою **cout** для відображення на екрані абсолютного значення числа -10. Йдеться про те, якщо функція є частиною виразу, то вона автоматично викликається для отримання значення, що повертається нею. У цьому випадку значення, що повертається функцією **abs**(), виявляється праворуч від оператора "<<" і тому законно відображається на екрані.

Зверніть також увагу на те, що наведений вище код програми містить заголовок <**cstdlib**>. Цей заголовок необхідний для забезпечення можливості виклику функції **abs**(). Кожного разу, коли Ви використовуєте бібліотечну функцію, у програмі необхідно помістити відповідний заголовок. Заголовок, окрім іншої інформації, повинен містити прототип бібліотечної функції.

***Параметр*** *– визначена функцією змінна, яка приймає аргумент, що передається функції.*

Під час розроблення функції, яка приймає один або декілька аргументів, іноді необхідно оголосити змінні, які зберігатимуть значення аргументів. Ці змінні називають *параметрами* функції. Наприклад, наступна функція виводить добуток двох цілочисельних аргументів, що передаються функції під час її виклику:

```
void funZ(int x, int y)
{
      cout << x * y << " ";
}
```

Під час кожного виклику функції funZ() здійснюється множення значення, що передається параметру x, на значення, передане параметру y. Проте пам'ятайте, що x і y – просто змінні, які приймають значення, які передаються під час виклику функції. Розглянемо таку коротку програму, яка демонструє використання функції funZ().

**Код програми 2.8. Демонстрація механізму використання функції funZ()**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

void funZ(int x, int y);     // Попереднє оголошення прототипу функції funZ()

int main()
{
      funZ(10, 20);
      funZ(5, 6);
      funZ(8, 9);

      getch(); return 0;
}

void funZ(int x, int y)      // Визначення функції funZ()
{
      cout << x * y << " ";
}
```

Ця програма виведе на екран числа 200, 30 і 72. Під час виклику функції funZ() С++-компілятор копіює значення кожного аргумента у відповідний параметр. У цьому випадку під час першого виклику функції funZ() число 10 копіюється в змінну x, а число 20 – в змінну y. Під час другого виклику 5 копіюється в x, а 6 – в y. Під час третього виклику 8 копіюється в x, а 9 – в y.

Якщо Ви ніколи не працювали з мовою програмування, у якій дозволені функції, які параметризуються, то описаний вище процес може видатися дещо дивним. Проте хвилюватися не варто: у міру перегляду інших С++-програм Вам стане значно зрозумілішим принцип використання функцій, їх аргументів і параметрів.

***Вартоазнати!*** *Термін* ***аргумент*** *належить до значення, яке використовується під час виклику функції. Змінна, яка приймає цей аргумент, називається* параметром. *Функції, які приймають аргументи, називаються* ***параметризованими функціями***.

Якщо С++-функції мають два або більше аргументів, то вони розділяються між собою комами. У цьому посібнику під терміном *перелік аргументів* необхідно розуміти аргументи, розділені між собою комами. Для розглянутої вище функції funZ() перелік аргументів виражений у вигляді x, y.

### 2.3.4. Повернення функціями аргументів

У мові програмування С++ багато бібліотечних функцій повертають значення. Наприклад, вже знайома нам функція **abs**() повертає абсолютне цілочисельне значення свого аргументу. Функції, які написано програмістом, також можуть повертати значення. У мові програмування С++ для повернення значення використовують настанову **return**. Загальний формат цієї настанови є таким:

**return** *значення*;

Неважко здогадатися, що тут елемент *значення* є значенням, що повертається функцією.

Щоб продемонструвати механізм повернення функціями значень, переробимо попередню програму так, як це показано далі. У цій версії функція funZ() повертає добуток своїх аргументів. Зверніть увагу на те, що розташування функції праворуч від оператора присвоєння означає присвоєння змінній (розташованій зліва) значення, що повертається цією функцією.

**Код програми 2.9. Демонстрація механізму повернення функціями значень**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int funZ(int x, int y);      // Попереднє оголошення прототипу функції funZ()

int main()
{
    int rezult;
```





```
    rezult = funZ(10, 11);   // Присвоєння значення, що повертається функцією.
    cout << "Відповідь дорівнює " << rezult;

    getch(); return 0;
}

// Ця функція повертає значення.
int funZ(int x, int y)           // Визначення функції funZ()
{
    return x * y;                // Функція повертає добуток x і y.
}
```

У наведеному прикладі функція funZ() повертає результат обчислення виразу x * y за допомогою настанови **return**. Потім значення цього результату присвоюється змінній rezult. Таким чином, значення, що повертається настановою **return**, стає значенням функції funZ() у програмі, яка її викликає.

Оскільки у цій версії програми функція funZ() повертає значення, то її ім'я у визначенні не передує слово **void**[1]. Оскільки існують різні типи змінних, існують і різні типи значень, що повертаються функціями. Тут функція funZ() повертає значення цілого типу. Тип значення, що повертається функцією, передує її імені як у прототипі, так і у визначенні.

У попередніх версіях мови програмування С++ для типів значень, що повертаються функціями, існувала домовленість, що діє за замовчуванням. Якщо тип значення, що повертається функцією, не вказано, то передбачалося, що ця функція повертає цілочисельне значення. Наприклад, функція funZ(), згідно з тією домовленістю, могла бути записана так:

```
funZ(int x, int y)          // За замовчуванням тип значення,
                            // що повертається функцією, використовується тип int.
{
    return x * y;           // Функція повертає добуток x і y.
}
```

У цьому записі за замовчуванням передбачається цілочисельний тип значення, що повертається функцією, оскільки не задано ніякого іншого типу. Проте правило встановлення цілого типу за замовчуванням було знехтуване стандартом мови програмування С++. Незважаючи на те, що більшість компіляторів підтримують це правило заради зворотної сумісності, програміст повинен безпосередньо задавати тип значення, що повертається кожною функцією, яку він пише. Але, якщо Вам доведеться мати справу зі старими версіями С++-програм, то цю домовленість необхідно мати на увазі.

Досягши настанови **return,** функція негайно завершується, а увесь решта програмний код ігнорується. Функція може містити декілька настанов **return**. Повернення з функції можна забезпечити за допомогою настанови **return** без вказання значення, що повертається, але таку її форму допустимо застосовувати тільки для функцій, які не повертають ніяких значень і оголошені з використанням ключового слова **void**.

[1] Згадаймо, слово **void** використовується тільки у тому випадку, коли функція не повертає ніякого значення.

### 2.3.5. Спеціальна функція main()

Як зазначалося вище, функція **main**() – спеціальна, оскільки це перша функція, яка викликається у процесі виконання програми. На відміну від деяких інших мов програмування, у яких виконання завжди починається "зверху", тобто з першого рядка коду програми, кожна С++-програма завжди починається з виклику основної функції **main**() незалежно від її розташування у програмі[1].

У програмі може бути тільки одна функція **main**(). Якщо спробувати залучити до програми декілька функцій **main**(), то вона "не знатиме", з якої з них почати роботу. Насправді більшість компіляторів легко виявить помилку цього типу і повідомить про неї. Як було зазначено вище, оскільки функція **main**() вбудована у мову програмування С++, то вона не вимагає прототипу.

## 2.4. Поняття про логічну та циклічну настанови

Для аналізу більш реальних прикладів конкретних програм нам необхідно познайомитися з двома простими С++-настановами: **if** і **for**[2].

### 2.4.1. Логічна настанова if

Настанова **if** у мові програмування С++ діє подібно до настанови **if**, визначеної в будь-якій іншій мові програмування. Її простий формат є таким:

**if**(*умова*) *настанова*;

У цьому записі елемент *умова* – вираз, який під час обчислення може виявитися значенням, що дорівнює ІСТИНІ або ФАЛЬШІ. У мові програмування С++ ІСТИНА представляється ненульовим значенням, а ФАЛЬШ – нулем. Якщо *умова*, або умовний вираз, є істинним, елемент *настанова* виконається, інакше – ні. У процесі виконання такої настанови

```
if(10 < 11) cout << "10 менше ніж 11";
```

на екрані відобразиться фраза: 10 менше ніж 11.

---

*Логічна настанова* ***if*** *дає змогу зробити вибір між двома виконуваними гілками програми.*

---

Такі оператори порівняння, як "<" (менше) і ">=" (більше або дорівнює), використовуються в багатьох інших мовах програмування. Але необхідно пам'ятати, що у мові програмування С++ як оператор рівності застосовується подвійний символ "дорівнює" (==). У наведеному нижче прикладі **cout**-настанова не виконається, оскільки умовний вираз дає значення ФАЛЬШ. Іншими словами, оскільки 10 не дорівнює 11, то **cout**-настанова не відобразить на екрані вітання:

```
if(10 == ll) cout << "Привіт";
```

[1] Все ж таки зазвичай функцію **main**( ) розміщують першою, щоб її було легко знайти.
[2] Детальніше їх опис наведено нижче в цьому навчальному посібнику.





Безумовно, операнди умовного виразу необов'язково мають бути константами. Вони можуть бути змінними і навіть містити звернення до функцій.

У наведеному нижче коді програми показано приклад використання **if**-настанови. У процесі виконання цієї програми користувачу пропонується ввести два числа, а потім повідомляється результат їх порівняння.

**Код програми 2.10. Демонстрація механізму використання if-настанови**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int a, b;

    cout << "Введіть перше число: "; cin >> a;
    cout << "Введіть друге число: "; cin >> b;
    if(a < b) cout << "Перше число менше від другого.";
    if(a == b) cout << "Перше число збігається з другим.";
    if(a > b) cout << "Перше число більше від другого.";

    getch(); return 0;
}
```

### 2.4.2. Циклічна настанова for

Цикл **for** повторює вказану настанову задану кількість разів. Настанова **for** у мові програмування С++ діє практично так само, як настанова **for**, визначена в таких мовах програмування, як Java C#, Pascal і Visual Basic. Її простий формат є таким:

**for**(*ініціалізація*; *умова*; *інкремент*) *настанова*;

У цьому записі елемент *ініціалізація* є настанова присвоєння, яка встановлює *керівній змінній циклу* початкове значення. Ця змінна діє як лічильник, який керує роботою циклу. Елемент *умова* є виразом, у якому тестується значення керівної змінної циклу. Результат цього тестування визначає, виконається цикл **for** ще раз чи ні. Елемент *інкремент* – вираз, який визначає, як змінюється значення керівної змінної циклу після кожної ітерації. Цикл **for** виконуватиметься доти, доки обчислення елемента *умова* дає істинний результат. Як тільки умова стане помилковою, виконання програми продовжиться з настанови, що знаходиться наступною за циклом **for**.

---

*Цикл* ***for*** *– одна з циклічних настанов, визначених мовою С++.*

---

Наприклад, наведений нижче код програми за допомогою циклу **for** виводить на екран числа від 1 до 100.

**Код програми 2.11. Демонстрація механізму використання for-циклу**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
```

```
{
    int pm;

    for(pm=1; pm<=100; pm=pm+1) cout << pm << " ";

    getch(); return 0;
}
```

На рис. 2.1 схематично показано виконання циклу **for** у наведеному прикладі. Як бачимо, спочатку змінна pm ініціалізується числом 1. Під час кожного повторення циклу перевіряється умова pm<=100. Якщо результат перевірки виявляється істинним, **cout**-настанова виводить значення змінної pm, після чого її вміст збільшується на одиницю. Коли значення змінної pm перевищить значення 100, то умова, що перевіряється, видасть значення ФАЛЬШ, і виконання циклу припиниться.

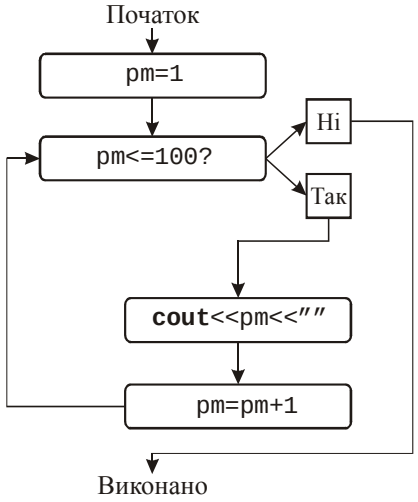



**Рис. 2.1. Виконання циклу for**

У професійно написаному C++-коді програми рідко трапляються настанови pm = pm+1, оскільки для настанов такого роду у мові програмування C++ передбачена спеціальна скорочена форма: pm++ (див. розд. 3.5.3). Оператор "++" називається *оператором інкремента*. Він збільшує операнд на одиницю. Оператор "++" доповнюється оператором "--" (*оператором декремента*), який зменшує операнд на одиницю. За допомогою оператора інкремента використану в попередній програмі настанову **for** можна переписати так:

```
for(pm=1; pm<=100; pm++) cout << pm << " ";
```

## 2.5. Структуризація C++-програми

Початковою "цеглинкою" кожної складної програми є її проект. На основі аналізу системи заходів формується набір проектів, що становлять основу розроблення програми. Вони потім об'єднуються в блоки та підпрограми. У кожній з ланок структури програми має бути доступна для огляду кількість елементів, щоб забезпечити її керованість. Реалізація проектів розподіляються в часі, більшість з яких узгоджуються між собою.

Оцінювання витрат та ефективності проектів, підпрограм, програми загалом – вирішальний етап технології структурного програмування. По кожному проекту оцінюються необхідні витрати на розроблення, інвестиції та ін. Отримані дані підсумовуються по блоках проектів, підпрограмами і програмою загалом. При цьому можуть виявлятися неефективні, збиткові проекти та їхні блоки. Приймаються рішення про підвищення їх ефективності або про вилучення. Інколи можливе їх збереження, якщо збиток перекривається додатковим доходом від реалізації суміжних проектів. У підсумку цього етапу уточнюється структура програми, а в окремих випадках може бути прийняте рішення про припинення роботи над нею, якщо вона виявиться збитковою, неефективною. Визначаються джерела фінансування програми.





### 2.5.1. Поняття про блоки програми

Оскільки мова програмування C++ структурована (а також об'єктно-орієнтована), то вона підтримує розроблення блоків програми. *Блок* – логічно пов'язана група програмних настанов, які обробляються як єдине ціле. У мові програмування C++ програмний блок створюється шляхом розміщення послідовності настанов між фігурними відкритою і закритою дужками. У наведеному нижче прикладі

```
if(x<10) {
    cout << "Дуже мало, спробуйте ще раз.";
    cin >> x;
}
```

обидві настанови, розташовані після **if**-настанови (всередині фігурних дужок) виконаються тільки у тому випадку, якщо значення змінної x менше 10. Ці дві настанови (разом з фігурними дужками) представляють блок коду програми. Вони становлять логічно неподільну групу: жодна з цих настанов не може виконатися без іншої. З використанням блоків коду програми багато алгоритмів реалізуються чіткіше і більш ефективно. Вони також дають змогу краще зрозуміти дійсну природу алгоритмів.

---

***Блок** – набір логічно взаємопов'язаних між собою настанов.*

---

У наведеному нижче коді програми використано блок логічного коду програми. Введіть цю програму і виконайте її, після чого Ви зрозумієте, як працює кожен блок коду програми.

**Код програми 2.12. Демонстрація механізму використання логічних блоків у коді програми**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int a, b;
    cout << "Введіть перше число: "; cin >> a;
    cout << "Введіть друге число: "; cin >> b;

    if(a < b) {
        cout << "Перше число менше від другого" << endl;
        cout << "Їх різниця дорівнює: " << b-a;
    }
```

```
        if(a >= b) {
            cout << "Перше число більше від другого, або дорівнює йому" << endl;
            cout << "Їх сума дорівнює: " << a+b;
        }

        getch(); return 0;
}
```

Ця програма пропонує користувачу ввести два числа з клавіатури. Якщо перше число менше від другого, то буде виконано обидві **cout**-настанови. Інакше обидві вони будуть пропущені, після чого перевіриться друга умова. Якщо перше число більше від другого, то буде виконано наступні дві **cout**-настанови. У будь-якому з випадків ні за яких умов не виконається тільки одна з **cout**-настанов.

### 2.5.2. Механізм використання оператора "крапки з комою" та особливості розташування настанов

У мові програмування C++ крапка з комою означає кінець настанови, тобто кожна окрема настанова має завершуватися оператором "крапкою з комою". Як було зазначено в попередніх розділах, блок – набір логічно взаємопов'язаних між собою настанов, які поміщені всередині фігурних дужок. Блок *не* завершується крапкою з комою. Оскільки блок складається з настанов, кожна з яких завершується оператором "крапкою з комою", то в додатковій крапці з комою немає ніякого сенсу. Ознакою ж кінця блоку слугує закрита фігурна дужка (навіщо ще одна ознака?).

Мова програмування C++ не сприймає кінець рядка як ознаку кінця настанови. Тому для компілятора не має значення, у якому місці рядка розташовується настанова. Наприклад, з погляду C++-компілятора, такий фрагмент програмного коду

```
x = y;
y = y + 1;
funZ(x, y);
```

аналогічний такому рядку:

```
x = y; y = y + 1; funZ(x, y);
```

### 2.5.3. Практика застосування відступів

Розглядаючи попередні приклади, Ви, ймовірно, помітили, що деякі настанови зміщені дещо до правого краю. Мова програмування C++ – мова вільної форми запису настанов, тобто її синтаксис не пов'язаний позиційними або форматними обмеженнями. Це означає, що для C++-компілятора не важливо, як будуть розташовані настанови стосовно одна до одної. Але у програмістів з роками виробився стиль застосування відступів, який значно підвищує читабельність програм. У цьому посібнику ми дотримуємося цього стилю і Вам радимо чинити так само. Згідно з цим стилем, після кожної відкритої дужки робиться черговий відступ вправо, а після кожної закритої дужки початок відступу повертається до колишнього рівня. Існують також деякі певні настанови, для яких передбачаються додаткові відступи (про них буде сказано дещо попереду).





## 2.6. Елементи визначення мови програмування C++

Якщо Ви мандруєте світом, то Вам може знадобитися послуга того, хто розмовляє англійською. У кожній мові є свій спосіб запитати: "Ви розмовляєте англійською?", наприклад, англійське "Do you speak English?", німецьке "Sprechen Sie Englisch?", французьке "Vous debitent anglais?", португальське "Voск fala ingles?". У цих прикладах різними є не лише слова, а й їхнє розташування в реченні: підмети, присудки та доповнення розміщені в різному порядку. Порядок слів визначається синтаксисом мови. Тобто, синтаксис – встановлені правила, згідно з якими у мові будується речення.

Як і у розмовній мові, кожна мова програмування має свій синтаксис. Синтаксис мови програмування – словник, граматика, правила використання слів та утворення складніших структур. Синтаксис визначає правила написання рядків коду програми та порядок їх поєднання у функціонуючу програму. Наприклад, в усіх сучасних мовах програмування можуть виконуватись оператори **if**…**ifelse**…**else**. Завдяки ним забезпечується вибір одного з двох варіантів дій, залежно від певних умов.

Вивчення мови програмування полягає у вивченні її словника, синтаксису та способу використання базових конструкцій мови. Ви маєте запам'ятати основні зарезервовані слова. Певні слова вважаються "зарезервованими", або "ключовими", оскільки вони використовуються лише як команди мови. Кожне з цих слів має спеціальне призначення. Наприклад, слова If та Else є зарезервованими в більшості мов програмування. Вони використовуються для створення в коді програми настанов, що забезпечують прийняття рішень.

Коли Ви вивчаєте мову програмування C++, окрім синтаксису, слід осягнути її функціональні можливості. Пам'ятайте: не всі мови програмування є багатофункціональними. Але всі сучасні мови програмування мають спільні риси та, як правило, схоже функціональне призначення.

### 2.6.1. Поняття про ключові слова

У стандарті мови програмування C++ визначено 63 ключових слова, які показано в табл. 2.1. Ці ключові слова (у поєднанні з синтаксисом операторів і роздільників) утворюють визначення мови програмування C++. У ранніх версіях мови програмування C++ визначено ключове слово **overload**, але тепер воно застаріло.

Необхідно мати на увазі, що у мові програмування C++ розрізняється рядкове і прописне написання букв. Ключові слова не є винятком, тобто всі вони мають бути написані рядковими буквами. Наприклад, слово **RETURN** не буде розпізнано як ключове слово **return**.

**Табл. 2.1. Ключові слова мови програмування С++**

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

### 2.6.2. Розроблення ідентифікаторів користувача

У мові програмування С++ ідентифікатором є ім'я, яке надається функції, змінній або іншому елементу, визначеному користувачем. Ідентифікатори можуть складатися з одного або декількох символів (значущими мають бути перші 1024 символи). Імена змінних повинні починатися з букви або символу підкреслення. Подальшим символом може бути буква, цифра і символ підкреслення. Символ підкреслення можна використовувати для поліпшення читабельності імені змінної, наприклад first_name. У мові програмування С++ прописні і рядкові букви сприймаються як різні символи, тобто myvar і MyVar – різні імена. Ось декілька прикладів допустимих ідентифікаторів:

first x Addrl MaxLoad name23 top my_varsample13

У мові програмування С++ не можна використовувати як ідентифікатори ключові слова. Не можна також використовувати як ідентифікатори імена стандартних функцій (наприклад, **abs**). Необхідно пам'ятати, що ідентифікатор не повинен починатися з цифри. Так, 12x – неприпустимий ідентифікатор. Зазвичай, програміст має можливість самостійно називати змінні та інші програмні елементи на свій розсуд, але переважно ідентифікатор повинен відображати призначення або змістову характеристику елемента, якому він належить.

### 2.6.3. Механізм використання стандартної бібліотеки

У прикладах програм, представлених у цьому розділі, використовувалася функція **abs**(). По суті функція **abs**() не є частиною мови програмування С++, але її "знає" кожен С++-компілятор. Ця функція, як і багато інших, входить до складу *стандартної бібліотеки.* У прикладах цього навчального посібника ми детально розглянемо використання багатьох бібліотечних функцій мови програмування С++.





У мові програмування С++ визначено достатньо великий набір функцій, які містяться в стандартній бібліотеці. Ці функції призначені для виконання різних задач, що часто трапляються у процесі програмування, в т.ч. операції введення-виведення даних, математичні обчислення і оброблення рядків. Під час використання програмістом бібліотечної функції компілятор автономно зв'язує об'єктний код цієї функції з об'єктним кодом програми.

***Стандартна бібліотека*** *мови програмування С++ містить багато вбудованих функцій, які програмісти можуть використовувати у своїх програмах.*

Оскільки стандартна бібліотека мови програмування С++ достатньо велика, то в ній можна знайти багато корисних функцій, якими дійсно часто користуються програмісти. Бібліотечні функції можна застосовувати подібно до будівельних блоків, з яких зводиться будівля. Щоб не "винаходити велосипед", ознайомтеся з документацією на бібліотеку використовуваного Вами компілятора. Якщо Ви самі напишете функцію, яка "переходитиме" з Вами від програми у програму, її також можна помістити в бібліотеку.

Крім бібліотеки функцій, кожен С++-компілятор також містить *бібліотеку класів*, яка є об'єктно-орієнтованою бібліотекою. Нарешті, у мові програмування С++ визначено стандартну бібліотеку шаблонів (Standard Template Library – бібліотека STL). Вона надає процедури "багатократного використання", які можна налаштовувати відповідно до конкретних вимог. Але, перш ніж застосовувати бібліотеку класів або бібліотеку STL, нам необхідно познайомитися з класами, об'єктами і зрозуміти, у чому полягає суть шаблона.

# Розділ 3. ОСНОВНІ ТИПИ ДАНИХ У МОВІ ПРОГРАМУВАННЯ С++

Як зазначалося у розд. 2, всі змінні у мові програмування С++ мають бути оголошені до їх використання. Це необхідно для компілятора, якому потрібно мати інформацію про типи даних, які містяться в змінних. Тільки у цьому випадку компілятор зможе належним чином скомпілювати настанови, у яких використовуються змінні. У мові програмування С++ визначено сім основних типів даних: символьний, символьний двобайтовий, цілочисельний, з плинною крапкою, з плинною крапкою подвійної точності, логічний (або булевий) і такий, що "не має значення". Для оголошення змінних цих типів використовують ключові слова **char wchar_t**, **int**, **float**, **double**, **bool** і **void** відповідно. Типи і розміри значень в бітах і діапазони представлення для кожного з цих семи типів наведено в табл. 3.1. Необхідно пам'ятати, що розміри і діапазони, які використовуються Вашим компілятором, можуть відрізнятися від наведених у цьому посібнику. Найбільша відмінність існує між 16- і 32-розрядними середовищами: для представлення цілочисельного значення в 16-розрядному середовищі використовується, як правило, 16 біт, а в 32-разрядному" – 32.

Змінні типу **char** використовуються для зберігання 8-розрядних ASCII-символів (наприклад букв Л, Б або В) або будь-яких інших 8-розрядних значень. Щоб задати символ, необхідно помістити його в одинарні лапки. Тип **wchar_t** призначений для зберігання символів, що входять до складу великих символьних наборів. Ймовірно, Вам відомо, що в деяких природних мовах (наприклад китайській) визначено дуже велику кількість символів, для яких 8-розрядного представлення (забезпечуване типом **char**) зовсім недостатньо. Для вирішення проблем такого роду у мові програмування С++ і був доданий тип **wchar_t**, який Вам стане у пригоді, якщо Ви плануєте виходити з своїми програмами на міжнародний ринок.

Змінні типу **int** дають змогу зберігати цілочисельні значення (що не містять дробових компонентів). Змінні цього типу часто використовують для керування циклами і в логічних настановах. До змінних типу **float** і **double** звертаються або для оброблення чисел з дробовою частиною, або у разі потреби виконання операцій над дуже великими чи дуже малими числами. Типи **float** і **double** відрізняються значенням найбільшого (і найменшого) числа, які можна зберігати за допомогою змінних цих типів. Як це показано в табл. 3.1, тип **double** у мові програмування С++ дає змогу зберігати число, що приблизно вдесятеро перевищує значення типу **float**.

Тип **bool** призначений для зберігання булевих (тобто ІСТИНА/ФАЛЬШ) значень. У мові програмування С++ визначені дві булеві константи: **true** і **false**, що є єдиними значеннями, які можуть мати змінні типу **bool**. Як уже розглядалося вище, тип **void** використовують для оголошення функції, яка не





повертає значення. Інші можливості використання типу **void** розглядаються нижче у цьому посібнику.

**Табл. 3.1. Основні типи даних у мові програмування С++, типові розміри значень та діапазони їх представлення**

| Тип | Розмір у бітах | Діапазон |
|---|---|---|
| **char** | 8 | -127 ÷ +127 або 0 ÷ +255 |
| **wchar_t** | 16 | 0 ÷ +45535 |
| **int** (16-розрядне середовище) | 16 | -32768 ÷ +32767 |
| **int** (32-розрядне середовище) | 32 | -2147483648 ÷ +2147483647 |
| **float** | 32 | 3.4E-38 ÷ 3.4E+38 |
| **double** | 64 | 1,7E-308 ÷ 1,7E+308 |
| **bool** | - | **true** або **false** |
| **void** | - | Без значення |

## 3.1. Оголошення змінних

Загальний формат настанови оголошення змінних має такий вигляд:

*тип перелік_змінних x*;

У цьому записі елемент *тип* означає допустимий у мові програмування С++ тип даних, а елемент *перелік_змінних* може складатися з одного або декількох імен (ідентифікаторів), розділених між собою комами. Ось декілька прикладів оголошень змінних:

```
int     c, d, f;
char    ch, chr;
float   f, balance;
double  d;
```

Згідно зі стандартом мови програмування С++, перші 1024 символи будь-якого імені (у тому числі і імені змінної) є значущими. Це означає, що коли двоє імен відрізняються між собою хоча б одним символом з перших 1024, то компілятор розглядатиме їх як різні імена.

*У мові програмування С++ ім'я будь-якої змінної ніяк не пов'язане з її типом даних, які вона може зберігати.*

Змінні можуть бути оголошені всередині функцій, у визначенні параметрів функцій і поза всіма функціями. Залежно від місця оголошення вони називаються локальними змінними, формальними параметрами і глобальними змінними відповідно. Про важливість цих трьох типів змінних ми поговоримо далі, а поки що стисло розглянемо кожен x них окремо.

### 3.1.1. Локальні змінні

Змінні, які оголошуються усередині функції, називаються локальними. Їх можуть використовувати тільки настанови, що належать тілу цієї функції. Локальні змінні, оголошені в одній функції, невідомі жодним зовнішнім функціям. Розглянемо конкретний приклад.

**Код програми 3.1. Демонстрація механізму використання локальних змінних**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void FunC();                 // Попереднє оголошення прототипу функції FunC()

int main()
{
    int x;                   // Локальна змінна для функції main().

    x = 10;
    FunC(); cout << endl;
    cout << x;               // Виводиться число 10.

    getch(); return 0;
}

void FunC()                  // Визначення функції FunC()
{
    int x;                   // Локальна змінна для функції FunC()
    x = -199;
    cout << x;               // Виводиться число -199.
}
```

У цьому коді програми цілочисельна змінна з іменем x оголошена двічі: спочатку у функції **main**(), а потім у функції FunC(). Але змінна x з функції **main**() не має жодного стосунку до змінної x з функції FunC(). Іншими словами, зміни, яким піддається змінна x з функції FunC(), ніяк не позначаються на змінній x з функції **main**(). Тому наведена вище програма виведе на екран числа -199 і 10.

---

*Локальна змінна відома тільки функції, у якій вона визначена.*

У мові програмування C++ локальні змінні створюються під час виклику функції та руйнуються при виході з неї. Те саме можна сказати і про пам'ять, що виділяється для локальних змінних: під час виклику функції в неї записуються відповідні значення, а при виході з функції пам'ять звільняється. Це означає, що локальні змінні не підтримують своїх значень між викликами функцій.

У деяких літературних джерелах, присвячених мові програмування C++, локальна змінна називається *динамічною* або *автоматичною змінною*. Але у цьому посібнику ми дотримуватимемося поширенішого і відомого терміну – *локальна змінна*.

### 3.1.2. Формальні параметри

Як наголошувалося в розд. 2, якщо функція має аргументи, то вони мають бути оголошені. Їх оголошення здійснюється за допомогою формальних





параметрів. Як це показано у наведеному нижче коді програми, формальні параметри оголошуються після імені функції, усередині круглих дужок.

```
int FunD(int first, int last, char ch)
{
}
```

У цьому записі функція FunD() має три параметри з іменами first, last і ch. За допомогою такого оголошення ми повідомляємо компіляторові тип кожної із змінних, які прийматимуть значення, що передаються функції. Хоча формальні параметри виконують спеціальне завдання – отримання значень аргументів, що передаються функції, їх можна також використовувати в тілі функції як звичайні локальні змінні. Наприклад, ми можемо присвоїти їм будь-які значення або використовувати у довільних (допустимих у мові програмування C++) виразах. Але, подібно до будь-яких інших локальних змінних, їх значення втрачаються після завершення роботи функції.

---

***Формальний параметр** – локальна змінна, яка набуває значення аргументу, що передається функції.*

### 3.1.3. Глобальні змінні

Щоб наділити змінну "всепрограмною" популярністю, її необхідно зробити глобальною. На відміну від локальних, глобальні змінні зберігають свої значення протягом всього часу життя (часу існування) програми. Щоби створити глобальну змінну, її необхідно оголосити поза всіма функціями. Доступ до глобальної змінної можна отримати з будь-якої функції.

---

*Глобальні змінні відомі всій програмі.*

У наведеному нижче коді програми змінна pm оголошується поза всіма функціями. Її оголошення передує функції **main**(). Але її з таким самим успіхом можна розмістити у іншому місці, головне, щоб вона не належала якій-небудь іншій функції. Пам'ятайте: оскільки змінну необхідно оголосити до її використання, глобальні змінні найкраще оголошувати на початку програми.

**Код програми 3.2. Демонстрація механізму використання глобальних змінних**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Fun1();                 // Попереднє оголошення прототипу функції Fun1()
void Fun2();                 // Попереднє оголошення прототипу функції Fun2()

int pm;                      // Це глобальна змінна.
int main()
{
    int i;                   // Це локальна змінна.

    for(i=0; i<10; i++) {
        pm = i * 2;
```

```
            Fun1();
        }
        getch(); return 0;
}

void Fun1()                     // Визначення функції Fun1()
{
        cout << "pm: " << pm;   // Звернення до глобальної змінної.
        cout << endl;          // Виведення символу нового рядка.
        Fun2();
}

void Fun2()                     // Визначення функції Fun2()
{
        int pm;                // Це локальна змінна.
        for(pm=0; pm<3; pm++) cout << '.';
}
```

Незважаючи на те, що змінна pm не оголошується ні у основній функції **main**(), ні у функції Fun1(), обидві вони можуть її використовувати. Але у функції Fun2() оголошується локальна змінна pm. Тут під час звернення до змінної pm здійснюється доступ до локальної, а не до глобальної змінної. Важливо пам'ятати: якщо глобальна і локальна змінні мають однакові імена, то всі посилання на суперечливе ім'я змінної усередині функції, у якій визначена локальна змінна, належать локальній, а не глобальній змінній.

## 3.2. Поняття про модифікатори типів даних

У мові програмування C++ перед такими типами даних, як **char**, **int** і **double**, дозволяється використовувати модифікатори. Модифікатор слугує для зміни значення базового типу, щоб він точніше відповідав конкретній ситуації. Перерахуємо можливі модифікатори типів:

**signed**, **unsigned**, **long**, **short**

Модифікатори **signed**, **unsigned**, **long** і **short** можна застосовувати до цілочисельних базових типів. Окрім цього, модифікатори **signed** і **unsigned** можна використовувати з типом **char**, а модифікатор **long** – з типом **double**. Всі допустимі комбінації базових типів і модифікаторів для 16- і 32-розрядних середовищ наведено в табл. 3.2 і 3.3. У цих таблицях також вказано типи і розміри значень в бітах і діапазони представлення для кожного типу. Безумовно, реальні діапазони, що підтримуються Вашим компілятором, необхідно уточнити у відповідній документації.

Вивчаючи ці таблиці, зверніть увагу на кількість бітів, що виділяються для зберігання коротких, довгих і звичайних цілочисельних значень. Зауважте: у більшості 16-розрядних середовищ розмір (у бітах) звичайного цілочисельного значення збігається з розміром короткого цілого. Також зверніть увагу на те, що в більшості 32-розрядних середовищ розмір (у бітах) звичайного цілочисельного значення збігається з розміром довгого цілого.





**Табл. 3.2. Допустимі комбінації базових типів і модифікаторів для 16-розрядного середовища**

| Тип | Розмір у бітах | Діапазон |
|---|---|---|
| **char** | 8 | -128 ÷ +127 |
| **unsigned char** | 8 | 0 ÷ +255 |
| **signed char** | 8 | -128 ÷ +127 |
| **int або enum** | 16 | -32 768 ÷ +32 767 |
| **unsigned int** | 16 | 0 ÷ +65 535 |
| **signed int** | 16 | Аналогічний типу **int** |
| **short int** | 16 | Аналогічний типу **int** |
| **unsigned short int** | 16 | Аналогічний типу **unsigned int** |
| **signed short int** | 16 | Аналогічний типу **short int** |
| **long int** | 32 | -2 147 483 648 ÷ +2 147 483 647 |
| **signed long int** | 32 | Аналогічний типу **long int** |
| **unsigned long int** | 32 | 0 ÷ +4 294 967 295 |
| **float** | 32 | 3,4E-38 ÷ 3,4E+38 |
| **double** | 64 | 1,7E-308 ÷ 1,7E+308 |
| **long double** | 80 | 3,4E-4932 ÷ 1,1E+4932 |

**Табл. 3.3. Всі допустимі комбінації базових типів і модифікаторів для 32-розрядного середовища**

| Тип | Розмір у бітах | Діапазон |
|---|---|---|
| **char** | 8 | -128 ÷ +127 |
| **unsigned char** | 8 | 0 ÷ +255 |
| **signed char** | 8 | -128 ÷ +127 |
| **int** | 32 | -2 147 483 648 ÷ +2 147 483 647 |
| **unsigned int** | 32 | 0 ÷ +4 294 967 295 |
| **signed int** | 32 | Аналогічний типу **int** |
| **short int** | 16 | -32 768 ÷ +32 767 |
| **unsigned short int** | 16 | 0 ÷ +65 535 |
| **signed short int** | 16 | -32 768 ÷ +32 767 |
| **long int** | 32 | -2 147 483 648 ÷ +2 147 483 647 |
| **signed long int** | 32 | Аналогічний типу **signed int** |
| **unsigned long int** | 32 | Аналогічний типу **unsigned int** |
| **float** | 32 | 3,4E-38 ÷ 3,4E+38 |
| **double** | 64 | 1,7E-308 ÷ 1,7E+308 |
| **long double** | 80 | 3,4E-4932 ÷ 1,1E+4932 |
| **long long** | 100 | ±9 223 372 036 854 775 808 |

"Собака заритий" в C++-визначенні базових типів. Згідно зі стандартом мови програмування C++, розмір довгого цілого повинен бути не меншим за розмір звичайного цілочисельного значення, а розмір звичайного цілочисельного значення повинен бути не меншим від розміру короткого цілого. Розмір звичайного цілочисельного значення повинен залежати від середовища виконання. Це означає, що в 16-розрядних середовищах для зберігання значень типу **int** використовується 16 біт, а в 32-розрядних – 32 біти. При цьому найменший допустимий розмір для цілочисельних значень в будь-якому середовищі повинен становити 16 біт. Оскільки стандарт мови програмування

C++ визначає тільки відносні вимоги до розміру цілочисельних типів, то немає гарантії, що один тип буде більший (за кількістю бітів), ніж інший. Проте розміри, вказані в обох таблицях, справедливі для багатьох компіляторів.

Незважаючи на дозвіл мови C++, використання модифікатора **signed** для цілочисельних типів є надлишковим, оскільки оголошення за замовчуванням передбачає значення зі знаком. Строго кажучи, тільки конкретна реалізація визначає, яким буде **char**-оголошення: зі знаком чи без нього. Але для більшості компіляторів під оголошенням типу **char** розуміють значення зі знаком. Таким чином, у таких середовищах використання модифікатора **signed** або **char**-оголошення також є надлишковим. У цьому посібнику вважається, що **char**-значення мають знак.

Відмінність між цілочисельними значеннями із знаком і без нього полягає в інтерпретації старшого розряду. Якщо задано цілочисельне значення із знаком, C++-компілятор згенерує код з урахуванням того, що старший розряд значення використовується як прапорець знаку. Якщо прапорець знаку дорівнює 0, то число вважається позитивним, а якщо він дорівнює 1 – негативним. Негативні числа майже завжди представляються в додатковому коді. Для отримання додаткового коду програми всі розряди числа беруться в зворотному коді, а потім отриманий результат збільшується на одиницю.

Цілочисельні значення із знаком ("+" чи "-") використовуються в багатьох алгоритмах, але максимальне число, яке можна представити із знаком, становить тільки половину від максимального числа, яке можна представити без знаку. Розглянемо, наприклад, максимально можливе 16-розрядне ціле число (32 767): 01111111 11111111

Якби старший розряд цього значення із знаком дорівнював 1, то воно б інтерпретувалося як -1 (у додатковому коді). Але, якщо оголосити його як **unsigned int**-значення, то після встановлення його старшого розряду в +1 ми отримали б число 65 535.

Щоб зрозуміти відмінність в C++-інтерпретації цілочисельних значень із знаком і без нього, виконаємо таку коротку програму.

**Код програми 3.3. Демонстрація механізму реалізації відмінності між signed- і unsigned-значеннями цілого типу**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    short int c;             // Коротке int-значення із знаком
    short unsigned int d;    // Коротке int-значення без знаку

    d = 60000;
    c = d;
    cout << c << " " << d;

    getch(); return 0;
}
```





У процесі виконання програма виведе два числа:

-5536 60000

Йдеться про те, що бітова комбінація, яка представляє число 60000 як коротке цілочисельне значення без знаку, інтерпретується як коротке **int**-значення із знаком як число -5536.

У мові програмування C++ передбачено скорочений спосіб оголошення **unsigned**-, **short**- і **long**-значень цілого типу. Це означає, що під час оголошення **int**-значень достатньо використовувати слова **unsigned**, **short** і **long**, не вказуючи тип **int**, тобто тип **int** мається на увазі. Наприклад, такі дві настанови оголошують цілочисельні змінні без знаку:

```
unsigned x; unsigned int y;
```

Змінні типу **char** можна використовувати не тільки для зберігання ASCII-символів, але і для зберігання числових значень. Змінні типу **char** можуть містити "невеликі" цілі числа в діапазоні -128 ÷ +127 і тому їх можна використовувати замість **int**-змінних, якщо Вас влаштовує такий діапазон представлення чисел. Наприклад, у наведеному нижче коді програми **char**-змінну використовують для керування циклом, який виводить на екран алфавіт англійської мови.

**Код програми 3.4. Демонстрація механізму реалізації можливості виведення алфавіту в зворотному порядку**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    char letter;

    for(letter = 'z'; letter >= 'A'; letter--) cout << letter;

    getch(); return 0;
}
```

Якщо цикл **for** у наведеному прикладі здається дещо дивним, то врахуйте, що символ 'A' представляється в комп'ютері як число, а значення від 'z ' до 'A' є послідовними і розташовані в спадному порядку.

## 3.3. Поняття про літерали

Вище було використано літерали в усіх попередніх прикладах програм. А зараз настав час вивчити їх детальніше.

***Літерали** (ще називаються константами) – фіксовані значення, які не можуть бути змінені програмою.*

Константи можуть мати будь-який базовий тип даних. Спосіб представлення кожної константи залежить від її типу. Символьні константи поміщають в одинарні лапки. Наприклад 'a' і '%' є символьними літералами. Якщо

необхідно присвоїти символ змінної типу **char**, використовується настанова, подібна до такої: ch = 'z';

Щоб використовувати двобайтовий символьний літерал (тобто константу типу **wchar_t**), необхідно, щоби потрібному символу передувала буква L. Наприклад, так:

```
wchar_t wc;
wc = L'A';
```

У цьому записі змінній wc присвоюється двобайтова символьна константа, еквівалентна букві A.

Цілочисельні константи задають як числа без дробової частини. Наприклад, 10 і -100 – цілочисельні літерали. Дійсні літерали повинні містити десяткову крапку, за якою знаходиться дробова частина числа, наприклад 11.123. Для дійсних констант можна також використовувати експоненціальне представлення чисел.

Існує два основні дійсні типи: **float** і **double**. Окрім цього, існує декілька модифікацій базових типів, які утворюються за допомогою модифікаторів типів. Цікаво, а як же компілятор визначає тип літерала? Наприклад, число 123.23 має тип **float** або **double**? Відповідь на це запитання складається з двох частин. По-перше, C++-компілятор автономно робить певні припущення щодо літералів. По-друге, при бажанні програміст може безпосередньо вказати тип літерала. За замовчуванням компілятор пов'язує цілочисельний літерал з сумісним і одночасно найменшим за займаною пам'яттю тип даних, починаючи з типу **int**. Отже, для 16-розрядних середовищ число 10 буде пов'язано з типом **int**, а 103 000 – з типом **long int**.

Єдиним винятком з правила "найменшого типу" є дійсні (з плинною крапкою) константи, яким за замовчуванням присвоюється тип **double**. У багатьох випадках такі стандарти роботи компілятора цілком прийнятні. Проте у програміста є можливість точно визначити потрібний тип.

Щоб задати точний тип числової константи, використовується відповідний суфікс. Для дійсних типів діють такі суфікси: якщо дійсне число завершити буквою F, воно оброблятиметься з використанням типу **float**, а якщо буквою L – типу **long double**. Для цілочисельних типів суфікс U означає використання модифікатора типу **unsigned**, а суфікс L – **long**. Для задавання модифікатора **unsigned long** треба вказати обидва суфікси U і L. Нижче наведено деякі приклади використання модифікаторів.

**Табл. 3.4. Приклади використання модифікаторів для цілих і дійсних типів**

| Тип даних | Приклади констант |
|---|---|
| **int** | 1,123, 21000, -234 |
| **long int** | 35000L, -34L |
| **unsigned int** | 10000U, 987U, 40000U |
| **unsigned long** | 12323UL, 900000UL |
| **float** | 123.23F, 4.34e-3F |
| **double** | 23.23, 123123.33, –0.9876324 |
| **long double** | 1001.2L |





### 3.3.1. Шістнадцяткові та вісімкові літерали

Іноді зручно замість десяткової системи числення використовувати вісімкову або шістнадцяткову. У вісімковій системі основою слугує число 8, а для відображення всіх чисел використовуються цифри від 0 до 7. У вісімковій системі число 10 має те саме значення, що число 8 в десятковій. Система числення з основою 16 називається *шістнадцятковою* і використовує цифри від 0 до 9 плюс букви від A до F, що означають шістнадцяткові "цифри" 10, 11, 12, 13, 14 і 15. Наприклад, шістнадцяткове число 10 дорівнює числу 16 в десятковій системі. Оскільки ці дві системи числення (шістнадцяткова і вісімкова) використовуються у програмах достатньо часто, то у мові програмування C++ дозволено при бажанні задавати цілочисельні літерали не в десятковій, а в шістнадцяткової або вісімковій системі. Шістнадцятковий літерал повинен починатися з префікса 0x (нуль і буква x) або 0X, а вісімковий – з нуля. Наведемо два приклади:

```
int hex = 0xFF;         // Число 255 в десятковій системі
int oct = 011;          // Число 9 в десятковій системі
```

### 3.3.2. Рядкові літерали

Мова програмування C++ підтримує ще один вбудований тип літерала, що називається рядковим. *Рядок* – набір символів, поміщених у подвійні лапки, наприклад "це тест". Ви вже бачили приклади рядків у деяких **cout**-настановах, за допомогою яких виводився текст на екран. При цьому зверніть увагу ось на що. Хоча мова програмування C++ дає змогу визначати рядкові літерали, вона не має вбудованого рядкового типу даних. Рядки у мові програмування C++, як буде показано далі у цьому посібнику, підтримуються у вигляді символьних масивів[1].

***О! ережно!*** *Не варто плутати рядки з символами. Символьний літерал поміщається в одинарні лапки, наприклад 'a'. Проте "a" − вже рядок, що містить тільки одну букву.*

### 3.3.3. Символьні керівні послідовності

З виведенням більшості друкованих символів чудово справляються символьні константи, поміщені в одинарні лапки, але є такими "примірники" (наприклад, символ повернення каретки), які неможливо ввести в початковий код програми з клавіатури. Деякі символи (наприклад, одинарні та подвійні лапки) у мові програмування C++ мають спеціальне призначення, тому іноді їх не можна ввести безпосередньо. З цієї причини у мові програмування C++ дозволено використовувати ряд спеціальних символьних послідовностей (що містять символ "зворотна коса риска"), які також називаються символьними керівними послідовностями. Їх перелік наведено в табл. 3.5.

[1] Окрім цього, стандарт мови програмування C++ підтримує рядковий тип за допомогою бібліотечного класу **string**, який також описаний нижче в цьому навчальному посібнику.

**Табл. 3.5. Символьні керівні послідовності**

| Код | Значення |
|---|---|
| \b | Повернення на одну позицію |
| \f | Подача сторінки (для переходу до початку наступної сторінки) |
| \n | Новий рядок |
| \r | Повернення каретки |
| \t | Горизонтальна табуляція |
| \" | Подвійна лапка |
| \' | Одинарна лапка (апостроф) |
| \\ | Зворотна коса межа |
| \v | Вертикальна табуляція |
| \a | Звуковий сигнал (дзвінок) |
| \? | Знак, запитання |
| \N | Вісімкова константа (де N – сама вісімкова константа) |
| \xN | Шістнадцяткова константа (де N – сама шістнадцяткова константа) |

Використання керівних послідовностей продемонструємо на прикладі наведеної нижче програми. Під час її виконання буде виконано повідомлення про перехід на новий рядок, виведено символ зворотної косої риски і виконано повідомлення про повернення на одну позицію.

**Код програми 3.5. Демонстрація механізму використання символьних керівних послідовностей**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
      cout << "\n" << "\\b";
      getch(); return 0;
}
```

## 3.4. Механізм ініціалізації змінних

Під час оголошення змінної можна присвоїти їй певне значення, тобто ініціалізувати її, записавши після імені знак рівності та початкове значення. Загальний формат ініціалізації має такий вигляд:

*тип ім'я_змінної = значення*;

Ось декілька прикладів.

```
char ch = 'a';
int first = 0;
float balance = 123.23F;
```

Незважаючи на те, що змінні часто ініціалізують константами, мова програмування C++ дає змогу ініціалізувати змінні динамічно, тобто за допомогою будь-якого виразу, дійсного на момент ініціалізації. Як буде показано далі, ініціалізація має важливе значення під час роботи з об'єктами.

Глобальні змінні ініціалізувалися тільки на початку програми. Локальні змінні ініціалізувалися під час кожного входження у функцію, у якій вони





оголошені. Всі глобальні змінні ініціалізувалися нульовими значеннями, якщо не вказані ніякі інші ініціалізації. Неініціалізовані локальні змінні матимуть невідомі значення до першої настанови присвоєння, у якій вони використовуються.

Розглянемо простий приклад ініціалізації змінних. У наведеному нижче коді програми використано функцію total(), яка призначена для обчислення суми всіх послідовних чисел, починаючи з одиниці і закінчуючи числом, переданим їй як аргумент. Наприклад, сума ряду чисел, обмеженого числом 3, дорівнює 1 + 2 + 3 = 6. В процесі обчислення підсумкової суми функція total() відображає проміжні результати. Зверніть увагу на використання змінної sum у функції total().

**Код програми 3.6. Демонстрація механізму ініціалізації змінних**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void total(int x);           // Попереднє оголошення прототипу функції total()
int main()
{
      cout << "Обчислення суми чисел від 1 до 5" << endl; total(5);
      cout << " Обчислення суми чисел від 1 до 6" << endl; total(6);
      getch(); return 0;
}

void total(int x)            // Визначення функції total()
{
      int sum = 0;           // Ініціалізація змінної sum
      int i, pm;
      for(i=1; i<=x; i++) {
            sum = sum + i;
            for(pm=0; pm<10; pm++) cout << '.';
                  cout << "Проміжна сума = " << sum << endl;
      }
}
```

Результати виконання цієї програми є такими:

```
Обчислення суми чисел від 1 до 5.
.....Проміжна сума = 1
.....Проміжна сума = 3
.....Проміжна сума = 6
.....Проміжна сума = 10
.....Проміжна сума = 15
Обчислення суми чисел від 1 до 6.
.....Проміжна сума = 1
.....Проміжна сума = 3
.....Проміжна сума = 6
.....Проміжна сума = 10
.....Проміжна сума = 15
.....Проміжна сума = 21
```

Як видно з результатів розрахунку, під час кожного виклику функції total() змінна sum ініціалізується нулем.

## 3.5. Оператори С++-програми

Оператори програми, написаної мовою С++, керують процесом її виконання. У мові С++, як і в інших мовах програмування, є ряд операторів, за допомогою яких можна виконувати цикли, вказувати іншим операторам для виконання та передавати керування на іншу ділянку програми. У мові програмування С++ є такі оператори: оператор **break**; оператор **goto** і оператори з мітками; складений оператор; оператор **if**; оператор continue; порожній оператор; оператор **do**; оператор **return**; оператор **expression**; оператор **switch**; оператор **for**; оператор **while**.

Оператори мови С++ складаються з ключових слів, виразів і інших операторів. При виконанні програми, написаної мовою С++, її оператори виконуються в тому порядку, в якому вони з'являються в програмі, якщо немає оператора, який би явно передавав керування в інше місце програми.

### 3.5.1. Поняття про вбудовані оператори

У мові програмування С++ визначено широкий набір вбудованих операторів, які дають в руки програмісту потужні важелі керування при створенні і обчислення різноманітних виразів. Оператор (**operator**) – символ, який вказує компіляторові на виконання конкретних математичних дій або логічних маніпуляцій. У мові програмування С++ є чотири загальні класи операторів: *арифметичні*, *порозрядні*, *логічні* та *оператори відношення*. Окрім них визначено інші оператори спеціального призначення. У цьому розділі розглядаються арифметичні, логічні оператори та оператори відношення.

### 3.5.2. Арифметичні оператори

У табл. 3.6 перераховано арифметичні оператори, дозволені для застосування у мові програмування С++. Дія операторів +; * і / збігається з дією аналогічних операторів у будь-якій іншій мові програмування (та і в алгебрі, якщо вже на те пішло). Їх можна застосовувати до даних будь-якого вбудованого числового типу. Після застосування оператора ділення (/) до цілого числа залишок буде відкинутий. Наприклад, результат цілочисельного ділення 10/3 буде дорівнювати 3.

**Табл. 3.6. Арифметичні оператори**

| Оператор | Дія |
|---|---|
| + | додавання |
| - | віднімання, а також унарний мінус |
| * | множення |
| / | ділення |
| % | ділення за модулем |
| -- | декремент |
| ++ | інкремент |





Залишок від ділення можна отримати за допомогою оператора ділення за модулем (%). Цей оператор працює практично так само, як у інших мовах програмування: повертає залишок від ділення без остачі. Наприклад, 10 % 3 дорівнює 1. Це означає, що у мові програмування С++ оператора "%" не можна застосовувати до типів з плинною крапкою (**float** або **double**). Ділення за модулем застосовано тільки до цілочисельних типів. Використання цього оператора продемонстровано у наведеному нижче коді програми.

**Код програми 3.7. Демонстрація механізму використання оператора "ділення за модулем"**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int x = 10, y = 3;

    cout << x/y;             // Буде відображене число 3.
    cout << endl;
    cout << x%y;             /* Буде відображене число 1,
                                тобто залишок від ділення без остачі */
    cout << endl;
    x = 1; y = 2;
    cout << x/y << " " << x%y; // Будуть виведені числа 0 і 1.

    getch(); return 0;
}
```

У останньому рядку результатів виконання цієї програми дійсно будуть виведені числа 0 і 1, оскільки при цілочисельному діленні 1/2 отримаємо 0 із залишком 1, тобто вираз 1%2 дає значення 1.

Унарний мінус, по суті, є множенням значення свого єдиного операнда на -1. Іншими словами, будь-яке числове значення, якому передує знак "-", змінює свій знак на протилежний.

### 3.5.3. Оператори інкремента і декремента

У мові програмування С++ є два оператори, яких немає в деяких інших мовах програмування. Це оператори інкремента (++) і декремента (--). Вони згадувалися в розд. 2.4.2, коли йшлося про настанову організації циклу **for**. Оператор інкремента здійснює додавання до операнда число 1, а оператор декремента віднімає 1 від свого операнда. Це означає, що настанова

```
x = x + 1;
```

аналогічна такій настанові:

```
++x;
```

А настанова

```
x = x – 1;
```

аналогічна такій настанові:

```
--x;
```

Оператори інкремента і декремента можуть знаходитися як перед своїм операндом (*префіксна форма*), так і після нього (*постфіксна форма*). Наприклад, настанову

```
x = x + 1;
```

можна переписати у вигляді префіксної форми

```
++x;          // Префіксна форма оператора інкремента.
```

або у вигляді постфіксної форми:

```
x++;          // Постфіксна форма оператора інкремента.
```

У попередньому прикладі не мало значення, у якій формі було застосовано оператор інкремента: префіксній або постфіксній. Але, якщо оператор інкремента або декремента використовується як частина більшого виразу, то форма його застосування дуже важлива. Якщо такий оператор застосовується в префіксній формі, то мова програмування C++ спочатку виконає цю операцію, щоб операнд набув нового значення, яке потім буде використано іншою частиною виразу. Якщо ж оператор застосовується в постфіксній формі, то C# використовує у виразі його старе значення, а потім виконає операцію, внаслідок якої операнд знайде нове значення. Для розуміння сказаного розглянемо такий фрагмент коду програми:

```
x = 10;
y = ++x;
```

У цьому випадку значення змінної y буде дорівнювати 11. Але, якщо у цьому коді префіксну форму запису замінити постфіксною, то значення змінної y буде дорівнювати 10:

```
x = 10;
y = x++;
```

У обох випадках змінна x набуде значення 11. Різниця полягає тільки у тому, в який момент вона дорівнюватиме 11 (до або після присвоєння її значення змінній y). Для програміста надзвичайно важливо мати можливість керувати тривалістю виконання операції інкремента або декремента.

Більшість C++-компіляторів для операцій інкремента і декремента створюють ефективніший код порівняно з кодом, що генерується під час використання звичайного оператора додавання і віднімання одиниці. Тому професіонали вважають за краще використовувати (де це можливо) оператори інкремента і декремента.

**Табл. 3.7. Порядок виконання дій арифметичними операторами**

| Пріоритет | Оператори |
|---|---|
| Найвищий | ++ -- |
| | - (унарний мінус) |
| | * / % |
| Нижчий | + – |

Оператори одного ієрархічного рівня обчислюються компілятором зліва направо. Безумовно, для зміни порядку обчислень можна використовувати





круглі дужки, які обробляються у мові програмування C++ так само, як практично в усіх інших мовах програмування. Операції або набір операцій, поміщених у круглі дужки, набувають вищий пріоритет порівняно з іншими операціями виразу. Унарні оператори навпаки – виконуються справа наліво.

### 3.5.4. Оператори відношення та логічні оператори

Оператори відношення та логічні (булеві) оператори, які часто йдуть "рука в руку", використовуються для отримання результатів у вигляді значень ІСТИНА/ФАЛЬШ. Оператори відношення оцінюють за "двобальною системою" відношення між двома значеннями, а логічні визначають різні способи поєднання дійсних і помилкових значень. Оскільки оператори відношення генерують ІСТИНА/ФАЛЬШ – результати, то вони часто виконуються з логічними операторами. Тому вони і розглядаються у одному розділі.

**Табл. 3.8. Оператори відношення та логічні оператори**

| **Оператори відношення** | **Значення** |
|---|---|
| == | дорівнює |
| != | не дорівнює |
| > | більше |
| < | менше |
| >= | більше або дорівнює |
| <= | менше або дорівнює |
| **Логічні оператори** | **Значення** |
| && | І |
| \|\| | АБО |
| ! | НЕ |

Операторы відношення та логічні (булеві) оператори перераховано в табл. 3.8. Зверніть увагу на те, що у мові програмування C++ як оператор відношення "не дорівнює" використовує символ " !=", а для оператора "дорівнює" – подвійний символ рівності (==). Згідно зі стандартом мови програмування C++, результат виконання операторів відношення і логічних операторів має тип **bool**, тобто у процесі виконання операцій відношення і логічних операцій виходять значення **true** або **false**. Під час використання старших компіляторів результати виконання цих операцій мали тип **int** (нуль або ненульове ціле, наприклад 1). Ця відмінність в інтерпретації значень має в основному теоретичну основу, оскільки мова програмування C++ автоматично перетворює значення **true** в 1, а значення **false** – в 0, і навпаки.

***Нео! хідноапам'ятати!*** *У мові програмування C++ будь-яке ненульове число оцінюється як* ***true****, а нуль – як* ***false****.*

Операнди, що беруть участь в операціях "з'ясування" відношення, можуть мати практично будь-який тип, головне, щоб їх можна було порівнювати. Що стосується логічних операторів, то їх операнди повинні мати тип **bool**, і результат логічної операції завжди матиме тип **bool**. Оскільки у мові програмування C++ будь-яке ненульове число оцінюється як істинне (**true**), а

нуль еквівалентний помилковому значенню (**false**), то логічні оператори можна використовувати в будь-якому виразі, який дає нульовий або ненульовий результат.

Логічні оператори використовують для підтримки базових логічних операцій І, АБО і НЕ відповідно до такої таблиці істинності. У ній 1 використовується як значення ІСТИНА, а 0 – як значення ФАЛЬШ.

| **p** | **q** | **p** І **q** | **p** АБО **q** | НЕ **p** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 |

Незважаючи на те, що мова програмування С++ не містить вбудованого логічного оператора, що "виключає АБО" (XOR), його неважко "створити" на основі вбудованих. Подивіться, як наведено нижче функція використовує оператори І, АБО і НЕ для виконання операції, що "виключає АБО":

```
bool XOR (bool a, bool b)
{
    return (a || b) &&!(a && b);
}
```

Ця функція використовується у наведеному нижче коді програми. Вона відображає результати застосування операторів І, АБО і що "виключає АБО" до значень, що вводяться Вами ж[1].

**Код програми 3.8. Демонстрація механізму використання логічної операції** XOR**()**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

bool XOR(bool a, bool b);    // Попереднє оголошення логічної функції
int main()
{
    bool p, q;
    cout << "Введіть P (0 або 1): "; cin >> p;
    cout << "Введіть Q (0 або 1): "; cin >> q;
    cout << "P І Q: " << (p && q) << endl;
    cout << "P АБО Q: " << (p || q) << endl;
    cout << "P XOR Q: " << XOR(p, q) << endl;

    getch(); return 0;
}

bool XOR(bool a, bool b)             // Визначення логічної функції
{
    return (a || b) &&!(a && b);
}
```

[1] Необхідно пам'ятати, що тут одиниця буде оброблена як значення **true**, а нуль – як **false**.





Ось як виглядає можливий результат виконання цієї програми.

```
Введіть P (0 або 1): 1
Введіть Q (0 або 1): 1
P І Q: 1
P АБО Q: 1
P XOR Q: 0
```

У цьому коді програми зверніть увагу ось на що. Хоча параметри функції XOR() вказані з типом **bool**, користувач вводить цілочисельні значення (0 або 1). У цьому немає нічого дивного, оскільки мова програмування С++ автоматично перетворить число 1 в **true**, а 0 – в **false**. І навпаки, при виведенні на екран **bool**-значення, що повертається функцією XOR(), воно автоматично перетвориться в число 0 або 1 (залежно від того, яке значення "повернулося": **false** або **true**). Цікаво відзначити, що, коли типи параметрів функції XOR() і тип значення, що повертається нею, замінити типом **int**, ця функція працюватиме абсолютно так само. Причина проста: вся справа в автоматичних перетвореннях, що виконуються С++-компілятором між цілочисельними і булевими значеннями.

Як оператори відношення, так і логічні оператори мають нижчий пріоритет порівняно з арифметичними операторами. Це означає, що такий вираз, як 10 > 1+12 буде обчислено так, як би його було записано у такому вигляді:

```
10 > (1+12)
```

Результат цього виразу, звичайно ж, дорівнює значенню ФАЛЬШ. Окрім цього, погляньте ще раз на настанови виведення результатів роботи попередньої програми на екран.

```
cout << "P І Q: " << (p && q) << endl;
cout << "P АБО Q: " << (p || q) << endl;
```

Без круглих дужок, у які поміщені вирази p && q і p || q, тут обійтися не можна, оскільки оператори && і || мають нижчий пріоритет, ніж оператор виведення даних.

За допомогою логічних операторів можна об'єднати в одному виразі будь-яку кількість операцій відношення. Наприклад, у цьому виразі об'єднано відразу три операції відношення:

```
var>15 ||!(10<pm) && 3<=item
```

У наведеній нижче таблиці наведено пріоритет операторів відношення і логічних операторів.

**Табл. 3.9. Пріоритет операторів відношення та логічних операторів**

| Пріоритет | Оператори |
|---|---|
| Найвищий | ! |
| | > >= < <= |
| | == != |
| | && |
| Нижчий | \|\| |

## 3.6. Особливості запису арифметичних виразів

Оператори, літерали і змінні – все складові арифметичних виразів. Ймовірно, Ви вже знайомі з виразами з попереднього досвіду програмування або ще зі шкільного курсу алгебри. У наступних підрозділах спробуємо коротко розглянути основні аспекти механізму запису виразів, які стосуються їх використання у мові програмування С++.

### 3.6.1. Перетворення типів у виразах

Якщо у виразі змішані різні типи літералів і змінних, то компілятор перетворить їх до одного типу. По-перше, всі **char**- і **short int**-значення автоматично перетворяться (з розширенням "типорозміру") до типу **int**. Цей процес називається цілочисельним розширенням (integer promotion). По-друге, всі операнди перетворяться (також з розширенням "типорозміру") до типу найбільшого операнда. Цей процес називається розширенням типу (type promotion), причому він здійснюється по-операційно. Наприклад, якщо один операнд має тип **int**, а інший – **long int**, то тип **int** розширюється в тип **long int**. Або, якщо хоча б один з операндів має тип **double**, будь-який інший операнд призводить до типу **double**. Це означає, що такі перетворення, як з типу **char** в тип **double**, цілком допускаються. Після перетворення обидва операнди матимуть один і той самий тип, а результат виконання операції – тип, який збігається з типом операндів.

```
char ch;
int i;
float f;
double d;

result=(ch/i) + (f*d) - (f+i)
```

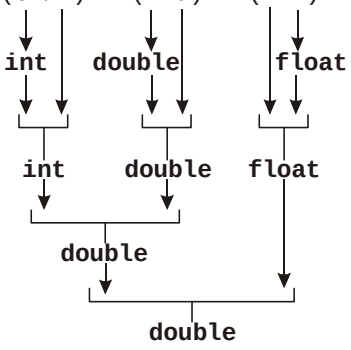



**Рис. 3.1. Приклад перетворення типів у мові програмування С++**

Розглянемо, наприклад, перетворення типів, які схематично представлено на рис. 3.1. Спочатку символ ch піддається процесу "розширення" типу і перетвориться в значення типу **int**. Потім результат виконання операції ch/c приводиться до типу **double**, оскільки результат добутку f*d має тип **double**. Результат виконання всього виразу отримає тип **double**, оскільки до моменту його обчислення обидва операнди матимуть тип **double**.

### 3.6.2. Перетворення, що відбуваються зі змінними типу bool

Як було зазначено вище, значення типу **bool** автоматично перетворяться в цілі числа 0 або 1 під час використання у виразі цілого типу. Під час перетворення цілочисельного результату в тип **bool** нуль перетвориться в **false**, а





ненульове значення – в **true**. І хоча тип **bool** відносно недавно був доданий в мову програмування С++, виконання автоматичних перетворень, пов'язаних з типом **bool**, означає, що його введення у мові програмування С++ не має негативних наслідків для коду програми, написаного для попередніх версій мови програмування С++. Понад це, автоматичні перетворення дають змогу мові програмування С++ підтримувати початкове визначення значень ФАЛЬШ і ІСТИНА у вигляді нуля і ненульового значення. Таким чином, тип **bool** дуже зручний для програміста.

### 3.6.3. Операція приведення типів даних

У мові програмування С++ передбачено можливість встановлення для виразу заданий тип. Для цього використовується операція приведення типів (cast). Мова програмування С++ визначає п'ять видів таких операцій. У цьому розділі розглянемо тільки один з них, а інші чотири – описано нижче у цьому посібнику (після теми побудови об'єктів).

Отже, загальний формат операції приведення типів є таким: (*тип*) *вираз*. У цьому записі елемент *тип* вказує на тип, до якого необхідно привести вираз. Наприклад, якщо виникає бажання, щоб вираз x/2 мав тип **float**, необхідно написати таке: (**float**) x/2.

Приведення типу розглядається як унарний оператор, і тому він має такий самий пріоритет, як і інші унарні оператори.

Іноді операція приведення типів виявляється дуже корисною. Наприклад, у наведеному нижче коді програми для керування циклом використовується деяка цілочисельна змінна, що входить до складу виразу, результат обчислення якого необхідно отримати з дробовою частиною.

**Код програми 13.9. Демонстрація механізму виконання операції приведення типів**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()                   // Виводимо значення і та значення i/2 з дробовою частиною.
{
    for(int i=1; i<=100; ++i)
        cout << i << "/ 2 дорівнює: " << (float) i / 2 << endl;

    getch(); return 0;
}
```

Без оператора приведення типу (**float**) виконалося б тільки цілочисельне ділення. Приведення типів у цьому випадку гарантує, що на екрані буде відображена і дробова частина результату. Аналогічний результат можна отримати, якщо записати таку настанову:

```
cout << i << "/ 2 дорівнює: " << i / 2. << endl;
```

У цій настанові число 2. означає 2.0, тобто воно є дійсним, а ділення змінної і цілого типу на константу дійсного типу дає результат також дійсного типу.

### 3.6.4. Механізм використання пропусків і круглих дужок

Будь-який вираз у мові програмування С++ для підвищення читабельності може містити пропуски (або символи табуляції). Наприклад, наступні два вирази достатньо однакові, але другий прочитати набагато легше:

```
x=10/y*(127/x);
x = 10 / y * (127 / x);
```

Круглі дужки (так само, як і у математиці) підвищують пріоритет виконання операцій, що містяться усередині них. Використання надлишкових або додаткових круглих дужок не приведе до помилки або уповільнення обчислення виразу. Іншими словами, від них не буде ніякої шкоди, та зате скільки користі! Адже вони допоможуть проясняти (у першу чергу для Вас самих, не говорячи вже про тих, кому доведеться розбиратися у цьому без Вас) точний порядок виконання обчислень. Скажіть, наприклад, який з таких двох виразів легше зрозуміти?:

```
x = y/3-34*tmp+127;
x = (y/3) – (34*tmp) + 127;
```

Зрозуміло, більшість відповість – другий. Хоча ця відповідь і є правильною, проте серед професійних програмістів це свідчитиме про не знання пріоритетів виконання дій у будь-якій мові програмування.





# Розділ 4. ПОНЯТТЯ ПРО НАСТАНОВИ КЕРУВАННЯ ХОДОМ ВИКОНАННЯ С++-ПРОГРАМИ

У цьому розділі дізнаємося, як керувати ходом виконання С++-програми. Існує три категорії керівних настанов: *настанови вибору* (**if**, **switch**), *ітераційні настанови* (що складаються з **for**-, **while**- і **do-while**-циклів) і *настанови переходу* (**break**, **continue**, **return** і **goto**). За винятком **return**, всі решта перераховані вище настанови описано у цьому розділі.

## 4.1. Механізм використання настанови вибору if

Настанова **if** була представлена у розд. 2, але тут розглянемо її детальніше. Повний формат її запису є таким:

**if**(*вираз*) *настанова*;
**else** *настанова*;

У цьому записі під елементом *настанова* розуміємо одну настанову мови програмування С++. Частина **else** необов'язкова. Замість елемента *настанова* може бути використаний *блок настанов*. У цьому випадку формат запису **if**-настанови набуде такого вигляду:

```
if(вираз) {
        послідовність настанов
}
else {
        послідовність настанов
}
```

Якщо елемент *вираз*, який є умовним виразом, під час обчислення дасть значення ІСТИНА, то буде виконана **if**-настанова; інакше – **else**-настанова (якщо така існує). Обидві настанови ніколи одночасно не виконуються. Умовний вираз, який керує виконанням **if**-настанови, може мати будь-який тип, що є дійсним для С++-виразів, але головне, щоб результат його обчислення можна було інтерпретувати як значення ІСТИНА або ФАЛЬШ.

*Логічна настанова* ***if*** *дає змогу зробити вибір між двома виконуваними гілками програми.*

Використання **if**-настанови розглянемо на прикладі коду програми, яка є версією гри "Вгадай магічне число". Програма відображає випадкове число і пропонує його вгадати. Якщо Ви відгадуєте число, то програма виводить на екран повідомлення схвалення ** Правильно **. У цьому коді програми представлена ще одна бібліотечна функція **rand**(), яка повертає випадково вибране ціле число від нуля до **RAND_MAX**. Для використання цієї функції необхідно приєднати до програми заголовок <**cstdlib**>.

**Код програми 4.1. Демонстрація механізму роботи програми "Вгадай магічне число"**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int magic;       // Магічне число
    int guess;       // Варіант користувача
    clrscr();        // Очищення екрану

    magic = rand();  // Отримуємо випадкове число.

    cout << "Введіть свій варіант магічного числа: "; cin >> guess;

    if(guess == magic) cout << "** Правильно **";

    getch(); return 0;
}
```

У цьому коді програми для перевірки того, чи збігається з "магічним числом" варіант, запропонований користувачем, використовують оператор відношення "==". У випадку збігу чисел на екран виводиться повідомлення ** Правильно **.

Спробуємо удосконалити нашу програму, тобто в її нову версію внесемо **else**-гілку для виведення повідомлення про те, що припущення користувача виявилося неправильним.

**Код програми 4.2. Демонстрація механізму роботи програми "Вгадай магічне число": перше удосконалення**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int magic;       // Магічне число
    int guess;       // Варіант користувача
    clrscr();        // Очищення екрану
    magic = rand();  // Отримуємо випадкове число.
    cout << "Введіть свій варіант магічного числа: "; cin >> guess;

    if(guess == magic) cout << "** Правильно **";
    else cout << "... Дуже шкода, але Ви помилилися.";

    getch(); return 0;
}
```





### 4.1.1. Умовний вираз

Іноді новачків у мові програмування С++ збиває з пантелику той факт, що для керування **if**-настановою можна використовувати будь-який дійсний С++-вираз. Іншими словами, тип виразу необов'язково обмежувати *операторами відношення* та *логічними операторами* або *операндами* типу **bool**. Головне, щоб результат обчислення умовного виразу можна було інтерпретувати як значення ІСТИНА або ФАЛЬШ. Як зазначалося у попередньому розділі, нуль автоматично перетвориться в **false**, а всі ненульові значення – в **true**. Це означає, що будь-який вираз, який дає внаслідок обчислення нульове або ненульове значення, можна використовувати для керування **if**-настановою. Наприклад, наведена вище програма зчитує з клавіатури два цілі числа і відображає частку від ділення першого на друге. Щоб не допустити ділення на нуль, у програмі використано **if**-настанову.

**Код програми 4.3. Демонстрація механізму використання if-настанови**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int a, b;

    cout << "Введіть два числа: "; cin >> a, b;

    if(b) cout << a/b << endl;
    else cout << "На нуль ділити не можна" << endl;

    getch(); return 0;
}
```

Зверніть увагу на те, що значення змінної b (дільник) порівнюється з нулем за допомогою настанови **if**(b), а не настанови **if**(b != 0). Йдеться про те, що, коли значення b = 0, умовний вираз, який керує настановою с, оцінюється як ФАЛЬШ, то це призводить до виконання **else**-гілки. Інакше (коли b містить ненульове значення) умова оцінюється як ІСТИНА, тобто ділення легко здійснюється. Немає ніякої потреби використовувати наступну **if**-настанову, яка до того ж не свідчить про хороший стиль програмування мовою С++:

```
if(b != 0) cout << a/b << endl;
```

Ця форма **if**-настанови вважається застарілою і потенційно неефективною.

### 4.1.2. Вкладені if-настанови

*Вкладені* **if**-настанови виникають у тому випадку, якщо елемент *настанова* (див. повний формат запису) використовується інша **if**-настанова. Вкладені **if**-настанови дуже популярні у програмуванні. Головне тут – пам'ятати, що **else**-настанова завжди належить до найближчої **if**-настанови, яка

знаходиться усередині того ж програмного блоку, але ще не пов'язана ні з якою іншою **else**-настановою. Ось приклад:

```
if(c) {
    if(d) statement1;
    if(f) statement2;   // Ця if-настанова
    else statement3;  // пов'язана з цією else-настановою.
}
else statement4; // Ця else-настанова пов'язана з if(c).
```

Як стверджується в коментарях, остання **else**-настанова не пов'язана з настановою **if**(d), оскільки вони не знаходяться в одному блоці (незважаючи те, що ця **if**-настанова – найближча, яка не має при собі "**else**-пари"). Внутрішня **else**-настанова пов'язана з настановою **if**(f), оскільки вона – найближча і знаходиться усередині того ж блоку.

---

*Вкладена **if**-настанова – настанова, яку використовують як елемент* настанови *будь-якої іншої **if**- або **else**-настанови.*

Мова програмування C++ дає змогу 256 рівнів вкладення, але на практиці рідко доводиться вкладати **if**-настанови на "таку глибину".

Продемонструємо використання вкладених настанов за допомогою чергового удосконалення програми "Вгадай магічне число" (тут гравець отримує реакцію програми на неправильну відповідь).

**Код програми 4.4. Демонстрація механізму роботи програми "Вгадай магічне число": друге удосконалення**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int magic;          // Магічне число
    int guess;          // Варіант користувача
    magic = rand();   // Отримуємо випадкове число.
    cout << "Введіть свій варіант магічного числа: "; cin >> guess;

    if(guess == magic) {
        cout << "** Правильно **" << endl;
        cout << magic << " і є те саме магічне число" << endl;
    }
    else {
        cout << "... Дуже шкода, але Ви помилилися.";
        if(guess > magic)
            cout << "Ваш варіант перевищує магічне число" << endl;
        else
            cout << "Ваш варіант є меншим від магічного числа" << endl;
    }
    getch(); return 0;
}
```





### 4.1.3. Конструкція "сходинок" if-else-if

Дуже поширеною у програмуванні конструкцією, в основі якої знаходиться вкладена **if**-настанова, є "сходинки" **if-else-if**. Її можна представити в такому вигляді:

```
if(умова)
    настанова;
else
    if(умова)
        настанова;
    else
        if(умова)
            настанова;
        else
            настанова;
```

У цьому записі під елементом *умова* розуміють умовний вираз, який обчислюється зверху вниз. Як тільки у якій-небудь гілці виявиться істинний результат, то буде виконану настанову, пов'язану з цією гілкою, а всі решта "сходинки" опускаються. Якщо виявиться, що жодна з умов не є істинною, то буде виконано останню **else**-настанову (можна вважати, що вона здійснює роль умови, яка діє за замовчуванням). Якщо останню **else**-настанову не задано, а всі інші виявилися помилковими, то взагалі ніяка дія не буде виконана. Роботу "сходинок" **if-else-if** продемонструємо на прикладі наведеного нижче коду програми.

**Код програми 4.5. Демонстрація механізму використання конструкції "сходинок" if-else-if**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    for(int x=0; x<6; x++) {
        if(x==1) cout << "x дорівнює одиниці" << endl;
        else if(x==2) cout << "x дорівнює двом" << endl;
        else if(x==3) cout << "x дорівнює трьом" << endl;
        else if(x==4) cout << "x дорівнює чотирьом" << endl;
        else cout << "x не потрапляє в діапазон від 1 до 4" << endl;
    }

    getch(); return 0;
}
```

Результати виконання цієї програми є такими:

```
x не потрапляє в діапазон від 1 до 4.
x дорівнює одиниці.
x дорівнює двом.
x дорівнює трьом.
x дорівнює чотирьом.
x не потрапляє в діапазон від 1 до 4.
```

Як бачимо, остання **else**-настанова здійснюється тільки у тому випадку, коли всі попередні **if**-умови дали помилковий результат.

## 4.2. Механізм використання настанови багатовибірного розгалуження switch

Настанова **switch** – настанова багатовибірного розгалуження, яка дає змогу вибрати одну з множини альтернатив. Хоча різнонаправлене тестування можна реалізувати за допомогою послідовності вкладених **if**-настанов, однак у багатьох ситуаціях настанова **switch** виявляється набагато ефективнішим і очевидним рішенням.

### 4.2.1. Особливості роботи настанови

Настанова багатовибірного розгалуження працює таким чином. Значення виразу послідовно порівнюється з константами із заданого переліку. Внаслідок виявлення збігу для однієї з умов порівняння здійснюється послідовність настанов, пов'язана з цією умовою. Загальний формат запису настанови **switch** є таким:

```
switch (вираз) {
    case константа1:
        послідовність настанов
        break;
    case константа2:
        послідовність настанов
        break;
    case константа3:
        послідовність настанов
        break;
    ...
    default:
        послідовність настанов
}
```

Елемент *вираз* настанови **switch** під час обчислення повинен давати цілочисельне або символьне значення. Вирази, що мають, наприклад, тип з плинною крапкою, тут не дозволені. Дуже часто як керівний **switch**-вираз використовується одна змінна.

*Настанова **break** завершує виконання коду програми, що визначається настановою **switch**.*

Послідовності настанов **default**-гілки виконуються у тому випадку, якщо жодна із заданих **case**-констант не співпаде з результатом обчислення **switch**-виразу. Гілка **default** є необов'язковою. Якщо вона відсутня, то при не співпаданні результату розрахунку виразу ні з однією з **case**-констант ніякої дії виконано не буде. Якщо такий збіг все-таки виявиться, то виконуватимуться настанови, відповідні цій **case**-гілці, доти, доки не трапиться настанова **break**





або не буде досягнуто кінець **switch**-настанови (або у **default**-, або в останній **case**-гілці).

*Настанови **default**-гілки виконуються у тому випадку, якщо жодна з **case**-констант не співпаде з результатом обчислення **switch**-виразу.*

Отже, для застосування **switch**-настанови необхідно знати таке:

- настанова **switch** відрізняється від настанови **if** тим, що **switch**-вираз можна тестувати тільки з використанням умови рівності (тобто на збіг **switch**-виразу із заданими **case**-константами), тоді як умовний **if**-вираз може бути будь-якого типу;
- ніякі дві **case**-константи в одній **switch**-настанові не можуть мати однакових значень;
- настанова **switch** зазвичай ефективніша, ніж вкладені **if**-настанови;
- послідовність настанов, пов'язана з кожною **case**-гілкою, не є блоком. Проте повна **switch**-настанова визначає блок. Значущість цього твердження стане очевидною після того, як ми більше дізнаємося про мову програмування С++.

Згідно зі стандартом мови програмування С++, **switch**-конструкція може мати не більше ніж 16 384 **case**-настанов. Але на практиці (виходячи з міркувань ефективності) зазвичай обмежуються набагато меншою їх кількістю.

Використання **switch**-настанови продемонстровано у наведеному нижче коді програми. Вона створює просту "довідкову" систему, яка описує призначення **for**-, **if**- і **switch**-настанов. Після відображення переліку пропонованих тем, згідно з якими можливе надання довідки, програма переходить в режим очікування доти, доки користувач не зробить свій вибір. Введене користувачем значення використовується в настанові **switch** для відображення інформації по вказаній темі[1].

**Код програми 4.6. Демонстрація механізму використання switch-настанови на прикладі "довідкової" системи**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int vybir;
    cout << "Довідка на теми: \n" << endl;
    cout << "1. for" << endl;
    cout << "2. if" << endl;
    cout << "3. switch\n" << endl;

    cout << "Введіть номер теми (1-3): "; cin >> vybir;
    cout << endl;

    switch(vybir) {
        case 1: cout << "for – найуніверсальніший цикл в С++" << endl;
            break;
        case 2: cout << "if – настанова умовного розгалуження" << endl;
```

[1] Пропонуємо читачу як вправу доповнити інформацію по вивчених темах, а також ввести в цю "довідкову" систему нові теми.

```
            break;
        case 3: cout << "switch – багатовибірне розгалуження" << endl;
            break;
        default: cout << "програміст повинен ввести число від 1 до 3" << endl;
    }
    getch(); return 0;
}
```

Ось один з варіантів виконання цієї програми.

```
Довідка на теми:

1. for
2. if
3. switch

Введіть номер теми (1-3): 2
if – настанова умовного розгалуження.
```

Формально настанова **break** є необов'язковою, хоча здебільшого використання **switch**-конструкцій вона наявна. Настанова **break**, що знаходиться в послідовності настанов будь-якої **case**-гілки, призводить до виходу зі всієї **switch**-конструкції та передає керування настанові, розташованій відразу після неї. Але, якщо настанова **break** в **case**-гілці відсутня, то буде виконано всі настанови, пов'язані з даною **case**-гілкою, а також всі подальші настанови, що розташовані під нею, доти, доки все-таки не трапиться настанова **break**, що належить до однієї з подальших **case**-гілок, або не буде досягнуто кінець **switch**-конструкції.

Розглянемо уважно наведену нижче програму. Спробуйте передбачити, що буде відображено на екрані під час її виконання.

**Код програми 4.7. Демонстрація механізму використання switch-конструкції**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    for(int i=0; i<5; i++) {
    switch(i) {
        case 0: cout << "менше 1" << endl;
        case 1: cout << "менше за 2" << endl;
        case 2: cout << "менше за 3" << endl;
        case 3: cout << "менше за 4" << endl;
        case 4: cout << "менше за 5" << endl;
        }
        cout << endl;
    }
    getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми:





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

менше 2
менше 3
менше 4
менше 5

менше 3
менше 4
менше 5

менше 4
менше 5

менше 5
```

З цих результатів видно, якщо настанова **break** в одній **case**-гілці відсутня, то виконуються настанови, що належать наступній **case**-гілці. Як це показано в цьому прикладі, в **switch**-конструкцію можна помістити "порожні" **case**-гілки:

```
switch(c) {
    case 1:
    case 2:
    case 3: do_something();
        break;
    case 4: do_something_else();
        break;
}
```

Якщо змінна c цей фрагмент коду програми набуває значення 1, 2 або 3, то викликається функція do_something(). Якщо ж значення змінної c дорівнює 4, то робиться звернення до функції do_something_else(). Використання "пачки" декількох порожніх **case**-гілок характерне для випадків, коли вони використовують один і той самий програмний код.

### 4.2.2. Організація вкладених настанов багатовибірного розгалуження

Настанова багатовибірного розгалуження **switch** може бути використана як частина **case**-послідовності зовнішньої настанови **switch**. У цьому випадку вона називається вкладеною настановою **switch**. Необхідно відзначити, що **case**-константи внутрішніх і зовнішніх настанов **switch** можуть мати однакові значення, при цьому жодних конфліктів не виникне. Наприклад, такий фрагмент коду програми є цілком допустимим:

```
switch(ch1) {
    case 'A1': cout << "Ця константа А – частина зовнішньої настанови switch";
```

```
            switch(ch2) {
                case 'A': cout << "Ця константа А – частина"
                                << " внутрішньої настанови switch";
                    break;
                case 'B1': //...
                    break;
                case 'C1': //...
            }
            break;
        case 'B1': //...
            break;
        case 'C1': //...
    }
```

## 4.3. Механізм використання настанови організації циклу for

У розд. 2 було використано просту форму циклу **for**. У цьому розділі розглядається цей цикл детальніше, тобто дізнаємося, наскільки потужним і гнучким засобом програмування він є. Почнемо з традиційних форм його використання.

*Цикл* ***for*** *– найуніверсальніший оператор організації циклу мови програмування С++.*

Отже, загальний формат запису циклу **for** для багатократного виконання однієї настанови має такий вигляд:

**for**(*ініціалізація*; *вираз*; *інкремент*) *настанова*;

Якщо цикл **for** призначений для багатократного виконання не однієї настанови, а програмного блоку, то його загальний формат має такий вигляд:

```
for(ініціалізація; вираз; інкремент) {
    послідовність настанов
}
```

Елемент *ініціалізація* зазвичай є настановою присвоєння, яка встановлює *керівній змінній циклу* початкове значення, що дорівнює нулю. Ця змінна діє як лічильник, який керує роботою циклу. Елемент *вираз* є умовним виразом, у якому тестується значення керівної змінної циклу. Результат цього тестування визначає, виконається цикл **for** ще раз чи ні. Елемент *інкремент* – вираз, який визначає, як змінюється значення керівної змінної циклу після кожної ітерації. Зверніть увагу на те, що всі ці елементи циклу **for** повинні відділятися крапкою з комою. Цикл **for** виконуватиметься доти, доки обчислення елемента *вираз* дає істинний результат. Як тільки цей умовний вираз стане помилковим, цикл завершиться, а виконання програми продовжиться з настанови, що є наступною за циклом **for**.

У наведеному нижче коді програми цикл **for** використовують для виведення значень квадратного кореня з чисел від 1 до 99. Зверніть увагу на те, що у наведеному прикладі керівна змінна називається n.





**Код програми 4.8. Демонстрація механізму використання настанови організації циклу for**

```
#include <iostream>          // Потокове введення-виведення
#include <cmath>             // Використання математичних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    double sq_root;
    clrscr();            // Очищення екрану

    for(int n=1; n < 100; n++) {
        sq_root = sqrt((double) n);
        cout << n << " " << sq_root << endl;
    }

    getch(); return 0;
}
```

Ось як виглядають перші рядки результатів розрахунку, що виводяться цією програмою:

```
1 1
2 1.41421
3 1.73205
4 2
5 2.23607
6 2.44949
7 2.64575
8 2.82843
9 3
10 3.16228
11 3.31662
```

У цьому коді програми використано ще одну стандартну функцію мови програмування С++: **sqrt**(). Ця функція повертає значення квадратного кореня зі свого аргументу. Аргумент повинен мати тип **double**, і саме тому під час виклику функції **sqrt**() параметр n приводиться до типу **double**. Сама функція також повертає значення типу **double**. Зверніть увагу на те, що у програму внесено заголовок <**cmath**>, оскільки цей заголовний файл забезпечує підтримку функції **sqrt**().

***Вартоа' нати!*** *Окрім функції* ***sqrt()****, мова програмування С++ підтримує широкий набір інших математичних функцій, наприклад* ***sin()****,* ***cos()****,* ***tan()****,* ***log()****,* ***ceil()*** *і* ***floor()****. Необхідно також пам'ятати, що всі математичні функції вимагають приєднання до програми заголовка* <***cmath***>.

Керівна змінна циклу **for** може змінюватися як з позитивним, так і з негативним приростом, причому величина цього приросту також може бути будь-якою. Наприклад, наведений нижче код програми виводить числа в діапазоні від 100 до -100 з декрементом, що дорівнює 5.

**Код програми 4.9. Демонстрація механізму використання настанови організації циклу for з декрементом**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    for(int i=100; i>=-100; i=-5) cout << i << " ";

    getch(); return 0;
}
```

Важливо розуміти, що умовний вираз завжди тестується на початку виконання циклу **for**. Це означає, що коли перша ж перевірка умови дасть значення ФАЛЬШ, програмний код тіла циклу не виконається жодного разу. Ось приклад:

```
for(pm=10; pm < 5; pm++)
cout << pm;     // Ця настанова не виконається.
```

Цей цикл ніколи не виконається, оскільки вже під час входу в нього значення його керівної змінної pm більше п'яти. Це робить умовний вираз (pm < 5) помилковим із самого початку. Тому навіть одна ітерація цього циклу не буде виконана.

### 4.3.1. Варіанти використання настанови організації циклу for

Настанова організації циклу **for** – одна з найбільш гнучких настанов у мові програмування C++, оскільки саме вона дає змогу отримати широкий діапазон варіантів її використання. Наприклад, для керування циклом **for** можна використовувати декілька змінних. Для розуміння сказаного розглянемо такий фрагмент коду програми:

```
for(x=0, y=10; x<=10; ++x, --y) cout << x << " " << y << endl;
```

У цьому записі комами відокремлюються дві настанови ініціалізації та два інкрементні вирази. Це робиться для того, щоби компілятор "розумів", що існує дві настанови ініціалізації та дві настанови інкремента (декремента). У мові програмування C++ кома є оператором, який, по суті, означає "зроби це і те". Інші застосування оператора "кома" ми розглянемо нижче у цьому посібнику, але найчастіше він використовується в циклі **for**. Під час входу у цей цикл ініціалізувалися обидві змінні – x і y. Після виконання кожної ітерації циклу змінна x інкрементується, а змінна y декрементується. Використання декількох керівних змінних у циклі іноді дає змогу спростити алгоритми. У розділах ініціалізації та інкремента циклу **for** можна використовувати будь-яку кількість настанов, але зазвичай їх кількість не перевищує двох.

Умовним виразом, який керує циклом **for**, може бути будь-який допустимий C++-вираз. При цьому він не обов'язково повинен містити керівну змінну циклу. У наведеному нижче прикладі цикл виконуватиметься доти, доки користувач не натисне на клавішу клавіатури. У цьому коді програми





представлена ще одна (дуже важлива) бібліотечна функція: **kbhit()**. Вона повертає значення ФАЛЬШ, якщо жодна клавіша не була натиснута на клавіатурі, і значення ІСТИНА – в іншому випадку. Функція чекає натиснення клавіші, даючи змогу тим самим циклу виконуватися доти, доки натискання не відбудеться. Функція **kbhit()** не визначається стандартом мови програмування C++, але включена в розширення мови програмування C++, яке підтримується більшістю компіляторів. Для її використання у програму необхідно внести заголовок <**conio**>.

**Код програми 4.10. Демонстрація механізму використання у циклі for функції, яка реагує на натискання клавіші**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()       // Виведення чисел на екран до натиснення будь-якої клавіші.
{
    clrscr();   // Очищення екрану
    for(int i=0; !kbhit(); i++) cout << i << " ";

    getch(); return 0;
}
```

На кожній ітерації циклу викликається функція **kbhit()**. Якщо після запуску програми натиснути на будь-яку клавішу, то ця функція поверне значення ІСТИНА, внаслідок чого вираз **!kbhit()** дасть значення ФАЛЬШ, і цикл зупиниться. Але, якщо не натискати на клавішу, то функція поверне значення ФАЛЬШ, а вираз **!kbhit()** дасть значення ІСТИНА, що дасть змогу циклу продовжувати "крутитися".

***Вартоа' нати!*** *Функція* ***kbhit()*** *не входить до складу стандартної бібліотеки мови програмування C++. Йдеться про те, що стандартна бібліотека визначає тільки мінімальний набір функцій, який повинні мати всі C++-компілятори. Функція* ***kbhit()*** *не включена в цей мінімальний набір, оскільки не всі середовища можуть підтримувати взаємодію з клавіатурою. Проте функцію* ***kbhit()*** *підтримують практично всі серійно випущені C++-компілятори. Виробники компіляторів можуть забезпечити підтримку більшої кількості функцій, ніж це необхідно для дотримання мінімальних вимог стосовно стандартної бібліотеки мови програмування C++. Додаткові ж функції дають змогу ширше використовувати можливості середовища програмування. Якщо для Вас не проблематичне питання переносності коду програми в інше середовище виконання, то Ви можете вільно використовувати всі функції, які підтримуються Вашим компілятором.*

### 4.3.2. Відсутність елементів у визначенні циклу

У мові програмування C++ дозволено опустити будь-який елемент заголовка циклу (ініціалізація, умовний вираз, інкремент) або навіть все відразу.

Наприклад, ми хочемо написати цикл, який повинен виконуватися доти, доки з клавіатури не буде введене число 123. Ось як виглядає така програма.

**Код програми 4.11. Демонстрація механізму реалізації відсутності елементів у визначенні циклу for**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      clrscr();              // Очищення екрану
      for(int x=0; x != 123) {
            cout << "Введіть число: "; cin >> x;
      }

      getch(); return 0;
}
```

У цьому коді програми в заголовку циклу **for** відсутній вираз інкремента. Це означає, що під час кожного повторення циклу здійснюється тільки одна дія: значення змінної x порівнюється з числом 123. Але, якщо ввести з клавіатури число 123, умовний вираз, який перевіряється в циклі, стане помилковим, і цикл завершиться. Оскільки вираз інкремента в заголовку циклу **for** відсутній, то керівна змінна циклу не модифікується.

Наведемо ще один варіант організації циклу **for**, в заголовку якого, як це показує такий фрагмент коду програми, відсутній розділ ініціалізації:

```
cout << "Введіть номер позиції: ";
cin >> x;
for(; x < limit; x++) cout << " ";
```

У цьому записі настанови організації циклу **for** є порожній розділ ініціалізації, а керована змінна x ініціалізується значенням, що вводиться користувачем, з клавіатури до входу в цикл.

До розміщення виразу ініціалізації за межами циклу, як правило, вдаються тільки у тому випадку, коли початкове значення генерується складним процесом, який незручно помістити у визначення циклу. Окрім цього, розділ ініціалізації залишають порожнім і у разі, коли керування циклом здійснюється за допомогою параметра певної функції, а як початкове значення керівної змінної циклу використовується значення, яке отримує параметр під час виклику функції.

### 4.3.3. Механізм реалізації нескінченного циклу

Нескінченний цикл – цикл, який ніколи не закінчується. Залишивши порожнім умовний вираз циклу **for**, можна створити нескінченний цикл (цикл, який ніколи не закінчується). Спосіб запису такого циклу показаний на прикладі такої конструкції циклу **for**:

```
for(;;) { //... }
```





Цей цикл працюватиме без кінця. Незважаючи на наявність деяких задач програмування (наприклад, командних процесорів операційних систем), які вимагають наявність нескінченного циклу, більшість "нескінченних циклів" – просто цикли із спеціальними вимогами до завершення. Ближче до кінця цього розділу буде показано, як завершити цикл такого типу. Зрозуміло, що це можна організувати за допомогою настанови **break**.

### 4.3.4. Цикли часової затримки роботи програми

У програмах часто використовують так звані цикли часової затримки. Їх завдання – просто затримати час роботи програми. Для запису таких циклів достатньо залишити порожнім тіло циклу, тобто опустити ті настанови, які повторює цикл на кожній ітерації. Ось приклад:

```
for(int x=0; x<1000; x++);
```

Цей цикл тільки інкрементує значення змінної x і не робить нічого більше. Крапка з комою (в кінці рядка) необхідна внаслідок того, що цикл **for** чекає отримати настанову, яка може бути порожньою (як у цьому випадку).

Перш ніж рухатися далі, не завадило б проекспериментувати з власними варіаціями на тему циклу **for**. Це Вам допоможе переконатися в його гнучкості та потужності.

## 4.4. Механізм використання інших ітераційних настанов

У більшості задач, що трапляються на практиці програмування, необхідно організувати багатократне виконання деякої дії. Така ділянка обчислювального процесу, що багато разів повторюється, називається циклом. Якщо заздалегідь відома кількість необхідних повторень, то цикл називається арифметичним. Якщо ж кількість повторень заздалегідь невідома, то говорять про ітераційний цикл.

У ітераційних циклах виробляється перевірка деякої умови і, залежно від результату цієї перевірки, відбувається або вихід з циклу, або повторення виконання тіла циклу. Якщо перевірка умови виробляється перед виконанням блоку операторів, то такий ітераційний цикл називається циклом з передумовою (цикл "доки"), а якщо перевірка виробляється після виконання тіла циклу, то це цикл з після умовою (цикл "до").

Особливість цих циклів полягає в тому, що тіло циклу з після умовою завжди виконується хоч би один раз, а тіло циклу з передумовою може жодного разу не виконатися. Залежно від вирішуваного завдання необхідно використовувати той або інший вигляд ітераційних циклів.

### 4.4.1. Ітераційна настанова while

Загальна форма організації циклу **while** має такий вигляд:

**while**(*вираз*) *настанова*;

У цьому записі під елементом *настанова* розуміють або одиночну настанову, або блок настанов. Роботою циклу керує елемент *вираз*, який є будь-яким допустимим С++-виразом. Елемент *настанова* здійснюється доти, доки умовний вираз повертає значення ІСТИНА. Як тільки цей вираз стає помилковим, то керування передається настанові, яка знаходиться за цим циклом.

*Настанова* ***while*** *– один із способів організації ітераційних циклів у мові програмування С++.*

Використання циклу **while** можна продемонструвати на прикладі такої невеликої програми. Практично всі компілятори підтримують розширений набір символів, який не обмежується символами ASCII. У розширеному наборі часто містяться спеціальні символи і деякі букви з алфавітів іноземних мов. ASCII-символи використовують значення, що не перевищують число 127, а розширений набір символів – значення з діапазону 128-255. У процесі виконання цієї програми виводяться всі символи, значення яких лежать в діапазоні 32-255 (32 – код пропуску). Виконавши цю програму, програміст повинен побачити ряд дуже цікавих символів.

**Код програми 4.12. Демонстрація механізму виведення усіх друкованих символів, в т.ч. розширений набір символів, якщо такі існують**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      unsigned char ch;
      clrscr();              // Очищення екрану
      ch = 32;
      while(ch) {
            cout << ch; ch++;
      }
      getch(); return 0;
}
```

Розглянемо **while**-вираз з попередньої програми. Можливо, Вас здивувало, що він складається всього тільки з однієї змінної ch. Але "скринька" тут відкривається просто. Оскільки змінна ch має тут тип **unsigned char**, то вона може містити значення від 0 до 255. Якщо її значення дорівнює 255, то після інкрементування воно "скидається" в нуль. Отже, факт рівності значення змінної ch нулю слугує зручним способом завершити **while**-цикл.

Подібно до циклу **for**, умовний вираз перевіряється під час входу в цикл **while**, а це означає, що тіло циклу (при помилковому результаті обчислення умовного виразу) може не виконатися жодного разу. Ця властивість циклу усуває необхідність окремого тестування до початку виконання циклу. Наступна програма виводить рядок, що складається з крапок. Кількість виведених крапок дорівнює значенню, яке вводить користувач. Програма не дає змоги виводити рядки, якщо їх довжина перевищує 80 символів. Перевірка на





допустимість кількості крапок, що виводяться, здійснюється усередині умовного виразу циклу, а не зовні.

**Код програми 4.13. Демонстрація механізму виведення рядка, який складається з крапок**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      int len;
      clrscr();              // Очищення екрану

      cout << "Введіть довжину рядка (від 1 до 79): "; cin >> len;
      while(len>0 && len<80) {
            cout << '.'; len--;
      }
      getch(); return 0;
}
```

Тіло **while**-циклу може взагалі не містити жодної настанови, наприклад:

```
while(rand()!= 100);
```

Цей цикл здійснюється доти, доки випадкове число, що генерується функцією **rand**(), не виявиться таким, що дорівнює числу 100.

### 4.4.2. Ітераційна настанова do-while

На відміну від циклів **for** і **while**, у яких умова перевіряється під час входу, цикл **do-while** перевіряє умову при виході з циклу. Це означає, що цикл **do-while** завжди здійснюється хоч би один раз. Його загальний формат має такий вигляд:

```
do {
      настанови;
} while(вираз);
```

Хоча фігурні дужки є необов'язковими, якщо елемент настанови складається тільки з однієї настанови, то вони часто використовують для поліпшення читабельності конструкції **do-while**, не допускаючи тим самим плутанини з циклом **while**. Цикл **do-while** здійснюється доти, доки залишається істинним елемент вираз, який є умовним виразом.

*Ітераційна настанова* ***do-while*** *– єдиний цикл, який завжди робить ітерацію хоч би один раз.*

У наведеному нижче коді програми цикл **do-while** здійснюється доти, доки користувач не введе число 100.

**Код програми 4.14. Демонстрація механізму використання настанови організації циклу do-while**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int n;
    do {
        cout << "Введіть число (100 – для виходу): "; cin >> n;
    } while(n != 100);

    getch(); return 0;
}
```

Використовуючи цикл **do-while**, ми можемо ще більше удосконалити програму "Вгадай магічне число". Цього разу програма "не випустить" Вас з циклу вгадування, доки Ви не вгадаєте це число.

**Код програми 4.15. Демонстрація механізму роботи програми "Вгадай магічне число": третє удосконалення**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int magic;          // Магічне число
    int guess;          // Варіант користувача
    clrscr();           // Очищення екрану

    magic = rand();     // Отримуємо випадкове число.
    do {
        cout << "Введіть свій варіант магічного числа: "; cin >> guess;
        if(guess == magic) {
            cout << "** Правильно ** ";
            cout << magic << " і є те саме магічне число" << endl;
        }
        else {
            cout << "Дуже шкода, але Ви помилилися.";
            if(guess > magic)
                cout << "Ваш варіант перевищує магічне число" << endl;
            else
                cout << "Ваш варіант менше магічного числа" << endl;
        }
    } while(guess != magic);

    getch(); return 0;
}
```





### 4.4.3. Механізм використання настанови переходу continue

У мові програмування С++ існує засіб "дострокового" виходу з поточної ітерації циклу. Цим засобом є настанова **continue**. Вона примусово здійснює перехід до наступної ітерації, опускаючи виконання коду програми, що залишився, в поточній. Наприклад, у наведеному нижче коді програми настанову **continue** використовують для "прискореного" пошуку парних чисел в діапазоні від 0 до 100.

**Код програми 4.16. Демонстрація механізму використання настанови переходу continue**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    for(int x=0; x<=100; x++) {
        if(x%2) continue;
        cout << x << " ";
    }

    getch(); return 0;
}
```

У цьому коді програми виводяться тільки парні числа, оскільки внаслідок виявлення непарного числа відбувається передчасний перехід до наступної ітерації, і **cout**-настанова опускається.

---

*Настанова **continue** дає змогу негайно перейти до виконання наступної ітерації циклу.*

У циклах **while** і **do-while** настанова **continue** передає керування безпосередньо настанові, що перевіряє умовний вираз, після чого циклічний процес триває. А в циклі **for** після виконання настанови **continue** спочатку обчислюється інкрементний вираз, а потім – умовний. І тільки після цього циклічний процес буде продовжено.

### 4.4.4. Механізм використання настанови break для виходу з циклу

За допомогою настанови **break** можна організувати негайний вихід з циклу, знехтувавши виконанням коду програми, що залишився в його тілі, і перевірку умовного виразу. Завдяки виявленню усередині циклу настанови **break** цикл завершується, а керування передається настанові, що є наступною після циклу. Розглянемо простий приклад.

**Код програми 4.17. Демонстрація механізму використання настанови break для виходу з циклу**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен
```

```
int main()
{
    // Цикл працює для значень t від 0 до 9, а не до 100!
    for(int t=0; t<100; t++) {
        if(t==10) break;
        cout << t << " ";
    }
    getch(); return 0;
}
```

Ця програма виведе на екран числа від 0 до 9, а не до 100, оскільки настанова **break** при значенні параметра циклу t, що дорівнює 10, забезпечує негайний вихід з циклу.

*Настанова **break** дає змогу негайно вийти з циклу.*

Настанова **break** зазвичай використовується в циклах, у яких при створенні особливих умов необхідно забезпечити негайне їх завершення. Такий фрагмент містить приклад ситуації, коли після натиснення клавіші виконання циклу зупиняється:

```
for(int i=0; i<1000; i++) {
    // Виконання якихось дій.
    if(kbhit()) break;
}
```

Настанова **break** призводить до виходу з самого внутрішнього циклу. Розглянемо приклад.

**Код програми 4.18. Демонстрація механізму використання настанови break для виходу з внутрішнього циклу**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    clrscr();            // Очищення екрану
    for(int t=0; t<100; t++) {
        int pm = 1;
        for(;;) {
            cout << pm << " ";
            pm++;
            if(pm==10) break;
        }
        cout << endl;
    }
    getch(); return 0;
}
```

Ця програма 100 разів виводить на екран числа від 0 до 9. Під час кожного виконання настанови **break** керування передається назад в зовнішній цикл **for**.





***Вартоа' нати!*** *Настанова **break**, яка завершує виконання настанови **switch**, впливає тільки на настанову **switch**, а не на цикл, що містить її.*

На прикладі попередньої програми Ви переконалися в тому, що у мові програмування С++ за допомогою настанови **for** можна організувати нескінченний цикл[1]. Для виходу з нескінченного циклу необхідно використовувати настанову **break**. Безумовно, настанову **break** можна використовувати і для завершення нескінченного циклу.

### 4.4.5. Організація вкладених циклів

Як було продемонстровано на прикладі попередньої програми, один цикл можна вкласти в іншій. У мові програмування С++ дозволено використовувати до 256 рівнів вкладення. Вкладені цикли використовуються для вирішення завдань найрізноманітнішого профілю. Наприклад, у наведеному нижче коді програми настанова організації вкладеного циклу **for** дає змогу знайти прості числа в діапазоні від 2 до 1000.

**Код програми 4.19. Демонстрація механізму виведення простих чисел у діапазоні від 2 до 1000**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int j;
    clrscr();                // Очищення екрану

    for(int i=2; i<1000; i++) {
        for(j=2; j<=(i/j); j++)
            if(!(i%j)) break; // Якщо число має множник, значить воно не просте.
        if(j>(i/j)) cout << i << " – просте число" << endl;
    }

    getch(); return 0;
}
```

Ця програма визначає, чи є простим число, яке міститься в змінній i, шляхом послідовного його ділення на значення, розташоване між числом 2 і результатом обчислення виразу i/j[2]. Якщо залишок від ділення i/j дорівнює нулю, то це означає, що число i не є простим. Але, якщо внутрішній цикл завершиться повністю (без дострокового завершення роботи з використанням настанови **break**), то це означає, що поточне значення змінної i дійсно є простим числом.

[1] Нескінченні цикли можна також створювати, використовуючи настанови **while** або **do-while**, але цикл **for** – традиційне вирішення переважної більшості питань.

[2] Зупинити перебір множників можна на значенні виразу i/j, оскільки число, яке перевищує i/j, вже не може бути множником значення i.

## 4.5. Механізм використання настанови goto

Довгі роки настанова безумовного переходу **goto** була непопулярною у програмістів, оскільки сприяла, із їхньої точки зору, створенню "спагеттікоду програми". Проте настанова **goto**, як і раніше, використовується, а іноді й навіть дуже ефективно. У цьому навчальному посібнику не робиться спроба "реабілітації" цієї настанови як одної з форм керування роботою програмою. Понад це, у будь-якій ситуації (у області програмування) можна обійтися без настанови **goto**, оскільки вона не є елементом, що забезпечує повноту опису мови програмування. Водночас, у певних ситуаціях її використання може бути дуже корисним. У цьому посібнику ми вирішили обмежити використання настанови **goto** межами цього розділу, оскільки, на думку більшості програмістів, вона вносить у програму тільки безлад і робить її практично нечитабельною. Але, оскільки використання настанови **goto** в деяких випадках може зробити намір програміста зрозумілішим, їй варто приділити певну увагу.

### 4.5.1. Настанова goto – настанова безумовного переходу

Настанова **goto** вимагає наявності у програмі мітки. *Мітка* – дійсний у мові програмування C++ ідентифікатор, за яким поставлено двокрапку. У процесі виконання настанови **goto** керування програмою передається настанові, вказаній за допомогою мітки. Мітка повинна знаходитися в одній функції з настановою **goto**, яка посилається на цю мітку.

***Мітка** – ідентифікатор, за яким знаходиться двокрапка.*

Наприклад, за допомогою настанови **goto** і мітки можна організувати такий цикл на 100 ітерацій:

```
x = 1;
loop1:
x++;
if(x < 100) goto loopl;
```

Іноді настанову **goto** варто використовувати для виходу з глибоко вкладених настанов циклу. Для розуміння сказаного розглянемо такий фрагмент коду програми:

```
for(...) {
    for(...) {
        while(...) {
            if(...) goto stop;
            ...
        }
    }
}

stop:
cout << "Помилка у програмі" << endl;
```





Щоб замінити настанову **goto**, довелося б виконати ряд додаткових перевірок. У цьому випадку настанова **goto** істотно спрощує програмний код. Простим застосуванням настанови **break** тут не обійшлося, оскільки вона забезпечила б вихід тільки з самого внутрішнього циклу.

***Вартоа' нати!** Настанову **goto** необхідно застосовувати обмежено (як сильнодіючі ліки). Якщо без неї Ваш код програми буде менш читабельним або для Вас важлива швидкість виконання програми, то в таких випадках використання настанови **goto** допустиме.*

### 4.5.2. Приклад використання настанов керування ходом виконання програм

Наведений нижче приклад є останньою версією програми "Вгадай магічне число". У ній використано багато засобів C++-програмування, представлених у цьому розділі. Пропонуємо, перш ніж переходити до наступного розділу, переконатися у тому, що добре розумієте всі розглянуті тут елементи мови програмування C++. Цей варіант програми дає змогу згенерувати нове число, зіграти в гру і вийти з програми.

**Код програми 4.20. Демонстрація механізму роботи програми "Вгадай магічне число": остання версія**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void play(int m);            // Попереднє оголошення функції

int main()
{
    int magic;          // Магічне число
    int guess;          // Варіант користувача
    magic = rand();
    do {
        cout << "1. Отримати нове магічне число" << endl;
        cout << "2. Зіграти" << endl;
        cout << "3. Вийти з програми" << endl;
        do {
            cout << "Введіть свій варіант: "; cin >> option;
        } while(option<1 || option>3);
        switch(option) {
            case 1: magic = rand();
                break;
            case 2: play(magic);
                break;
            case 3: cout << "До побачення!" << endl;
                break;
        }
    } while(option != 3);
```

```
        getch(); return 0;
}

// Зіграємо в гру.
void play(int m)
{
        int t, x;

        for(t=0; t<100; t++) {
                cout << "Вгадайте магічне число: "; cin >> x;
                if(x==m) {
                        cout << "** Правильно **" << endl;
                        return;
                }
                else
                        if(x<m) cout << "Замало" << endl;
                        else cout << "Забагато" << endl;
        }
        cout << "Ви використали всі шанси вгадати магічне число. "
                << "Спробуйте знову" << endl;
}
```





# Розділ 5. МАСИВИ ТА РЯДКИ – ЗАСОБИ ГРУПУВАННЯ ВЗАЄМОПОВ'ЯЗАНИХ МІЖ СОБОЮ ЗМІННИХ

***Масив*** (array) – перелік змінних однакового типу, звернення до яких відбувається із застосуванням імені, загального для всіх його елементів. У мові програмування С++ масиви можуть бути одно-, дво- та багатовимірними, хоча в основному використовуються одновимірні масиви. Масиви є зручними засобами для групування взаємопов'язаних між собою змінних.

***Рядок*** (**string**) – тип даних, значеннями якого є довільна послідовність символів алфавіту. Кожна змінна такого типу може бути представлена фіксованою кількістю байтів або мати довільну довжину. Деякі мови програмування накладають обмеження на максимальну довжину рядка, але в більшості мов подібні обмеження відсутні.

Основні проблеми в машинному представленні рядкового типу: рядки можуть мати достатньо істотний розмір (до декількох десятків мегабайтів); з часом розмір може змінюватися, тобто виникають труднощі з додаванням і видаленням символів.

Поряд з числовими найчастіше використовуються символьні масиви, у яких зберігаються рядки. Як було зазначено вище, у мові програмування С++ не визначено вбудованого типу даних для зберігання рядків. Тому рядки реалізуються як масиви символів. Такий підхід до реалізації рядків дає С++-програмісту більше "важелів" керування порівняно з тими мовами, у яких використовується окремий рядковий тип даних.

## 5.1. Одновимірні масиви

*Одновимірний масив* – перелік взаємопов'язаних між собою змінних. Для оголошення одновимірного масиву використовують така форма запису:

```
тип ім'я_масиву[розмір];
```

У цьому записі за допомогою елемента запису *тип* оголошується базовий тип масиву. *Базовий тип* визначає тип даних кожного елемента, з яких складається масив. Кількість елементів, які зберігатимуться в масиві, визначається елементом *розмір*. Наприклад, у процесі виконання наведеної нижче настанови оголошується **int**-масив (що складається з 10 елементів) з іменем Array:

```
int Array[10];
```

Індекс у прямокутних дужках після імені масиву вказує на конкретний елемент масиву.

Доступ до окремого елемента масиву здійснюється за допомогою індексу, який описує позицію елемента усередині масиву. У мові програмуван-

ня C++ перший елемент масиву має нульовий індекс. Оскільки масив Array містить 10 елементів, то його індекси змінюються від 0 до 9. Щоб отримати доступ до елемента масиву за індексом, достатньо вказати потрібний номер елемента в квадратних дужках. Так, наприклад, першим елементом масиву Array є Array[0], а останнім – Array[9]. Розглянемо наведену нижче програму, яка ініціалізує масив Array квадратами чисел від 1 до 10.

**Код програми 5.1. Демонстрація механізму ініціалізації елементів одновимірного масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int Array[10]; // Ця настанова резервує область пам'яті для 10 елементів типу int.
    clrscr();            // Очищення екрану

    // Поміщаємо в масив значення.
    for(int t=0; t<10; ++t) Array[t] = (t+1)*(t+1);

    // Відображається масив.
    for(int t=0; t<10; ++t) cout << Array[t] << " ";

    getch(); return 0;
}
```

У мові програмування C++ всі масиви займають суміжні елементи пам'яті. Іншими словами, елементи масиву в пам'яті розташовані послідовно один за одним. Клітина з найменшою адресою належить до першого елемента масиву, а з найбільшою – до останнього. Наприклад, після виконання цього фрагмента коду програми

```
int Array[7];
for(int j=0; j<7; j++) Array[j] = j*j;
```

масив Array виглядатиме так:

| i[0] | i[1] | i[2] | i[3] | i[4] | i[5] | i[6] |
|---|---|---|---|---|---|---|
| 0 | 1 | 4 | 9 | 16 | 25 | 36 |

Для одновимірних масивів загальний розмір масиву в байтах обчислюється так:

*всього байтів = розмір типу елемента в байтах × кількість елементів.*

Масиви часто використовують під час програмування, оскільки дають змогу легко обробляти велику кількість взаємопов'язаних між собою змінних. Наприклад, у наведеному нижче коді програми створюється масив з десяти елементів, кожному елементу присвоюється випадкове число, а потім на екрані відображаються мінімальне і максимальне його значення.





**Код програми 5.1. Демонстрація механізму оброблення елементів одновимірного масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int min, max, Array[10];
    clrscr();                // Очищення екрану
    for(int i=0; i<10; i++) Array[i] = rand();

    // Знаходимо мінімальне значення.
    min = Array[0];
    for(int i=1; i<10; i++)
        if(min > Array[i]) min = Array[i];
    cout << "Мінімальне значення: " << min << endl;

    // Знаходимо максимальне значення.
    max = Array[0];
    for(int i=1; i<10; i++)
        if(max < Array[i]) max = Array[i];
    cout << "Максимальне значення: " << max << endl;

    getch(); return 0;
}
```

У мові програмування C++ не можна присвоїти один масив іншому. У наведеному нижче коді програми, наприклад, присвоєння aMas = bMas; неприпустимо:

```
int aMas[10], bMas[10];
//...
aMas = bMas; // Помилка!!!
```

Щоб помістити вміст одного масиву в інший, необхідно окремо виконати присвоєння кожного значення:

```
int aMas[10], bMas[10], i;
//...
for(i=1; i<10; i++) aMas[i] = bMas[i]; // Правильно!
//...
```

### 5.1.1. Організація контролю меж масивів

У мові програмування C++ не здійснюється ніякої перевірки порушення контролю меж масивів, тобто нічого не може перешкодити програмісту звернутися до масиву за його межами. Якщо це відбувається у процесі виконання настанови присвоєння, то можуть бути змінені значення в елементах пам'яті, виділених деяким іншим змінним або навіть Вашій програмі. Іншими словами, звернення до масиву (розміром у N елементів) за межею N-го елемента може призвести до руйнування програми за відсутності яких-небудь заува-

жень з боку компілятора і без видачі повідомлень про помилки під час роботи програми. Це означає, що вся відповідальність за дотримання "кордонів" масивів покладається тільки на програмістів, які повинні гарантувати коректну роботу з масивами. Іншими словами, програміст зобов'язаний використовувати масиви достатньо великого розміру, щоб в них можна було без ускладнень поміщати дані. Однак найкраще у програмі передбачити перевірку перетину "кордонів" масивів.

Наприклад, C++-компілятор "мовчки" скомпілює і дасть змогу запустити таку програму на виконання, хоча у ній відбувається вихід за межі масиву myArray.

***О! ережно!*** *Не виконуйте наведений нижче приклад програми. Це може призвести до руйнування Вашої системи.*

**Код програми 5.2. Демонстрація запису некоректного прикладу програми**

```
int main()
{
    int myArray[10];

    for(int i=0; i<100; i++) myArray[i] = i;

    return 1;
}
```

У цьому випадку цикл **for** виконає 100 ітерацій, хоча масив myArray призначений для зберігання тільки десяти елементів. У процесі виконання цієї програми можливий перезапис важливої інформації, що може призвести до аварійної зупинки програми.

Вас, можливо, дивує така "непередбачливість" мови програмування C++, яка виражається у відсутності вбудованих засобів динамічної перевірки на "недоторканність" меж масивів. Нагадаємо, проте, що мова програмування C++ призначена для професійних програмістів, і її завдання – надати їм можливість створювати максимально ефективний програмний код. Будь-яка перевірка коректності доступу засобами мови програмування C++ істотно уповільнює виконання програми. Тому такі дії залишено на розгляд програмістам. Як буде показано далі, у разі потреби програміст може сам визначити тип масиву і закласти у ньому перевірку непорушності меж.

### 5.1.2. Сортування елементів масиву

Однією з найпоширеніших операцій, що виконуються над масивами, є сортування. Існує багато різних алгоритмів сортування. Широко застосовується, наприклад, сортування перемішуванням і сортування методом Шелла. Відомий також алгоритм Quicksort (швидке сортування з розбиттям початкового набору даних на дві половини так, що будь-який елемент першої половини впорядкований щодо будь-якого елемента другої половини). Проте найпростішим вважають алгоритм сортування методом бульбашок. Незважаючи на те, що бульбашкове сортування не відрізняється високою ефективністю (і справді, його продуктивність неприйнятна для сортування великих масивів), його цілком успішно можна застосовувати для сортування масивів малого розміру.





Алгоритм сортування методом бульбашок отримав свою назву від способу, використовуваного для впорядковування елементів масиву. Тут виконуються операції порівняння, що повторюються, і у разі потреби міняються місцями суміжні елементи. При цьому елементи з меншими значеннями поступово переміщаються до одного кінця масиву, а елементи з великими значеннями – до іншого. Цей процес нагадує поведінку бульбашок повітря в резервуарі з водою. Бульбашкове сортування здійснюється шляхом декількох проходів по масиву, під час яких у разі потреби здійснюється перестановка елементів, що опинилися "не на своєму місці". Кількість проходів, що гарантують отримання відсортованих елементів масиву, дорівнює кількості елементів у масиві, зменшеному на одиницю.

У наведеному нижче коді програми реалізовано алгоритм сортування елементів масиву (цілого типу), що містить випадкові числа. Ця програма заслуговує уважного розбору.

**Код програми 5.3. Демонстрація механізму використання методу бульбашок для сортування елементів масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int Array[10];
    int a, b, t, size = 10;      // Кількість елементів, що підлягають сортуванню.

    // Поміщаємо в масив випадкові числа.
    for(t=0; t<size; t++) Array[t] = rand();

    // Відображаємо початковий масив.
    cout << "Початковий масив: ";
    for(t=0; t<size; t++) cout << Array[t] << " ";
    cout << endl;

    // Реалізація методу бульбашкового сортування.
    for(a=1; a<size; a++)
        for(b=size-1; b>=a; b--) {
            if(Array[b-1]> Array[b]) { // Елементи неврегульовані.
                // Міняємо елементи місцями.
                t = Array[b-1]; Array[b-1] = Array[b]; Array[b] = t;
            }
        } // Кінець бульбашкового сортування.

    // Відображаємо відсортований масив.
    cout << "Відсортований масив: ";
```

```
    for(t=0; t<size; t++) cout << Array[t] << " ";
    cout << endl;

    getch(); return 0;
}
```

Хоча алгоритм бульбашкового сортування придатний для невеликих масивів, для масивів великого розміру він стає неефективним. Більш універсальним вважають алгоритм Quicksort. У стандартну бібліотеку мови програмування C++ включена функція **qsort**(), яка реалізує одну з версій цього алгоритму. Але, перш ніж використовувати її, Вам необхідно вивчити більше засобів мови програмування C++.

## 5.2. Побудова символьних рядків

Найчастіше одновимірні масиви використовуються для побудови символьних рядків. У мові програмування C++ рядок визначається як символьний масив, який завершується нульовим символом ('\0'). Під час визначення довжини символьного масиву необхідно враховувати ознаку його завершення, тобто задавати його довжину на одиницю більше довжини найбільшого рядка, які передбачають зберігати у цьому масиві.

***Рядок*** – символьний масив, який завершується нульовим символом.

### 5.2.1. Оголошення рядкового літерала

Оголошуючи масив str, призначеного для зберігання 10-символьного рядка, потрібно використовувати таку настанову:

```
char str[11];
```

Заданий тут розмір (11) дає змогу зарезервувати місце для нульового символу в кінці рядка.

Як було зазначено вище, мова програмування C++ дає змогу визначати рядкові літерали. Пригадаємо, що *рядковий літерал* – перелік символів, поміщений в подвійні лапки. Ось декілька прикладів:

```
"Привіт"
"Мені подобається мова програмування C++"
"#$%@ @#$"
""
```

Рядок, наведений останнім (""), називається *нульовим*. Він складається тільки з одного нульового символу (ознаки завершення рядка). Нульові рядки використовуються для представлення порожніх рядків.

Програмісту не потрібно вручну добавляти в кінець рядкових констант нульові символи. C++-компілятор робить це автоматично. Отже, рядок "Привіт" в пам'яті розміщується так, як це показано на цьому рисунку:

| П | Р | И | В | І | Т | '\0' |
|---|---|---|---|---|---|---|





### 5.2.2. Зчитування рядків з клавіатури

Найпростіше зчитувати рядок з клавіатури, створивши масив, який прийме цей рядок за допомогою настанови **cin**. Зчитування рядка, введеного користувачем з клавіатури, відображено у наведеному нижче коді програми.

**Код програми 5.4. Демонстрація механізму використання cin-настанови для зчитування рядка з клавіатури**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[80];

    cout << "Введіть рядок: "; cin >> str;    // Зчитуємо рядок з клавіатури.
    cout << "Ось Ваш рядок: ";
    cout << str;

    getch(); return 0;
}
```

Хоча ця програма формально коректна, однак вона не позбавлена недоліків. Розглянемо такий результат її виконання:

```
Введіть рядок: Це перевірка
Ось Ваш рядок: Це
```

Як бачимо, під час виведення рядка, введеного з клавіатури, програма відображає тільки слово "Це", а не весь рядок. Йдеться про те, що оператор "<<" припиняє зчитування рядка, як тільки трапляється символ пропуску, табуляції або нового рядка (називатимемо ці символи пропускними). Для вирішення цього питання можна використовувати ще одну бібліотечну функцію **gets**(). Загальний формат її виклику є таким:

**gets**(*ім'я_масиву*);

Якщо у програмі необхідно зчитувати рядок з клавіатури, то викличте функцію **gets**(), а як аргумент передайте ім'я масиву, не вказуючи індексу. Після виконання цієї функції заданий масив міститиме текст, введений з клавіатури. Функція **gets**() зчитує символи, які вводяться користувачем, доти, доки він не натисне на клавішу <Enter>. Для виклику функції **gets**() у програму необхідно включити заголовок <**cstdio**>.

У наведеній нижче версії попередньої програми продемонстровано механізм використання функції **gets**(), яка дає змогу ввести в масив рядок символів, що містить пропуски.

**Код програми 5.5. Демонстрація механізму використання функції gets() для зчитування рядка з клавіатури**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <conio>             // Консольний режим роботи
```

```
using namespace std;          // Використання стандартного простору імен

int main()
{
    char str[80];

    cout << "Введіть рядок: "; gets(str); // Зчитуємо рядок з клавіатури.
    cout << "Ось Ваш рядок: ";
    cout << str;

    getch(); return 0;
}
```

Цього разу після запуску нової версії програми на виконання і введення з клавіатури тексту "Це простий тест" рядок зчитується повністю, а потім так само повністю і відображається:

```
Введіть рядок: Це простий тест
Ось Ваш рядок: Це простий тест
```

У цьому коді програми зверніть увагу на таку настанову:

```
cout << str;
```

У цьому записі (замість звичного літерала) використовують ім'я рядкового масиву. І хоча причина такого використання настанови **cout** нам стане зрозумілою після вивчення ще декількох розділів цього навчального посібника, поки що стисло зазначимо, що ім'я символьного масиву, який містить рядок, можна використовувати скрізь, де допустимо застосування рядкового літерала. При цьому майте на увазі, що ні оператор "<<", ні функція **gets**() не виконують граничної перевірки (на відсутність порушення меж масиву). Тому, якщо користувач введе рядок, довжина якого перевищує розмір масиву, то можливі неприємності, про які згадувалося вище. Із сказаного виходить, що обидва описаних тут варіанти зчитування рядків з клавіатури є потенційно небезпечними.

## 5.3. Застосування бібліотечних функцій для оброблення рядків

Мова програмування C++ підтримує багато функцій для оброблення рядків. Найпоширенішими з них є такі: **strepy()**, **strcpy()**, **strlen()**, **strcmp()**.

Для виклику всіх цих функцій у програму необхідно включити заголовок <**cstring**>. Тепер познайомимося з кожною функцією окремо.

### 5.3.1. Механізм використання функції strcpy()

Загальний формат виклику функції **strcpy**() є таким:

**strcpy**(*to*, *from*);

Функція **strcpy**() копіює вміст рядка *from* в рядок *to*. Необхідно пам'ятати, масив, який використовують для зберігання рядка *to*, повинен бути достатньо





великим, щоб в нього можна було помістити рядок з масиву from. Інакше масив to переповниться, тобто відбудеться вихід за його межі, що може призвести до руйнування програми.

Використання функції **strcpy**() продемонстровано у наведеному нижче коді програми, яка копіює рядок "Привіт" в рядок str:

**Код програми 5.6. Демонстрація механізму використання функції strcpy()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[80];

    strcpy(str, "Привіт");
    cout << str << endl << "";

    getch(); return 0;
}
```

### 5.3.2. Механізм використання функції strcat()

Звернення до функції **strcat**() має такий формат:

**strcat**(*s1*, *s2*);

Функція **strcat**() приєднує рядок *s2* до кінця рядка *s1*; при цьому рядок *s2* не змінюється. Обидва рядки повинні завершуватися нульовим символом. Результат виклику цієї функції, тобто остаточний рядок *s1* також завершуватиметься нульовим символом. Використання функції **strcat**() продемонстровано у наведеному нижче коді програми, яка має вивести на екран рядок "Привіт усім!".

**Код програми 5.7. Демонстрація механізму використання функції strcat()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char s1[20], s2[10];
    strcat(s1, "Привіт");
    strcat(s2, " усім!");
    strcat(s1, s2);
    cout << s1;

    getch(); return 0;
}
```

### 5.3.3. Механізм використання функції strcmp()

Звернення до функції **strcmp**() має такий формат:

**strcmp**(*s1*, s2);

Функція **strcmp**() порівнює рядок *s2* з рядком *s1* і повертає значення 0, якщо вони однакові. Якщо рядок *s1* лексикографічно (тобто відповідно до алфавітного порядку) більший від рядка *s2*, то повертається позитивне число. Якщо рядок *s1* лексикографічно менший від рядка *s2*, то повертається негативне число. Використання функції **strcpy**() продемонстровано у наведеному нижче коді програми, яка слугує для перевірки правильності пароля, введеного користувачем (для введення пароля з клавіатури і його верифікації слугує функція **password**()).

**Код програми 5.8. Демонстрація механізму використання функції strcmp()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <cstdio>            // Підтримка системи введення-виведення
using namespace std;         // Використання стандартного простору імен

bool password();

int main()
{
      if(password()) cout << "Вхід дозволено" << endl;
      else cout << "У доступі відмовлено" << endl;

      getch(); return 0;
}

// Функція повертає значення true, якщо пароль прийнятий, і значення false інакше.
bool password()
{
      char str[80];

      cout << "Введіть пароль: "; gets(str);
      if(strcmp(str, "пароль")) { // Рядки різні.
            cout << "Пароль недійсний" << endl;
            return false;
      }
      // Порівнювані рядки збігаються
      return true;
}
```

Під час застосування функції **strcmp**() важливо пам'ятати, що вона повертає число Про (тобто значення **false**), якщо порівнювані рядки однакові. Отже, якщо Вам необхідно виконати певні дії за умови збігу рядків, то програміст повинен використовувати оператор НЕ (!). Наприклад, у процесі виконання наведеної нижче програми запит вхідних даних продовжується доти, доки користувач не введе слово "Вихід".





**Код програми 5.9. Демонстрація механізму припинення введення вхідних даних за умови збігу рядків**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      char str[80];
      for(;;) {
            cout << "Введіть рядок: "; gets(str);
            if(!strcmp("Вихід", str)) break;
      }
      getch(); return 0;
}
```

### 5.3.4. Механізм використання функції strlen()

Загальний формат виклику функції **strlen**() є таким:

**strlen**(*s*);

У цьому записі *s* – рядок. Функція **strlen**() повертає довжину рядка, вказаного аргументом *s*. У процесі виконання наведеної нижче програми буде показано довжину рядка, введеного з клавіатури.

**Код програми 5.10. Демонстрація механізму використання функції strlen()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

int main()
{
      char str[80];

      cout << "Введіть рядок: "; gets(str);
      cout << "Довжина рядка дорівнює: " << strlen(str);

      getch(); return 0;
}
```

Якщо користувач введе рядок "Привіт усім!", програма виведе на екрані число 12. При підрахунку символів, з яких складається заданий рядок, ознака завершення рядка (нульовий символ) не враховується.

А у процесі виконання цієї програми рядок, введений з клавіатури, буде відображено на екрані в зворотному порядку. Наприклад, під час введення слова "привіт" програма відобразить слово тівирп. Необхідно пам'ятати, що рядки є символьні масиви, які дають змогу посилатися на кожен елемент (символ) окремо.

**Код програми 5.11. Демонстрація механізму відображення рядка в зворотному порядку**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      char str[80];

      cout << "Введіть рядок: "; gets(str);
      for(int i=strlen(str)-1; i>=0; i--) cout << str[i];

      getch(); return 0;
}
```

У наведеному нижче прикладі продемонструємо використання всіх розглянутих вище чотирьох рядкових функцій.

**Код програми 5.12. Демонстрація механізму використання всіх чотирьох рядкових функцій**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      char s1[80], s2[80];
      clrscr();              // Очищення екрану

      cout << "Введіть два рядки: "; gets(s1); gets(s2);
      cout << "Їх довжини дорівнюють: " << strlen(s1);
      cout << " " << strlen(s2) << endl;
      if(!strcmp(s1, s2))
            cout << "Рядки однакові" << endl;
      else
            cout << "Рядки не однакові" << endl;

      strcat(s1, s2);
      cout << s1 << endl;

      strcpy(s1, s2);
      cout << s1 << " і " << s2 << " ";
      cout << "тепер однакові" << endl;

      getch(); return 0;
}
```





Якщо запустити цю програму на виконання і на пропозицію ввести рядки "привіт" і "усім", то вона відобразить на екрані такі результати:

```
Їх довжини дорівнюють: 6 4
Рядки не однакові
привіт усім
всім і всім тепер однакові
```

Останнє нагадування: не забувайте, що функція **strcmp**() повертає значення **false**, якщо рядки однакові. Тому, якщо Ви перевіряєте рівність рядків, необхідно використовувати оператор "!" (НЕ), щоб реверсувати умову (тобто змінити її на зворотне), як було показано в попередній програмі.

### 5.3.5. Механізм використання ознаки завершення рядка

Факт завершення нульовими символами всіх C++-рядків можна використовувати для спрощення різних операцій над ними. Наведений нижче приклад дає змогу переконатися у тому, наскільки простий програмний код потрібен для заміни всіх символів рядка їх прописними еквівалентами.

**Код програми 5.13. Демонстрація механізму перетворення символів рядка в їх прописні еквіваленти**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <cctype>            // Робота з символьними аргументами
using namespace std;         // Використання стандартного простору імен

int main()
{
      char str[80];
      int i;

      strcpy(str, "test");
      for(i=0; str[i]; i++) str[i] = toupper(str[i]);
      cout << str;

      getch(); return 0;
}
```

Ця програма у процесі виконання виведе на екран слово TEST. Тут використовується бібліотечна функція **toupper**(), яка повертає прописний еквівалент свого символьного аргументу. Для виклику функції **toupper**() необхідно приєднати до програми заголовок <**cctype**>.

Зверніть увагу на те, що як умову завершення циклу **for** використано масив str, індексований змінній i, що керує (str[i]). Такий спосіб керування циклом цілком прийнятний, оскільки за дійсне значення у мові програмування C++ приймається будь-яке ненульове значення. Згадаймо, що всі друкарські символи представляються значеннями, не дорівнюють нулю, і тільки символ, що завершує рядок, дорівнює нулю. Отже, цей цикл працює доти, доки індекс не вкаже на нульову ознаку кінця рядка, тобто доки значення str[i] не стане нульовим. Оскільки нульовий символ відзначає кінець рядка, цикл зу-

пиняється точно там, де потрібно. При подальшій роботі з цим навчальним посібником Ви побачите багато прикладів, у яких нульова ознака кінця рядка використовують так само.

***Вартоа' нати!*** *Окрім функції* ***toupper()****, стандартна бібліотека мови програмування C++ містить багато інших функцій оброблення символів. Наприклад, функцію* ***toupper()*** *доповнює функція* ***tolower()****, яка повертає рядковий еквівалент свого символьного аргументу. Часто використовують такі функції, як* ***isalpha()****,* ***isdigit()****,* ***isspace()*** *і* ***ispunct()****, які приймають символьний аргумент і визначають, чи належить він до відповідної категорії. Наприклад, функція* ***isalpha()*** *повертає значення ІСТИНА, якщо її аргументом є буква (елемент алфавіту).*

## 5.4. Дво- та багатовимірні масиви

У програмуванні масив (англ. array) – одна з найпростіших структур даних, сукупність елементів одного типу даних, впорядкованих за індексами, які зазвичай репрезентовані натуральними числами, що визначають положення елемента в масиві. Масив може бути двовимірним (матрицею), та багатовимірним, тобто таким, де індексом є не одне число, а кортеж (сукупність) з декількох чисел, кількість яких збігається з розмірністю масиву. У переважній більшості мов програмування масив є стандартною вбудованою структурою даних.

### 5.4.1. Організація двовимірних масивів

У мові програмування C++ можна використовувати двовимірні масиви. Двовимірний масив, по суті, є списком одновимірних масивів. Щоб оголосити двовимірний масив цілочисельних значень розміром 10×20 з іменем num, достатньо записати таке:

```
int num[10][20];
```

Зверніть особливу увагу на це оголошення. На відміну від багатьох інших мов програмування, у яких під час оголошення масиву значення розмірностей відокремлюються комами, у мові програмування C++ кожна розмірність полягає у власну пару квадратних дужок.

Щоб отримати доступ до елемента масиву num з координатами 3×5, необхідно використовувати запис num[3][5]. У наведеному нижче прикладі в двовимірний масив поміщаються послідовні числа від 1 до 12.

**Код програми 5.14. Демонстрація механізму роботи з елементами двовимірного масиву**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
      int t, i, num[3][4];
      for(t=0; t<3; ++t) {
            for(i=0; i<4; ++i) {
                  num[t][i] = (t*4)+i+1;
                  cout << num[t][i] << " ";
            }
            cout << endl;
      }
      getch(); return 0;
}
```

У наведеному прикладі елемент масиву num[0][0] набуде значення 1, елемент num[0][1] – значення 2, елемент num[0][2] – значення 3 і т.д. Значення елемента num[2][3] буде дорівнювати числу 12. Схематично цей масив можна представити так, як це показано на рис. 5.1.

У двовимірному масиві позиція будь-якого елемента визначається двома індексами. Якщо представити двовимірний масив у вигляді таблиці даних, то один індекс означає рядок, а другий – стовпець. З цього виходить, якщо доступ до елементів масиву надати в порядку, у якому вони реально зберігаються в пам'яті, то правий індекс змінюватиметься швидше, ніж лівий.

***Нео! хідноатам'ятати!*** *Місце зберігання для всіх елементів масиву визначається під час компілювання програми. Окрім цього, пам'ять, виділена для зберігання елементів масиву, використовується протягом всього часу наявності масиву.*

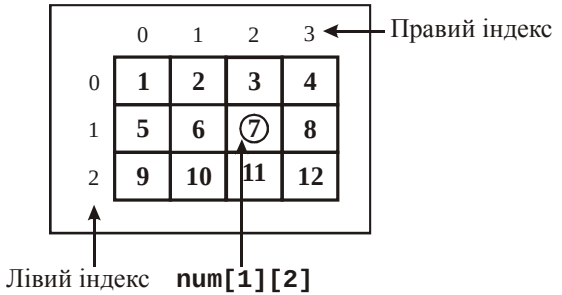



**Рис. 5.1. Схематичне представлення масиву num**

Для визначення кількості байтів пам'яті, займаної двовимірним масивом, використовують така формула:

кі-сть байтів = кі-сть рядків × кі-сть стовпців × розмір типу в байтах

Отже, двовимірний цілочисельний масив розмірністю 10×5 займає в пам'яті 10×5×2, тобто 100 байтів (якщо цілочисельний тип має розмір 2 байт).

### 5.4.2. Організація багатовимірних масивів

У мові програмування C++, окрім двовимірних, можна визначати масиви трьох і більш вимірів. Ось як оголошується багатовимірний масив:

*тип ім'я*[*розмір1*][*розмір2*]...[*розмірN*];

Наприклад, за допомогою такого оголошення створюється тривимірний цілочисельний масив розміром 4×10×3:

```
int multidim[4][10][3];
```

Як було зазначено вище, пам'ять, виділена для зберігання всіх елементів масиву, використовується протягом всього часу наявності масиву. Масиви з числом вимірювань, що перевищує три, використовуються нечасто, хоча б тому, що для їх зберігання потрібен великий об'єм пам'яті. Наприклад, зберігання елементів чотиривимірного символьного масиву розміром 10×6×9×4 займе 2 160 байтів. А якщо кожну розмірність збільшити в 10 разів, то займана масивом пам'ять зросте до 21 600 000 байтів. Як бачимо, великі багатовимірні масиви здатні "з'їсти" великий об'єм пам'яті, а програма, яка їх використовує, може дуже швидко наштовхнутися на проблему відсутності пам'яті.

## 5.5. Ініціалізація елементів масивів

Ознакою масиву при описі є наявність парних дужок []. Константа або константний вираз в квадратних дужках задає число елементів масиву. При описі масиву може бути виконана ініціалізація його елементів. Існує два методи ініціалізації елементів масивів: розмірних і безрозмірних масивів.

### 5.5.1. Ініціалізація елементів "розмірних" масивів

У мові програмування C++ передбачено можливість ініціалізації елементів масиву. Формат ініціалізації елементів масиву подібний до формату ініціалізації інших змінних:

*тип ім'я_масиву*[*розмір*] = [*перелік_значень*];

У цьому записі елемент *перелік_значень* є перелік значень ініціалізації елементів масиву, розділених між собою комами. Тип кожного значення ініціалізації повинен бути сумісний з базовим типом масиву (елементом *тип*). Перше значення ініціалізації буде збережено в першій позиції масиву, друге значення – в другій і т.д. Зверніть увагу на те, що крапка з комою ставиться після закритої фігурної дужки (}).

Наприклад, в такому прикладі 10-елементний цілочисельний масив ініціалізувався числами від 1 до 10:

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

Після виконання цієї настанови елемент Array[0] набуде значення 1, а елемент Array[9] –значення 10.

Для символьних масивів, призначених для зберігання рядків, передбачено скорочений варіант ініціалізації, який має таку форму:

```
char ім'я_масиву[розмір] = "рядок";
```

Наприклад, такий фрагмент коду програми ініціалізує масив str фразою "привіт".

```
char str[7] = "привіт";
```

Це рівнозначно по-елементній ініціалізації:





```
char str[7] = {'п', 'р', 'и', 'в', 'і', 'т', '\0'};
```

Оскільки у мові програмування C++ рядки повинні завершуватися нульовим символом, то переконайтеся, що під час оголошення масиву його розмір вказано з врахуванням ознаки кінця. Саме тому в попередньому прикладі масив str оголошений як 7-елементний, хоча у слові "привіт" тільки шість букв. Під час використання рядкового літерала компілятор добавляє нульову ознаку кінця рядка автоматично.

Багатовимірні масиви ініціалізуються за аналогією з одновимірними. Наприклад, такий фрагмент коду програми масив Array ініціалізувався числами від 1 до 10 і квадратами цих чисел.

```
int Array[10][2] = {
        1, 1,
        2, 4,
        3, 9,
        4, 16,
        5, 25,
        6, 36,
        7, 49,
        8, 64,
        9, 81,
        10, 100
};
```

Тепер розглянемо, як елементи масиву Array розташовуються в пам'яті (рис. 5.2).

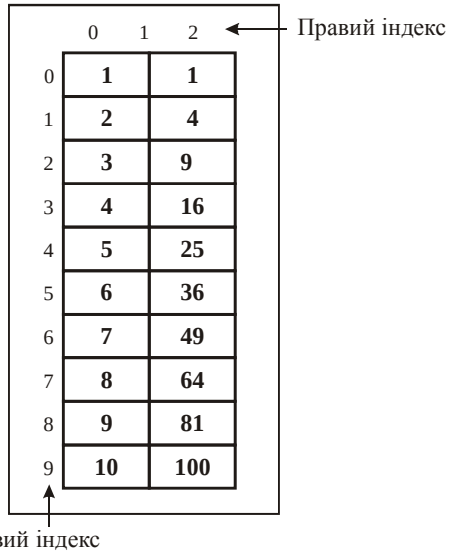



**Рис. 5.2. Схематичне представлення ініціалізованого масиву** Array

Під час ініціалізації багатовимірного масиву перелік ініціалізацій кожної розмірності (підгрупу ініціалізацій) можна помістити у фігурні дужки. Ось, наприклад, як виглядає ще один варіант запису попереднього оголошення:

```
int Array[10][2] = {
        {1, 1),
        {2, 4},
        {3, 9},
        {4, 16},
        {5, 25},
        {6, 36},
        {7, 49},
        {8, 64},
        {9, 81},
        {10, 100}
};
```

Під час використання підгруп ініціалізацій відсутні члени підгрупи ініціалізуються нульовими значеннями автоматично.

У наведеному нижче коді програми масив Array використовують для пошуку квадрата числа, введеного користувачем. Програма спочатку здійснює пошук заданого числа в масиві, а потім виводить відповідне йому значення квадрата.

**Код програми 5.15. Демонстрація механізму пошуку потрібного елемента у двовимірному масиві**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int Array[10][2] = {
        {1, 1},
        {2, 4},
        {3, 9},
        {4, 16),
        (5, 25},
        (6, 36},
        {7, 49},
        (8, 64},
        (9, 81},
        {10, 100}
        };

int main()
{
        int i, j;

        for(;;) {
                cout << "Введіть число від 1 до 10: "; cin >> i;
                if(i>=1)&&(i<=10) break;
        }
```





```
        // Пошук значення i.
        for(j=0; j<10; j++)
                if(Array[j][0] == i) break;
        cout << "Квадрат числа " << i << " дорівнює " << Array[j][1];

        getch(); return 0;
}
```

Глобальні масиви ініціалізувалися на початку виконання програми, а локальні – під час кожного виклику функції, у якій вони містяться. Розглянемо такий приклад:

**Код програми 5.16. Демонстрація механізму ініціалізації елементів двовимірного масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

void Fun1();

int main()
{
        Fun1();
        Fun1();

        getch(); return 0;
}

void Fun1() {
        char str[80] ="Це просто тест\n";

        cout << str;
        strcpy(str, "Змінено\n");          // Змінюємо значення рядка str.
        cout << str;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Це просто тест
Змінено
Це просто тест
Змінено
```

У цьому коді програми масив str ініціалізувався під час кожного виклику функції Fun1(). Той факт, що під час її виконання масив str змінюється, ніяк не впливає на його повторну ініціалізацію при подальших викликах функції Fun1(). Тому під час кожного входження в неї на екрані монітора з'являється такий текст:

```
Це просто тест
```

### 5.5.2. "Безрозмірна" ініціалізація елементів масивів

Припустимо, що ми використовуємо такий варіант ініціалізації елементів масиву для побудови таблиці повідомлень про помилки:

```
char e1[14] = "Ділення на 0\n";
char e2[23] = "Кінець файлу\n";
char e3[21] = "У доступі відмовлено\n";
```

Неважко припустити, що вручну незручно підраховувати символи у кожному повідомленні, щоб визначити коректний розмір масиву. На щастя, у мові програмування C++ передбачено можливість автоматичного визначення довжини масивів шляхом використання їх "безрозмірного" формату. Якщо в настанові ініціалізації масиву не вказано його розмір, то мова програмування C++ автоматично створить масив, розмір якого буде достатнім для зберігання всіх значень ініціалізацій. При такому підході попередній варіант ініціалізації елементів масиву для побудови таблиці повідомлень про помилки можна переписати так:

```
char e1[] = "Ділення на 0\n";
char e2[] = "Кінець файлу\n";
char e3[] = "У доступі відмовлено \n";
```

Крім зручності в первинному визначенні масивів, метод "безрозмірної" ініціалізації дає змогу змінити будь-яке повідомлення без переліку його довжини. Тим самим усувається можливість внесення помилок, викликаних випадковим прорахунком.

"Безрозмірна" ініціалізація елементів масивів не обмежується одновимірними масивами. Під час ініціалізації багатовимірних масивів Вам треба вказати усі дані, за винятком крайньої зліва розмірності, щоб C++-компілятор міг належним чином індексувати масив. Використовуючи "безрозмірну" ініціалізацію масивів, можна створювати таблиці різної довжини, даючи змогу компіляторові автоматично виділяти область пам'яті, достатню для їх зберігання.

У такому прикладі масив Array[][] оголошується як "безрозмірний":

```
int Array[][2] = {
        1, 1,
        2, 4,
        3, 9,
        4, 16,
        5, 25,
        6, 36,
        7, 49,
        8, 64,
        9, 81,
        10, 100
};
```

Перевага такої форми оголошення перед "габаритною" (з точним указанням усіх розмірностей) полягає у тому, що програміст може подовжувати або укорочувати таблицю значень ініціалізації, не змінюючи розмірності масиву.





## 5.6. Проблема організації масиву рядків

Існує спеціальна форма двовимірного символьного масиву, яка є масивом рядків. У його використанні немає нічого незвичайного. Наприклад, при програмуванні баз даних для з'ясування коректності команд, що вводяться користувачем, вхідні дані порівнюються з вмістом масиву рядків, у якому записано допустимі у цьому додатку команди.

### 5.6.1. Побудова масивів рядків

Для побудови масиву рядків використовується двовимірний символьний масив, у якому розмір лівого індексу визначає кількість рядків, а розмір правого – максимальну довжину кожного рядка. Наприклад, у процесі виконання такої настанови оголошується масив, призначений для зберігання 30 рядків завдовжки 80 символів:

```
char strArray[30][80];
```

***Масив рядків** – спеціальна форма двовимірного масиву символів.*

Отримати доступ до окремого рядка достатньо просто: достатньо вказати тільки лівий індекс. Наприклад, така настанова викликає функцію **gets**() для запису третього рядка масиву:

```
gets(strArray[2]);
```

Щоб краще зрозуміти, як потрібно поводитися з масивами рядків, розглянемо таку коротку програму, яка приймає рядки тексту, що вводяться з клавіатури, і відображає їх на екрані після введення порожнього рядка.

**Код програми 5.17. Демонстрація механізму введення рядка тексту і відображення його на екрані**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
        int t, i;
        char text[100][80];
        clrscr();            // Очищення екрану

        for(t=0; t<100; t++) {
                cout << t << ": "; gets(text[t]);
                if(!text[t][0]) break; // Вихід з циклу за порожнім рядком.
        }
        // Відображення рядків на екрані.
        for(i=0; i<t; i++)
                cout << text[i] << endl;
        getch(); return 0;
}
```

Зверніть увагу на те, як у програмі здійснюється перевірка на введення порожнього рядка. Функція **gets**() повертає рядок нульової довжини, якщо єдиною натиснутою клавішею виявилася клавіша <Enter>. Це означає, що першим байтом у рядку буде нульовий символ. Нульове значення завжди інтерпретується як помилкове, але узяте із запереченням (!) дає значення ІСТИНА, яке дає змогу виконати негайний вихід з циклу за допомогою настанови **break**.

### 5.6.2. Приклад використання масивів рядків

Масиви рядків зазвичай використовують для оброблення таблиць даних. Розглянемо, наприклад, спрощену базу даних службовців, у якій зберігається ім'я, номер телефону, кількість годин, відпрацьованих службовцями за звітний період, і розмір погодинного окладу для кожного службовця. Щоб створити таку програму для колективу, що складається з десяти службовців, визначимо чотири масиви (з них перші два будуть масивами рядків).

```
char name[10][80];          // Масив імен службовців.
char phone[10][20];         // Масив телефонних номерів службовців.
float hours[10];            // Масив годин, відпрацьованих за тиждень.
float oklad[10];            // Масив погодинних окладів.
```

Щоб ввести інформацію про кожного службовця, спробуємо скористатися функцією Enter().

```
// Функція введення інформації в базу даних
void Enter()
{
    char tmp[80];

    for(int i=0; i<10; i++) {
        cout << "Введіть прізвище службовця: "; cin >> name[i];
        cout << "Введіть номер телефону службовця: "; cin >> phone[i];
        cout << "Введіть кількість відпрацьованих годин: "; cin >> hours[i];
        cout << "Введіть оклад службовця: "; cin >> oklad[i];
    }
}
```

На підставі введених даних можна скласти звіт, обчисливши заробітну плату, яка належить кожному службовцю. Для цього скористаємося такою функцією report():

```
// Відображення звіту
void report()
{
    for(int i=0; i<10; i++) {
        cout << name[i] << " " << phone[i] << endl;
        cout << "Заробітна плата за тиждень: " << oklad[i]*hours[i];
        cout << endl;
    }
}
```





Повністю програму бази даних службовців наведено нижче. Зверніть особливу увагу на те, як реалізується доступ до кожного масиву. Ця версія програми ведення бази даних службовців ще далека від досконалості, оскільки введена в неї інформація втрачається відразу ж після виходу з програми. Трохи згодом Ви навчитеся зберігати інформацію в дисковому файлі.

**Код програми 5.18. Демонстрація простої програми ведення бази даних службовців**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

char name[10][80];           // Масив імен службовців.
char phone[10][20];          // Масив телефонних номерів службовців.
float hours[10];             // Масив годин, відпрацьованих за тиждень.
float oklad[10];             // Масив погодинних окладів.

int MenuSelect();
void Enter(), report();

int main()
{
    int vybir;
    clrscr();           // Очищення екрану

    do {
        // Отримуємо команду вибрану користувачем.
        vybir = MenuSelect();
        switch(vybir) {
            case 0: break;
            case 1: Enter();
                    break;
            case 2: report();
                    break;
            default: cout << "Спробуйте ще раз.\n" << endl;
        }
    } while(vybir != 0);
    getch(); return 0;
}

// Функція повертає команду, вибрану користувачем.
int MenuSelect()
{
    int vybir;

    cout << "0. Вихід з програми" << endl;
    cout << "1. Введення інформації" << endl;
    cout << "2. Генерування звіту" << endl;
    cout << endl << " Виберіть команду: "; cin >> vybir;
    return vybir;
}
```

```
// Функція введення інформації в базу даних.
void Enter()
{
        char tmp[80];

        for(int i=0; i<10; i++) {
                cout << "Введіть прізвище службовця: "; cin >> name[i];
                cout << "Введіть номер телефону службовця: "; cin >> phone[i];
                cout << "Введіть кількість відпрацьованих годин: "; cin >> hours[i];
                cout << "Введіть оклад службовця: "; cin >> oklad[i];
        }
}

// Відображення звіту.
void report()
{
        for(int i=0; i<10; i++) {
                cout << name[i] << " " << phone[i] << endl;
                cout << "Заробітна плата за тиждень: " << oklad[i]*hours[i];
                cout << endl;
        }
}
```





# Розділ 6. ОСОБЛИВОСТІ ЗАСТОСУВАННЯ ПОКАЖЧИКІВ

Покажчики, поза сумнівом, – один з найважливіших і складних аспектів мови програмування С++. Значною мірою потужність багатьох засобів мови програмування С++ визначається використанням покажчиків. Наприклад, завдяки ним забезпечується підтримка зв'язних списків і динамічного виділення області пам'яті, саме вони дають змогу функціям змінювати значення своїх аргументів. Однак ці питання будуть розглядатися в подальших розділах, а поки що (тобто у цьому розділі) розглянемо основні особливості застосування покажчиків і продемонструємо, як можна уникнути деяких потенційних проблем, пов'язаних з їх використанням.

Під час розгляду основних понять про покажчики нам доведеться використовувати таке поняття як розмір базових С++-типів даних. Для цього запам'ятаємо тільки те, що символи займають в пам'яті один байт, цілочисельні значення – чотири, значення з плинною крапкою типу **float** – чотири, а значення з плинною крапкою типу **double** – вісім (ці розміри характерні для типового 32-розрядного середовища).

## 6.1. Основні поняття про покажчики

***Покажчики*** – змінні, які зберігають адреси іншої змінної. Найчастіше ці адреси позначають місцезнаходження в пам'яті інших змінних. Наприклад, якщо змінна x містить адресу змінної y, то про змінну x говорять, що вона "вказує" на y.

*Змінні-покажчики* (або *змінні типу покажчика*) мають бути відповідно оголошені. Формат оголошення змінної-покажчика є таким:

*тип *ім'я_змінної*;

У цьому записі елемент *тип* означає базовий тип покажчика, причому він повинен бути допустимим С++-типом. Елемент *ім'я_змінної* є іменем змінної-покажчика.

Для розуміння сказаного розглянемо такий приклад. Щоб оголосити змінну p покажчиком на **int**-значення, використаємо таку настанову:

```
int *p;
```

Для оголошення покажчика на **float**-значення використаємо таку настанову:

```
float *p;
```

У загальному випадку використання символу "зірочка" (*) перед іменем змінної в настанові оголошення перетворює цю змінну на покажчик.

*Базовий тип покажчика визначає тип даних, на які він посилатиметься.*

Тип даних, на які посилатиметься покажчик, визначається його базовим типом. Розглянемо ще один приклад:

```
int *ip;            // Покажчик на цілочисельне значення
double *dp;         // Покажчик на значення типу double
```

Як зазначено в коментарях до цієї програми, змінна ip – покажчик на **int**-значення, оскільки його базовим типом є тип **int**, а змінна dp – покажчик на **double**-значення, оскільки його базовим типом є тип **double**, Отже, в попередніх прикладах змінну ip можна використовувати для вказівки на **int**-значення, а змінну dp – на **double**-значення. Проте запам'ятайте: не існує реального засобу, який би зміг перешкодити покажчику посилатися на "казна-що". Ось через це покажчики потенційно небезпечні.

## 6.2. Механізм використання покажчиків у поєднанні з операторами присвоєння

### 6.2.1. Оператори роботи з покажчиками

З покажчиками використовуються два оператори: "*" і "&". Оператор "&" – унарний. Він повертає адресу пам'яті, за якою розташований його операнд[1]. Наприклад, у процесі виконання такого фрагмента коду програми

```
ptr = &balance;
```

у змінну ptr поміщається адреса змінної balance. Ця адреса відповідає області у внутрішній пам'яті комп'ютера, яка належить змінній balance. Виконання цієї настанови ніяк не впливає на значення змінної balance. Призначення оператора "&" можна "перекласти" українською мовою як "адреса змінної", перед якою він знаходиться. Отже, наведену вище настанову присвоєння можна виразити так: "змінна ptr отримує адресу змінної balance". Щоб краще зрозуміти суть цього присвоєння, припустимо, що змінна balance розташована у області пам'яті з адресою 100. Отже, після виконання цієї настанови змінна ptr набуде значення 100.

Другий оператор роботи з покажчиками (*) слугує доповненням до першого (&). Це також унарний оператор, але він звертається до значення змінної, розташованої за адресою, заданою його операндом. Іншими словами, він посилається на значення змінної, яка адресується заданим покажчиком. Якщо (продовжуючи роботу з попередньою настановою присвоєння) змінна ptr містить адресу змінної balance, то у процесі виконання настанови

```
value = *ptr;
```

змінній **value** буде присвоєне значення змінної balance, на яку вказує змінна ptr. Наприклад, якщо змінна balance містить значення 3200, то після виконання останньої настанови змінна **value** міститиме значення 3200, оскільки це якраз те значення, яке зберігається за адресою 100. Призначення оператора "*" можна виразити словосполученням "за адресою". У цьому випадку попередню настанову можна прочитати так: "змінна **value** набуває значення (розташоване) за адресою ptr". Дію наведених вище двох настанов схематично показано на рис. 6.1.

[1] Згадаймо: унарному оператору потрібен тільки один операнд.





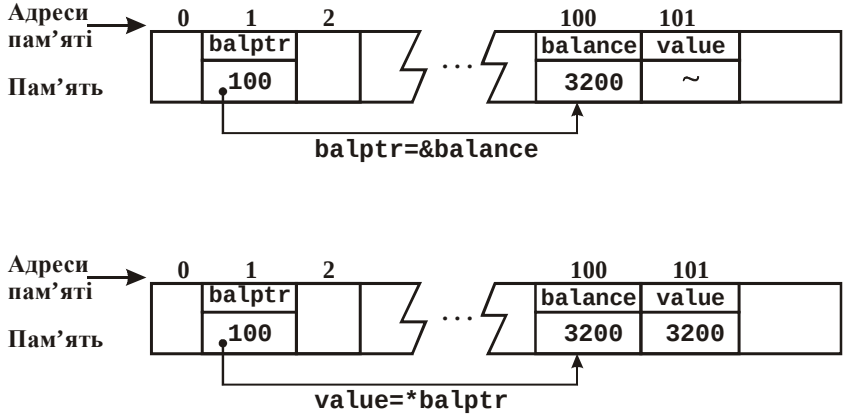



**Рис. 6.1. Дія операторів "*" і "&"**

Послідовність операцій, відображених на рис. 6.1, безпосередньо реалізується у наведеному нижче коді програми.

**Код програми 6.1. Демонстрація механізму використання покажчиків у поєднанні з операторами**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int balance;
    int *ptr;
    int value;

    balance = 3200;
    ptr = &balance;
    value = *ptr;
    cout << "Баланс дорівнює: " << value << endl;

    getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
Баланс дорівнює: 3200
```

Шкода, але знак множення (*) і оператор із значенням "за адресою" позначаються однаковими символами "зірочка", що іноді збиває з пантелику новачків у мові програмування С++. Ці операції ніяк не пов'язані одна з іншою. Майте на увазі, що оператори "*" і "&" мають вищий пріоритет, ніж будь-

який з арифметичних операторів, за винятком унарного мінуса, пріоритет якого такий самий, як у операторів, що вживаються для роботи з покажчиками.

***Операція непрямого доступу*** *– механізм використання покажчика для доступу до деякого об'єкта.*

Операції, що виконуються за допомогою покажчиків, часто називають *операціями непрямого доступу*, оскільки ми опосередковано отримуємо доступ до змінної за допомогою деякої іншої змінної.

### 6.2.2. Важливість застосування базового типу покажчика

На прикладі попередньої програми було показано можливість присвоєння змінній **value** значення змінної balance за допомогою операції непрямого доступу, тобто з використанням покажчика. Можливо, при цьому у Вас промайнуло запитання: "Як C++-компілятор дізнається про те, скільки необхідно скопіювати байтів у змінну **value** з області пам'яті, яка адресується покажчиком ptr?". Сформулюємо те саме запитання в більш загальному вигляді: як C++-компілятор передає належну кількість байтів у процесі виконання операції присвоєння з використанням покажчика? Відповідь звучить так. Тип даних, який адресується покажчиком, вважається базовим типом покажчика. Оскільки ptr є покажчиком на цілочисельний тип, то у цьому випадку C++-компілятор скопіює в змінну **value** з області пам'яті, яка адресується покажчиком ptr, чотири байти інформації (що справедливе для 32-розрядного середовища). Але якби ми мали справу з **double**-покажчиком, то в аналогічній ситуації скопіювалося б вісім байтів інформації.

Змінні-покажчики повинні завжди вказувати на відповідний тип даних. Наприклад, під час оголошення покажчика типу **int** компілятор "припускає", що всі значення, на які посилається цей покажчик, мають тип **int**. C++-компілятор просто не дасть змоги виконати операцію присвоєння за участю покажчиків (з обох боків від оператора присвоєння), якщо типи цих покажчиків несумісні (по суті не однакові).

Наприклад, такий фрагмент коду програми є некоректним:

```
int *p;
double f;
//... p = &f; // Помилка!
```

Некоректність цього фрагмента полягає в неприпустимості присвоєння **double**-покажчика **int**-покажчику. Вираз &f генерує покажчик на **double**-значення, а p – покажчик на цілочисельний тип **int**. Ці два типи несумісні, тому компілятор відзначить цю настанову як помилкову і не скомпілює програму.

Як було заявлено вище, під час присвоєння два покажчики мають бути сумісні за типом, проте це серйозне обмеження можна подолати (правда, на свій страх і ризик) за допомогою операції приведення типів. Наприклад, такий фрагмент коду програми тепер формально є більш коректним:

```
int *p;
double f;
//...
p = (int *) &f; // Тепер формальне все ОК!
```

Операція приведення до типу (**int** *) викличе перетворення **double**- до **int**-покажчика. Все ж таки використання операції приведення з цією метою є дещо сумнівним, оскільки саме базовий тип покажчика визначає, як компілятор поводитиметься з даними, на які він посилається. У цьому випадку, незважаючи на те, що покажчик p (після виконання останньої настанови) насправді вказує на значення з плинною крапкою, компілятор, як і раніше, "вважає", що він вказує на цілочисельне значення (оскільки p за визначенням – **int**-покажчик).

Щоб краще зрозуміти, чому використання операції приведення типів під час присвоєння одного покажчика іншому не завжди є допустимим, розглянемо таку програму.

**Код програми 6.2. Демонстрація роботи неправильного написання коду програми**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    double x, y;
    int *p;
    x = 123.23;
    p = (int *) &x;      // Використовуємо операцію приведення типів
                         // для присвоєння double-покажчика int-покажчику.
    y = *p;              // Що відбувається у процесі виконання цієї настанови?
    cout << y;           // Що виведе ця настанова?

    getch(); return 0;
}
```

Як бачимо, у цьому коді програми змінній p (точніше, покажчику на цілочисельне значення) присвоюється адреса змінної x (яка має тип **double**). Отже, коли змінній y присвоюється значення, яке адресується покажчиком p, то змінна y отримує тільки чотири байти даних (а не всі вісім, що є потрібними для **double**-значення), оскільки p – покажчик на цілочисельний тип **int**. Таким чином, у процесі виконання **cout**-настанови на екран буде виведено не число 123.23, а, висловлюючись мовою програмістів, "сміття"[1], яке насправді є молодшими 4-ма байтами мантиси.

### 6.2.3. Присвоєння значень за допомогою покажчиків

Під час присвоєння значення області пам'яті, яка адресується покажчиком, його (покажчик) можна використовувати з лівого боку від оператора

[1] Спробуйте виконати цю програму і переконайтеся в цьому самі.

присвоєння. Наприклад, у процесі виконання такої настанови (якщо p – покажчик на цілочисельний тип)

```
*p = 101;
```

число 101 присвоюється області пам'яті, в p, яка адресується покажчиком, Таким чином, цю настанову можна прочитати так: "за адресою p поміщаємо значення 101". Щоб інкрементувати або декрементувати значення, розташоване у області пам'яті, яка адресується покажчиком, можна використовувати настанову, подібну до такої:

```
(*p)++;
```

Круглі дужки тут є обов'язковими, оскільки оператор "*" має нижчий пріоритет, ніж оператор "++".

Присвоєння значень з використанням покажчиків продемонстровано у наведеному нижче коді програми.

**Код програми 6.3. Демонстрація механізму присвоєння значень з використанням покажчиків**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int *p, n;
    p = &n;
    *p = 100;
    cout << n << " ";
    (*p)++;
    cout << n << " ";
    (*p)--;
    cout << n << endl;

    getch(); return 0;
}
```

Ось такі результати генерує ця програма.

```
100 101 100
```

## 6.3. Механізм використання покажчиків у виразах

Покажчики можна використовувати у більшості виразів, що допускаються мовою програмування C++. Але при цьому потрібно застосовувати спеціальні правила і не забувати, що деякі частини таких виразів необхідно брати в круглі дужки, щоб гарантовано отримати бажаний результат.

### 6.3.1. Арифметичні операції над покажчиками

З покажчиками можна використовувати тільки чотири арифметичних оператори: ++; --; + і -. Щоб краще зрозуміти, що відбувається у процесі виконання арифметичних дій з покажчиками, почнемо з конкретного прикладу. Нехай p1 – покажчик на **int**-змінну з поточним значенням 2 000 (тобто p1 містить адресу 2 000). Після виконання (у 32-розрядному середовищі) виразу

```
p1++;
```

вміст змінної-покажчика p1 дорівнюватиме 2 004, а не 2 001! Йдеться про те, що під час кожного інкрементування покажчик p1 вказуватиме на *наступне* **int**-значення. Для операції декрементування справедливе зворотне твердження, тобто під час кожного декрементування значення p1 зменшуватиметься на 4. Наприклад, після виконання настанови

```
p1--;
```

покажчик p1 матиме значення 1 996, якщо до цього воно дорівнювало 2 000. Отже, кожного разу, коли покажчик інкрементується, він вказуватиме на область пам'яті, що містить наступний елемент базового типу цього покажчика. А під час кожного декрементування він вказуватиме на область пам'яті, що містить попередній елемент базового типу цього покажчика.

Для покажчиків на символьні значення результат операцій інкрементування і декрементування буде таким самим, як при "нормальній" арифметиці, оскільки символи займають тільки один байт. Але під час використання будь-якого іншого типу покажчика у процесі інкрементування або декрементування значення змінної-покажчика збільшуватиметься або зменшуватиметься на величину, що дорівнює розміру його базового типу.

Арифметичні операції над покажчиками не обмежуються використанням операторів інкремента і декремента. Із значеннями покажчиків можна виконувати операції додавання і віднімання, використовуючи як другий операнд цілочисельні значення. Вираз

```
p1 = p1 + 9;
```

примушує p1 посилатися на дев'ятий елемент базового типу покажчика p1 щодо елемента, на який p1 посилався до виконання цієї настанови.

Хоча додавати покажчики не можна, проте один покажчик все ж таки можна відняти від іншого (якщо вони обидва мають один і той самий базовий тип). Різниця покаже кількість елементів базового типу, які розділяють ці два покажчики.

Крім додавання покажчика з цілочисельним значенням і віднімання його від покажчика, а також обчислення різниці двох покажчиків, над покажчиками жодні інші арифметичні операції не виконуються. Наприклад, до покажчиків не можна додавати **float**- або **double**-значення.

Щоб зрозуміти, як формується результат виконання арифметичних операцій над покажчиками, виконаємо таку коротку програму. Вона виводить реальні фізичні адреси, які містять покажчик на **int**-значення (c) і покажчик на **float**-значення (f). Зверніть увагу на кожну зміну адреси (залежну від базового типу покажчика), яка відбувається під час повторення циклу[1]. Зверніть увагу також на те, що під час використання покажчика в **cout**-настанові його адреса

[1] Для більшості 32-розрядних компіляторів значення i збільшуватиметься на 4, а значення f – на 8.

автоматично відображається у форматі адресації, що вживається для поточного процесора і середовища виконання.

**Код програми 6.4. Демонстрація механізму виконання арифметичних операцій для роботи з покажчиками**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      int *c, d[10];
      double *f, g[10];

      c = d;
      f = g;

      for(int x=0; x<10; x++)
            cout << c + x << " " << f + x << endl;

      getch(); return 0;
}
```

Ось як виглядають можливі варіанти виконання цієї програми[1]:

| | |
|---|---|
| 0012FE5C 0012FE84 | 1245016 1244936 |
| 0012FE60 0012FE8C | 1245020 1244944 |
| 0012FE64 0012FE94 | 1245024 1244952 |
| 0012FE68 0012FE9C | 1245028 1244960 |
| 0012FE6C 0012FEA4 | 1245032 1244968 |
| 0012FE70 0012FEAC | 1245036 1244976 |
| 0012FE74 0012FEB4 | 1245040 1244984 |
| 0012FE78 0012FEBC | 1245044 1244992 |
| 0012FE7C 0012FEC4 | 1245048 1245000 |
| 0012FE80 0012FECC | 1245052 1245008 |

***Нео! хідноа пам'ятати!*** *Всі арифметичні операції над покажчиками виконуються щодо базового типу покажчика.*

### 6.3.2. Порівняння покажчиків

Покажчики можна порівнювати, використовуючи оператори відношення ==, < і >. Проте для того, щоби результат порівняння покажчиків піддавався інтерпретації, порівнювані покажчики мають бути якимсь чином пов'язані між собою. Наприклад, якщо p1 і p2 – покажчики, які посилаються на дві окремі та ніяк не пов'язані між собою змінні, то будь-яке порівняння p1 і p2 в загальному випадку немає сенсу. Але, якщо покажчики p1 і p2 вказують на змінні, між якими існує деякий зв'язок (як, наприклад, між елементами одного і того ж масиву), то результат порівняння покажчиків p1 і p2 може мати певний сенс. Нижче у цьому підрозділі розглянемо приклад програми, у якій використовується порівняння покажчиків.

[1] Ваші результати можуть відрізнятися від наведених, але інтервали між значеннями повинні бути такими самими





## 6.4. Покажчики і масиви – взаємозамінні поняття

У мові програмування C++ покажчики і масиви тісно пов'язані між собою, причому настільки, що часто поняття "покажчик" і "масив" взаємозамінні. У цьому підрозділі ми спробуємо простежити цей зв'язок. Для початку розглянемо такий фрагмент програми:

```
char str[80];
char *p1;
p1 = str;
```

У цьому записі str є іменем масиву, що містить 80 символів, а p1 – покажчик на тип **char**. Особливий інтерес представляє третій рядок, у процесі виконання якого змінній p1 присвоюється адреса першого елемента масиву str. Іншими словами, після цього присвоєння p1 вказуватиме на елемент str[0]. Йдеться про те, що у мові програмування C++ використання імені масиву без індексу генерує покажчик на перший елемент цього масиву. Таким чином, у процесі виконання операції присвоєння p1 = str адреса str[0] присвоюється покажчику p1. Це і є ключовим моментом, який необхідно чітко розуміти: неіндексоване ім'я масиву, використане у виразі, означає покажчик на початок цього масиву.

*Ім'я масиву без індексу утворює покажчик на початок цього масиву.*

Оскільки після розглянутого вище присвоєння покажчик p1 вказуватиме на початок масиву str, то p1 можна використовувати для доступу до елементів цього масиву. Наприклад, якщо потрібно отримати доступ до п'ятого елемента масиву str, використовується один з таких виразів: str[4] або *(p1+4).

У обох випадках буде виконане звернення до п'ятого елемента. Необхідно пам'ятати, що індексування елементів масиву починається з нуля, тому при індексі, що дорівнює чотирьом, забезпечується доступ до п'ятого елемента. Таке саме враження справляє підсумовування значення початкового покажчика (p1) з числом 4, оскільки p1 вказує на перший елемент масиву str.

Необхідність використання круглих дужок, у які поміщено вираз p1+4, пов'язана з тим, що оператор "*" має вищий пріоритет, ніж оператор додавання "+". Без цих круглих дужок вираз звівся би до отримання значення, яке адресується покажчиком p1, тобто значення першого елемента масиву, яке потім було б збільшено на 4.

***Вартоа нати!*** *Переконайтеся зайвий раз в правильності використання круглих дужок у виразах з покажчиками. Інакше помилку буде важко відшукати, оскільки зовні програма може виглядати цілком коректною. Якщо у Вас є сумніви в потребі їх використання, то прийміть рішення на їх зиск − шкоди від цього не буде.*

### 6.4.1. Основні відмінності між індексуванням елементів масивів і арифметичними операціями над покажчиками

У мові програмування C++ передбачено два способи доступу до елементів масивів: за допомогою індексування елементів масивів і арифметики покажчиків. Йдеться про те, що арифметичні операції над покажчиками іноді виконуються швидше, ніж індексування елементів масивів, особливо під час доступу до елементів, розташування яких відрізняється строгою впорядкованістю. Оскільки швидкодія часто є визначальним чинником при виборі тих або інших рішень під час програмування, то використання покажчиків для доступу до елементів масиву – характерна особливість багатьох C++-програм. Окрім цього, іноді покажчики дають змогу написати дещо компактніший код програми порівняно з використанням індексування елементів масивів.

Щоб краще зрозуміти відмінність між індексуванням елементів масивів і арифметичними операціями над покажчиками, розглянемо дві версії однієї і тієї ж самої програми. У наведеному нижче коді програми з рядка тексту виділяють слова, розділені між собою пропусками. Наприклад, з рядка "Привіт, друг" програма повинна виділити слова "Привіт" і "друг". Програмісти зазвичай називають такі розмежовані символьні послідовності *лексемами* (tmp). У процесі виконання програми вхідний рядок по-символьно копіюється в інший масив (з іменем tmp) доти, доки не трапиться пропуск. Після цього виділена лексема виводиться на екран, і процес триває доти, доки не буде досягнуто кінця рядка. Наприклад, якщо як вхідний рядок використовувати рядок "Це тільки простий тест.", то програма відобразить таке:

```
Це
тільки
простий
тест.
```

Ось як виглядає версія програми поділу рядка на слова з використанням арифметичних покажчиків.

**Код програми 6.5. Демонстрація механізму поділу рядка на слова з використанням арифметичних покажчиків**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[80], tmp[80];
    char *p, *q;

    cout << "Введіть речення: "; gets(str);
    p = str;
```





```
    // Зчитуємо лексему з рядка.
    while(*p) {
        q = tmp;     // Встановлюємо q для вказівки на початок масиву tmp.

        /* Зчитуємо символи доти, доки не трапиться або пропуск,
        або нульовий символ (ознака завершення рядка). */
        while(*p !=' ' && *p) {
            *q = *p; q++; p++;
        }
        if(*p) p++;   // Переміщаємося за пропуск.
        *q = '\0';    // Завершуємо лексему нульовим символом.
        cout << tmp << endl;
    }

    getch(); return 0;
}
```

А ось як виглядає версія тієї ж самої програми з використанням індексування елементів масивів.

**Код програми 6.6. Демонстрація механізму поділу рядка на слова з використанням індексування елементів масивів**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[80], tmp[80];

    cout << "Введіть речення: "; gets(str);

    // Зчитуємо лексему з рядка.
    for(int i=0;; i++) {
        // Зчитуємо символи доти, доки не трапиться пропуск,
        // або нульовий символ (ознака завершення рядка).
        for(int j=0; str[i] !=' ' && str[i]; j++, i++)
            tmp[j] = str[i];
        tmp[j] = '\0'; // Завершуємо лексему нульовим символом.
        cout << tmp << endl;
        if(!str[i]) break;
    }

    getch(); return 0;
}
```

У цих програм може бути різна швидкодія, що зумовлено особливостями генерування коду програми C++-компіляторами. Як правило, під час використання індексування елементів масивів генерується довший код (з великою кількістю машинних команд), ніж це робиться у процесі виконання

арифметичних дій над покажчиками. Тому не дивно, що професійно в написаному C++-коді програми частіше трапляються версії, які орієнтуються на оброблення покажчиків. Але, якщо Ви – програміст-початківець, то сміливо використовуйте індексування елементів масивів, доки не навчитеся вільно володіти покажчиками.

### 6.4.2. Механізм індексування покажчика

Як було показано вище, можна отримати доступ до масиву, використовуючи арифметичні дії над покажчиками. Цікаво, що у мові програмування C++ покажчик, який посилається на масив, можна індексувати так, як би це було ім'я масиву (це свідчить про тісний зв'язок між покажчиками і масивами). Відповідний такому підходу синтаксис забезпечує альтернативу арифметичним операціям над покажчиками, оскільки він є дещо зручнішим у деяких ситуаціях. Розглянемо конкретний приклад.

**Код програми 6.7. Демонстрація механізму індексування покажчика подібно до індексування масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <cctype>            // Робота з символьними аргументами
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char str[20] = "Я тебе люблю";
    char *p;

    p = str;             // Індексуємо покажчик.

    for(int i=0; p[i]; i++) p[i] = toupper(p[i]); // Повертає прописні символи
    cout << p;           // Відображаємо рядок.

    getch(); return 0;
}
```

У процесі виконання програма відобразить на екрані таке:

```
Я тебе люблю
```

Ось як працює ця програма. Спочатку в масив str вводиться рядок " Я тебе люблю". Потім адреса початку цього рядка присвоюється покажчику p. Після цього кожен символ рядка str за допомогою функції **toupper**() перетвориться в його прописний еквівалент за допомогою індексування покажчика p. Пам'ятайте, що вираз p[i] за своєю дією однаковий виразу *(p+i).

### 6.4.3. Взаємозамінність покажчиків і масивів

Вище було показано, що покажчики і масиви дуже тісно пов'язані. І дійсно, у багатьох випадках вони взаємозамінні. Наприклад, за допомогою покажчика, який містить адресу початку масиву, можна отримати доступ до





елементів цього масиву або за допомогою арифметичних дій над покажчиком, або за допомогою індексування елементів масиву. Проте в загальному випадку покажчики і масиви не є взаємозамінними. Розглянемо, наприклад, такий фрагмент коду програми:

```
int num[10];

for(int i=0; i<10; i++) {
    *num = i;  // Тут все гаразд.
    num++;     // Помилка – змінну num модифікувати не можна.
}
```

Тут використовується масив цілочисельних значень з іменем num. Як зазначено в коментарі, незважаючи на те, що абсолютно прийнятно застосувати до імені num оператор "*" (який зазвичай застосовується до покажчиків), проте абсолютно неприпустимо модифікувати значення num. Йдеться про те, що num – константа, яка вказує на початок масиву. І її, як наслідок, інкрементувати ніяк не можна. Іншими словами, хоча ім'я масиву (без індексу) дійсно генерує покажчик на початок масиву, однак його значення зміні не підлягає.

Хоча ім'я масиву генерує константу-покажчик, його, проте, (подібно до покажчиків) можна помістити у вирази, якщо, звичайно, воно при цьому не модифікується. Наприклад, наступна настанова, у процесі виконання якої елементу num[3] присвоюється значення 100, є цілком допустимою:

```
*(num+3) = 100; // Тут все гаразд оскільки num не змінюється.
```

### 6.4.4. Масиви покажчиків

Покажчики, подібно до інших типів даних, можуть зберігатися в масивах. Ось, наприклад, як виглядає оголошення 10-елементного масиву покажчиків на **int**-значення.

```
int *Array[10];
```

У цьому записі кожен елемент масиву Array містить покажчик на цілочисельне значення.

Щоб присвоїти адресу **int**-змінній з іменем var третьому елементу цього масиву покажчиків, записується таке:

```
Array[2] = &var;
```

Необхідно пам'ятати, що тут Array – масив покажчиків на цілочисельні значення. Елементи цього масиву можуть містити тільки значення, які є адресами змінних цілого типу. Ось тому змінна var передує оператору "&".

Щоб присвоїти значення змінної var цілочисельній змінній x за допомогою масиву Array, використовують такий синтаксис:

```
x = *Array[2];
```

Оскільки адреса змінної var зберігається в елементі Array[2], то застосування оператора "*" до цієї індексованої змінної дасть змогу набути значення змінній var.

Подібно до інших масивів, масиви покажчиків можна ініціалізувати. Як правило, масиви ініціалізованих покажчиків використовують для зберігання

покажчиків на рядки. Наприклад, щоб створити функцію, яка виводить щасливі передбачення, можна таким чином визначити масив **fortunes**:

```
char *fortunes[] = {
    "Незабаром гроші потечуть до Вас рікою.\n" ,
    "Ваше життя осяє нове кохання.\n" ,
    "Ви житимете довго і щасливо.\n" ,
    "Гроші, вкладені зараз в справу, незабаром принесуть дохід.\n" ,
    "Близький друг шукатиме Вашої підтримки.\n"
};
```

Не забувайте, що мова програмування C++ забезпечує зберігання всіх рядкових літералів у таблиці рядків, пов'язаній з конкретною програмою, тому масив потрібен тільки для зберігання покажчиків на ці рядки. Таким чином, для виведення другого повідомлення достатньо використовувати настанову, подібну до такої:

```
cout << fortunes[1];
```

Наведений нижче код програми передбачень є цілком коректною. Для отримання випадкових чисел у ній використовується функція **rand**(), а для отримання випадкових чисел в діапазоні від 0 до 4 – оператор ділення за модулем, оскільки саме такі числа можуть слугувати для доступу до елементів масиву за індексом.

**Код програми 6.8. Демонстрація механізму використання масиву покажчиків**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

char *fortunes[] = {
    "Незабаром гроші потечуть до Вас рікою.\n" ,
    "Ваше життя осяє нове кохання.\n" ,
    "Ви житимете довго і щасливо.\n" ,
    "Гроші, вкладені зараз в справу, незабаром принесуть дохід.\n" ,
    "Близький друг шукатиме Вашої підтримки.\n"
};

int main()
{
    int chance;
    cout << "Щоб дізнатися про свою долю, натисніть будь-яку клавішу: ";

    // Рандомізуємо генератор випадкових чисел.
    while(!kbhit()) rand();
    cout << endl;

    chance = rand();
    chance = chance % 5;
```





```
    cout << fortunes[chance];

    getch(); return 0;
}
```

Зверніть увагу на цикл **while**, який викликає функцію **rand**() доти, доки не буде натиснуто на яку-небудь клавішу. Оскільки функція **rand**() завжди генерує одну і ту саму послідовність випадкових чисел, важливо мати можливість програмно використовувати цю послідовність з певної довільної позиції[1]. Ефект випадковості досягається за рахунок повторних звернень до функції **rand**(). Коли користувач натисне на клавішу, цикл зупиниться на деякій випадковій позиції послідовності чисел, що генеруються, і ця позиція визначить номер повідомлення, яке буде виведено на екран. Нагадаємо, що функція **kbhit**() є достатньо поширеним розширенням бібліотеки функцій мови програмування C++, яка забезпечується багатьма компіляторами, але не входить в стандартний пакет бібліотечних функцій мови C++.

У наведеному нижче прикладі використовується двовимірний масив покажчиків для розроблення програми, яка відображає синтаксис-пам'ятку за ключовими словами мови програмування C++. У програмі ініціалізується перелік покажчиків на рядки. Перша розмірність масиву призначена для вказівки на ключові слова мови програмування C++, а друга – на короткий їх опис. Перелік завершується двома нульовими рядками, які використовуються як ознака кінця списку. Користувач вводить ключове слово, а програма повинна вивести на екран його опис. Як бачимо, цей перелік містить всього декілька ключових слів. Тому його продовження залишається за Вами.

**Код програми 6.9. Демонстрація механізму відображення синтаксис-пам'ятки за ключовими словами мови C++**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

char *keyword[][2] = {
    "for", "for(ініціалізація; умова; інкремент)",
    "if", "if(умова)... else ...",
    "switch", "switch(значення) { case-перелік }",
    "while", "while(умова)...",
    // Сюди потрібно додати решту ключових слів мови програмування C++.
    "", "" // Перелік повинен завершуватися нульовими рядками.
};

int main()
{
    char str[80];

    cout << "Введіть ключове слово: "; cin >> str;
```

[1] Інакше кожного разу після запуску програма видаватиме один і той самий "прогноз".

```
        // Відображаємо синтаксис.
        for(int i=0; *keyword[i][0]; i++)
            if(!strcmp(keyword[i][0], str)) cout << keyword[i][1];

        getch(); return 0;
}
```

Ось приклад виконання цієї програми:

```
Введіть ключове слово: for
for(ініціалізація; умова; інкремент)
```

У цьому коді програми зверніть увагу на керівний вираз, організований оператором циклу **for**. Він призводить до завершення циклу, коли елемент **keyword**[i][0] містить покажчик на нуль, який інтерпретується як значення ФАЛЬШ. Отже, цикл зупиняється тоді, коли трапляється нульовий рядок, який завершує масив покажчиків.

### 6.4.5. Покажчики і рядкові літерали

Можливо, Вас здивує спосіб оброблення C++-компіляторами рядкових літералів, подібних до такого:

```
cout << strlenf' C++-компілятор");
```

Якщо C++-компілятор виявляє рядковий літерал, то він зберігає його в *таблиці рядків* програми і генерує покажчик на потрібний рядок. Тому наведений нижче код програми є абсолютно коректною, у процесі виконання якої на екран виводиться фраза "Робота з покажчиками – суцільне задоволення!".

**Код програми 6.10. Демонстрація механізму роботи з рядковими літералами**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char *s;

    s = "Робота з покажчиками – суцільне задоволення!";
    cout << s;
    getch(); return 0;
}
```

У процесі виконання цієї програми символи, які утворюють рядковий літерал, зберігаються в таблиці рядків, а змінній s присвоюється покажчик на відповідний рядок у цій таблиці.

***Таблиця рядків*** *– таблиця, що згенерується компілятором для зберігання рядків, що використовуються у програмі.*





Оскільки покажчик на таблицю рядків конкретної програми під час використання рядкового літерала генерується автоматично, то можна спробувати використовувати цей факт для модифікації вмісту цієї таблиці. Проте таке рішення навряд чи можна назвати вдалим. Йдеться про те, що C++-компілятори створюють оптимізовані таблиці, у яких один рядковий літерал може використовуватися в двох (або більше) різних місцях програми. Тому "насильницька" зміна рядка може викликати небажані побічні ефекти. Понад це, рядкові літерали є константами, і деякі сучасні C++-компілятори просто не дають змоги змінювати їх вміст. А під час спроби зробити це C++-компілятор згенерує помилку тривалості виконання.

### 6.4.5. Приклад порівняння покажчиків

Вище ми вже наголошували на тому, що значення одного покажчика можна порівнювати з іншим. Але, щоб порівняння покажчиків мало сенс, порівнювані покажчики мають бути якимсь чином пов'язані один з одним. Найчастіше такий зв'язок встановлюється у разі, коли обидва покажчики вказують на елементи одного і того ж масиву. Наприклад, дано два покажчики (з іменами A і B), які посилаються на один і той самий масив. Якщо A менше ніж B, значить, покажчик A вказує на елемент, індекс якого менше від індексу елемента, яка адресується покажчиком B. Таке порівняння особливо корисне для визначення граничних умов.

Порівняння покажчиків продемонстровано у наведеному нижче коді програми, у якій створюються дві змінні типу покажчика. Одна (з іменем start) спочатку вказує на початок масиву, а друга (з іменем end) – на його кінець. У міру введення користувачем чисел масив послідовно заповнюється від початку до кінця. Кожного разу, коли в масив вводиться чергове число, покажчик start інкрементується. Щоб визначити, чи заповнився масив, у програмі просто порівнюються значення покажчиків start і end. Коли start перевищить end, то масив буде заповнений "повністю". Програмі залишиться тільки вивести вміст заповненого масиву на екран.

**Код програми 6.11. Демонстрація механізму порівняння покажчиків**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int num[10], *start, *end;

    start = num;
    end = &num[9];

    while(start <= end) {
        cout << "Введіть число: "; cin >> *start; start++;
    }
```

```
        start = num; // Відновлення початкового значення покажчика
        while(start <= end) {
                cout << *start << " "; start++;
        }

        getch(); return 0;
}
```

Як показано у цьому коді програми, оскільки start і end обидва вказують на спільний об'єкт (у цьому випадку ним є масив num), то їх порівняння може мати сенс. Подібне порівняння часто використовують у професійно написаному C++-коді програми.

## 6.5. Механізм ініціалізації покажчиків

Після того, як покажчик був оголошений, але до того, як йому було присвоєне якесь значення, покажчик містить невідоме значення. Спроба використати покажчик до присвоєння йому якогось значення є неприємною помилкою, оскільки вона може порушити роботу не тільки Вашої програми, але й операційної системи. Навіть якщо цього не сталося, результат роботи програми буде неправильним і знайти цю помилку буде достатньо складно.

Вважають, що покажчик, який вказує в "нікуди", має мати значення null, однак і це не робить його "безпечним". Після того, як він потрапить в праву або ліву частину оператора присвоєння, то він знову може стати "небезпечним". З іншого боку нульовий покажчик можна використати, наприклад, для позначення кінця масиву покажчиків.

Якщо була спроба присвоїти яке-небудь значення тому, на що вказує покажчик з нульовим значенням, система видає попередження, що з'являється під час роботи програми (або після завершення роботи програми) "Null pointer assignment". Поява цього повідомлення є мотивом для пошуку використання неініціалізованого покажчика в програмі.

### 6.5.1. Домовленість про використання нульових покажчиків

Оголошений, але не ініціалізований покажчик міститиме довільне значення. Під час спроби використати покажчик до присвоєння йому конкретного значення можна зруйнувати не тільки власну програму, але навіть і операційну систему (найгірший, треба сказати, тип помилки!). Оскільки не існує гарантованого способу уникнути використання неініціалізованого покажчика, то C++-програмісти прийняли процедуру, яка дає змогу уникати таких жахливих помилок. За домовленістю, якщо покажчик містить нульове значення, то вважається, що він ні на що не посилається. Це означає, що у випадку присвоєння нульового значення всім невживаним покажчикам і уникати його використання, то можна уникнути випадкового використання неініціалізованого покажчика. Рекомендуємо і Вам дотримуватися цієї практики програмування.





Під час оголошення покажчик будь-якого типу можна ініціалізувати нульовим значенням, наприклад, як це робиться в такій настанові.

```
float *p = 0; // p -- тепер нульовий покажчик.
```

Для тестування покажчика використовують настанову **if**(будь-який з таких її варіантів):

```
if(p)     // Виконуємо щось, якщо p -- не нульовий покажчик.
if(!p)    // Виконуємо щось, якщо p -- нульовий покажчик.
```

Дотримуючи згаданої вище домовленості про нульові покажчики, Ви можете уникнути багатьох серйозних проблем, що виникають під час їх використання.

### 6.5.2. Покажчики і 16-розрядні середовища

На сьогодні більшість обчислювальних середовищ є 32-розрядними, однак раніше немало користувачів працювали в 16-розрядних (в основному, це DOS і Windows 3.1) і, природно, з 16-розрядним кодом. Ці операційні системи були розроблені для процесорів серії Intel 8086, які містять такі модифікації, як 80286, 80386, 80486 і Pentium (під час роботи в режимі емуляції процесора 8086). І хоча під час написання нового коду програми програмісти, як правило, орієнтуються на використання 32-розрядного середовища виконання, однак раніше створені програми підтримують компактні 16-розрядні середовища. Позаяк деякі теми актуальні тільки для 16-розрядних середовищ, то програмістам, які працюють в них, буде корисно отримати інформацію про те, як адаптувати "старий" програмний код до нового середовища, тобто переорієнтовувати 16-розрядний код на 32-розрядний.

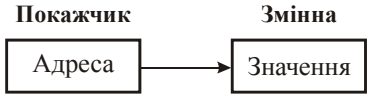
Під час написання 16-розрядного коду програми для процесорів серії Intel 8086 програміст має право розраховувати на шість способів компілювання програм, які відрізняються організацією комп'ютерної пам'яті. Програми можна компілювати для мініатюрної, малої, середньої, компактної, великої і величезної моделей пам'яті. Кожна з цих моделей по-своєму оптимізує простір, що резервується для даних, коду програми і стека. Відмінність в організації комп'ютерної пам'яті пояснюється використанням процесорами серії Intel 8086 сегментованої архітектури у процесі виконання 16-розрядного коду програми. У 16-розрядному сегментованому режимі процесори серії Intel 8086 ділять пам'ять на 16К сегментів.

У деяких випадках модель пам'яті може вплинути на поведінку покажчиків і Ваші можливості з їх використання. Основна проблема виникає під час інкрементування покажчика за межі сегмента. Розгляд особливостей кожної з 16-розрядних моделей пам'яті виходить за рамки цього навчального посібника. Головне, щоб Ви знали, що, коли Вам доведеться працювати в 16-розрядному середовищі і орієнтуватися на процесори серії Intel 8086, програміст повинен вивчити документацію, що додається до використовуваного Вами компілятора, і детально розібратися в моделях пам'яті та їх впливі на покажчики.

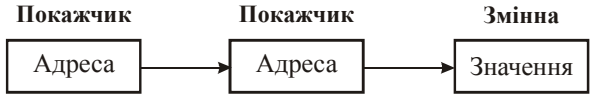
Під час написання програм для сучасного 32-розрядного середовища необхідно знати, що в ній використано єдину модель організації пам'яті, яка називається *однорівневою*, *несегментованою* або *лінійною* (flat model).

### 6.5.3. Багаторівнева непряма адресація

Можна створити покажчик, який посилатиметься на інший покажчик, а той – на кінцеве значення. Цю ситуацію називають *багаторівневою непрямою адресацією* (multiple indirection) або використанням *покажчика на покажчик*. Ідея багаторівневої непрямої адресації схематично проілюстрована на рис. 6.2. Як бачимо, значення звичайного покажчика (при однорівневій непрямій адресації) є адресою змінної, яка містить певне значення. У разі застосування покажчика на покажчик перший містить адресу другого, а той посилається на змінну, що містить певне значення.




**Однорівнева непряма адресація**




**Багаторівнева непряма адресація**

**Рис. 6.2. Однорівнева і багаторівнева непряма адресація**

Під час використання непрямої адресації можна організувати будь-яку бажану кількість рівнів, але, як правило, обмежуються тільки двома, оскільки збільшення їх кількості часто веде до виникнення концептуальних помилок.

Змінну, яка є покажчиком на покажчик, потрібно оголосити відповідним чином. Для цього достатньо перед її іменем поставити додатковий символ "зірочка"(*). Наприклад, таке оголошення повідомляє компіляторові про те, що balance – покажчик на покажчик на значення типу **int**:

```
int **balance;
```

Необхідно пам'ятати, що змінна balance тут – не покажчик на цілочисельне значення, а покажчик на покажчик на **int**-значення.

Щоб отримати доступ до значення, яке адресується покажчиком на покажчик, необхідно двічі застосувати оператор "*", як це показано в такому короткому прикладі.

**Код програми 6.12. Демонстрація механізму використання багаторівневої непрямої адресації**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
```





```
{
    int x, *p, **q;
    x = 10; p = &x; q = &p;
    cout << **q;              // Виводимо значення змінної x.
    getch(); return 0;
}
```

У цьому коді програми змінна p оголошена як покажчик на **int**-значення, а змінна q – як покажчик на покажчик на **int**-значення. У процесі виконання цієї програми ми набудемо значення змінної x, тобто число 10.

## 6.6. Виникнення проблем під час використання покажчиків

Для програміста немає нічого страшнішого, ніж покажчики, що "збунтувалися"! Покажчики можна порівняти з енергією атома: вони водночас і надзвичайно корисні та страшенно небезпечні. Якщо виникла проблема, пов'язана з отриманням покажчиком неправильного значення, то таку помилку відшукати найважче.

Труднощі виявлення помилок, пов'язаних з покажчиками, пояснюються тим, що сам по собі покажчик не виявляє проблему. Проблема може виявитися тільки опосередковано, можливо, навіть внаслідок виконання декількох настанов після "крамольної" операції з покажчиком. Наприклад, якщо один покажчик випадково отримає адресу "не тих" даних, то у процесі виконання операції з цим "сумнівним" покажчиком дані, що адресуються, можуть піддатися небажаній зміні, і, що тут найнеприємніше, то це те, що ця "таємна" зміна, як правило, проявляється набагато пізніше. Таке "запізнювання" істотно ускладнює пошук помилки, оскільки посилає Вас по "помилковому сліду". До того моменту, коли проблема стане очевидною, цілком можливо, що покажчик-винуватець зовні виглядатиме "нешкідливою овечкою", і Вам доведеться витратити ще немало часу, щоб знайти дійсну причину проблеми.

Оскільки для багатьох працювати з покажчиками – означає потенційно приректи себе на пошуки відповіді на запитання "Хто винен?", ми спробуємо розглянути можливі "яри" на шляху відважного програміста і показати деякі обхідні шляхи, що дають змогу уникнути виснажливих "мук творчості".

### 6.6.1. Поняття про неініціалізовані покажчики

Класичний приклад помилки, що допускається під час роботи з покажчиками, – використання неініціалізованого покажчика. Розглянемо такий фрагмент коду програми:

```
// Ця програма некоректна
int main()
{
    int x = 10, *p;
    *p = x;              // На що вказує змінна p?
    getch(); return 0;
}
```

У цьому записі покажчик p містить невідому адресу, оскільки його ніде не визначено. У Вас немає можливості дізнатися, де записано значення змінної x. При невеликих розмірах програми (наприклад, як у цьому випадку) її особливості (які полягають у тому, що покажчик p містить адресу, що не належить ні коду програми, ні області даних) можуть ніяк не виявлятися, тобто програма зовні працюватиме нормально. Але у міру її удосконалення і, відповідно, збільшення її обсягу, імовірність того, що p стане вказувати або на код програми, або на область даних, зросте. Врешті-решт колись раптово програма взагалі перестане працювати. Спосіб не допустити розроблення таких програм очевидний: перш ніж використовувати покажчик, потурбуйтеся про те, щоб він посилався на що-небудь дійсне!

### 6.6.2. Некоректне порівняння покажчиків

Порівняння покажчиків, які не посилаються на елементи одного і того ж масиву, в загальному випадку є некоректним і часто призводить до виникнення помилок. Ніколи не варто покладатися на те, що різні об'єкти будуть розміщені в пам'яті якимсь певним чином (десь поряд) або на те, що всі компілятори і операційні середовища оброблятимуть Ваші дані однаково. Тому будь-яке порівняння покажчиків, які посилаються на різні об'єкти, може призвести до несподіваних наслідків. Розглянемо такий приклад:

```
char str[80], ctr[80];
char *p1, *p2;

p1 = str;
p2 = ctr;
if(p1 < p2).
```

У цьому записі використовується некоректне порівняння покажчиків, оскільки мова програмування C++ не дає ніяких гарантій щодо розміщення змінних у пам'яті. Ваш програмний код повинен бути написаний так, щоб він працював однаково стійко незалежно від того, де розташовані дані в пам'яті.

Було б помилкою передбачати, що два оголошені масиви будуть розташовані в пам'яті поруч, і тому можна звертатися до них шляхом індексування їх за допомогою одного і того ж покажчика. Припущення про те, що інкрементований покажчик після виходу за межі першого масиву стане посилатися на другий, абсолютно ні на чому не обґрунтоване і тому є неправильним. Розглянемо уважно такий приклад:

```
int first[10], second[10];

int *p = first;
for(int t=0; t<20; ++t) {
        *p = t; p++;
}
```

Мета цієї програми – ініціалізувати елементи масивів first і second числами від 0 до 19. Проте цей програмний код не дає змоги сподіватися на досягнення бажаного результату, хоча у деяких умовах і під час використання





певних компіляторів ця програма працюватиме так, як задумав автор. Не варто покладатися також на те, що масиви first і second будуть розташовані в пам'яті комп'ютера послідовно, причому першим обов'язково буде масив first. Мова програмування C++ не гарантує певного розташування об'єктів у пам'яті, тому цю програму не можна вважати коректною.

### 6.6.3. Не встановлення покажчиків

Така (некоректна) програма повинна прийняти рядок, введений з клавіатури, а потім відобразити ASCII-код для кожного символу цього рядка[1]. Проте ця програма містить серйозну помилку.

**Код програми 6.13. Демонстрація роботи некоректної програми**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
        char str[80];
        char *p1;

        p1 = str;
        do {
                cout << "Введіть рядок: "; gets(p1); // Зчитуємо рядок.

                // Виводимо ASCII-значення кожного символу.
                while(*p1) cout << (int) *p1++ " ";
                cout << endl;

        } while(strcmp(str, "Кінець програми"));

        getch(); return 0;
}
```

Чи зможете Ви самі знайти тут помилку?

У наведеному вище варіанті програми покажчику p1 присвоюється адреса масиву str тільки один раз. Це присвоєння здійснюється поза циклом. Під час входу в **do-while**-цикл (тобто при першій його ітерації) p1 дійсно вказує на перший символ масиву str. Але під час другого проходу того ж циклу p1 покажчик міститиме значення, яке залишиться після виконання попередньої ітерації циклу, оскільки покажчик p1 не встановлюється заново на початок масиву str. Рано чи пізно межу масиву str буде порушено.

Ось як виглядає коректний варіант тієї ж самої програми.

[1] Звернемо Вашу увагу на те, що для виведення ASCII-кодів на екран використовується операція приведення типів.

**Код програми 6.14. Демонстрація механізму роботи коректної програми**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdio>            // Підтримка системи введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      char str[80];
      char *p1;

      do {
            p1 = str; // Встановлюємо p1 при кожній ітерації циклу.
            cout << "Введіть рядок: "; gets(p1); // Зчитуємо рядок.

            // Виводимо ASCII-значення кожного символу.
            while(*p1) cout << (int) *p1++ " ";
            cout << endl;

      } while(strcmp(str, "Кінець програми"));

      getch(); return 0;
}
```

Отже, у цьому варіанті програми на початку кожної ітерації циклу покажчик p1 встановлюється на початок рядка.

***Нео! Відновлювати пам'ять!*** *Щоб використання покажчиків було безпечним, потрібно у будь-який момент знати, на що вони посилаються.*





# Розділ 7. ОСОБЛИВОСТІ ЗАСТОСУВАННЯ C++-ФУНКЦІЙ

У цьому розділі поглиблено розглянуто функції – будівельні блоки мови програмування C++, а тому без повного їх розуміння неможливо стати успішним C++-програмістом. Ми вже торкнулися теми, яка стосується C++-функцій (див. розд. 2.3), і використали їх майже у кожному прикладі коду програми. У цьому розділі вони будуть розглядатися детальніше. Ця тема містить такі питання, як перегляд правил дії областей видимості функцій, рекурсивних функцій, деяких спеціальних властивостей функції **main**(), настанови **return** і прототипів функцій.

## 7.1. Правила дії областей видимості функцій

*Правила дії областей видимості* будь-якої мови програмування – правила, які дають змогу керувати доступом до об'єкта з різних частин програми. Іншими словами, правила дії областей видимості визначають, який програмний код має доступ до тієї або іншої змінної. Ці правила також визначають тривалість "життя" змінної. Як було зазначено вище, існує три види змінних: локальні змінні, формальні параметри і глобальні змінні. Тут ми розглянемо правила дії областей видимості з огляду на використання функцій.

*Правила дії областей видимості функцій визначають можливість отримання доступу до об'єкта і тривалість його існування.*

### 7.1.1. Локальні змінні

Як зазначалося вище, змінні, які оголошено всередині функції, називаються *локальними*. Але у мові програмування C++ передбачено "уважніше" відношення до локальних змінних, ніж ми могли помітити дотепер. У мові програмування C++ змінні можуть бути внесені в блоки. Це означає, що змінну можна оголосити усередині будь-якого блоку коду програми, після чого вона стане локальною змінною стосовно цього блоку[1]. Насправді змінні, локальні стосовно функції, утворюють просто спеціальний випадок більш загальної ідеї.

Локальну змінну можуть використовувати тільки настанови, внесені в блок, у якому ця змінна оголошена. Іншими словами, локальна змінна невідома за межами власного блоку коду програми. Отже, записані поза блоком настанови не можуть отримати доступ до об'єкта, який визначається усередині блоку.

[1] Необхідно пам'ятати, що блок починається з відкриваючої фігурної дужки і завершується такою, що його закриває.

Важливо розуміти, що локальні змінні існують тільки у процесі виконання програмного блоку, у якому вони оголошені. Це означає, що локальна змінна створюється під час входу в "свій" блок і руйнується при виході з нього. А оскільки локальна змінна руйнується при виході зі "свого" блоку, її значення втрачається.

Найпоширенішим програмним блоком є функція. У мові програмування C++ кожна функція визначає блок коду програми, який починається з відкритої фігурної дужки цієї функції та завершується її закритою фігурною дужкою. Код функції та її дані – її "приватна власність", і до неї не може отримати доступ жодна настанова з будь-якої іншої функції, за винятком настанови її виклику[1]. Тіло функції надійно приховане від решти частини програми і, якщо у функції не використовуються глобальні змінні, то вона не може зробити ніякого впливу на інші частини програми, однаково, як і ті на неї. Таким чином, вміст однієї функції зовсім незалежний від вмісту іншої. Іншими словами, програмні коди і дані, визначені в одній функції, не можуть взаємодіяти з кодами і даними, визначеними в іншій, оскільки дві функції мають різні області видимості.

Оскільки кожна функція визначає власну область видимості, змінні, які оголошено в одній функції, не роблять ніякого впливу на змінні, які оголошено в іншій, причому навіть у тому випадку, якщо ці змінні мають однакові імена. Розглянемо, наприклад, таку програму:

**Код програми 7.1. Демонстрація механізму використання області видимості локальних змінних**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void Fun();

int main()
{
        char str[] = "Це масив str у функції main().";
        cout << str << endl;
        Fun();
        cout << str << endl;

        getch(); return 0;
}

void Fun()
{
        char str[80];
        cout << "Введіть будь-який рядок: "; cin >> str;
        cout << str << endl;
}
```

[1] Наприклад, неможливо використовувати настанову `goto` для переходу в середину коду іншої функції.





Символьний масив str оголошується тут двічі: перший раз у функції **main**() і ще раз у функції Fun(). При цьому масив str, оголошений у функції **main**(), не має жодного стосунку до однойменного масиву з функції Fun(). Як пояснювалося вище, кожен масив (у цьому випадку str) відомий тільки блоку коду програми, у якому він оголошений. Щоб переконатися у цьому, достатньо виконати наведену вище програму. Як бачите, хоча масив str отримує рядок, що вводиться користувачем у процесі виконання функції Fun(), вміст масиву str у функції **main**() залишається незмінним.

Мова C++ містить ключове слово **auto**, яке можна використовувати для оголошення локальних змінних. Але оскільки всі не глобальні змінні є за замовчуванням **auto**-змінними, то до цього ключового слова практично ніколи не вдаються. Тому Ви не знайдете у цьому посібнику жодного прикладу з його використанням. Але, якщо Ви захочете все-таки застосувати його у своїй програмі, то знайте, що розміщувати його потрібно безпосередньо перед типом змінної, як це показано далі:

```
auto char ch;
```

Звичайною практикою є оголошення всіх змінних, що використовуються у функції, на початку програмного блоку цієї функції. У цьому випадку всякий, кому доведеться розбиратися в коді цієї функції, легко дізнається, які змінні в ній використовуються. Проте початок блоку функції – не єдине можливе місце для оголошення локальних змінних. Локальні змінні можна оголошувати в будь-якому місці блоку коду програми. Змінна, оголошена в блоці, є локальною стосовно цього блоку. Це означає, що така змінна не існує доти, доки не буде виконаний вхід в блок, а руйнування такої змінної відбувається при виході з її блоку. При цьому ніякий код поза цим блоком не може отримати доступ до цієї змінної (навіть програмний код, який належить тій самій функції).

Щоб краще зрозуміти зазначене вище, розглянемо таку програму:

**Код програми 7.2. Демонстрація механізму використання області видимості локальних змінних стосовно блоку**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

int main()
{
        int vybir;
        cout << "(1) додати числа або ";
        cout << "(2) конкатенувати рядки?: "; cin >> vybir;
        if(vybir == 1) {
                int a, b; // Активізуються дві int-змінні.
                cout << "Введіть два числа: "; cin >> a >> b;
                cout << "Сума дорівнює " << a + b << endl;
        }
        else {
                char s1[80], s2[80]; // Активізуються два рядки.
```

```
        cout << "Введіть два рядки: "; cin >> s1; cin >> s2;
        strcpy(s1, s2);
        cout << "Конкатенація дорівнює " << s1 << endl;
    }
    getch(); return 0;
}
```

Ця програма, залежно від вибору користувача, забезпечує введення двох чисел або двох рядків. Зверніть увагу на оголошення змінних a і b в **if**-блоці і змінних s1 і s2 у **else**-блоці. Існування цих змінних почнеться з моменту входу у відповідний блок і припиниться відразу після виходу з нього. Якщо користувач вибере додавання чисел, будуть створені змінні a і b, а якщо він захоче конкатенувати рядки – то змінні s1 і s2. Нарешті, ні до однієї з цих змінних не можна звернутися ззовні їх блоку, навіть з частини коду програми, що належить тій самій функції. Наприклад, якщо Ви спробуєте скомпілювати наступну (некоректну) версію програми, то отримаєте повідомлення про помилку.

**Код програми 7.2. Демонстрація механізму роботи некоректного звернення до локальних змінних**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    int vybir;

    cout << "(1) додати числа або ";
    cout << "(2) конкатенувати рядки?: "; cin >> vybir;
    if(vybir == 1) {
        int a, b; // Активізуються дві int-змінні.
        cout << "Введіть два числа: "; cin >> a >> b;
        cout << "Сума дорівнює " << a + b << endl;
    }
    else {
        char s1[80], s2[80]; /* Активізуються два рядки. */
        cout << "Введіть два рядки: "; cin >> s1; cin >> s2;
        strcpy(s1, s2);
        cout << "Конкатенація дорівнює " << s1 << endl;
    }
    a = 10; // *** Помилка ***. Змінна a тут невідома!

    getch(); return 0;
}
```

Оскільки у цьому випадку змінна a невідома поза своїм **if**-блоком, компілятор видасть помилку під час спроби її використовувати.

Якщо ім'я змінної, оголошеної у внутрішньому блоці, збігається з іменем змінної, оголошеної в зовнішньому блоці, то "внутрішня" змінна перевизначає "зовнішню" у межах області видимості внутрішнього блоку. Розглянемо такий приклад.





**Код програми 7.3. Демонстрація механізму перевизначення "зовнішньої" змінної на "внутрішню"**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int c = 10, d = 100;

    if(d > 0) {
        int c;          // Ця змінна c відокремлена від зовнішньої змінної c.
        c = d / 2;
        cout << "Внутрішня змінна c: " << c << endl;
    }
    cout << "Зовнішня змінна c: " << c << endl;

    getch(); return 0;
}
```

Ось як виглядають результати виконання цієї програми.

```
Внутрішня змінна c: 50
Зовнішня змінна c: 10
```

Тут змінна c, оголошена усередині **if**-блоку, перевизначає або приховує зовнішню змінну c. Зміни, яким піддалася внутрішня змінна c, не роблять ніякого впливу на зовнішню змінну c. Понад це, поза **if**-блоком внутрішня змінна c більше не існує, і тому зовнішня змінна c знову стає видимою.

Оскільки локальні змінні створюються з кожним входом і руйнуються з кожним виходом з програмного блоку, у якому вони оголошені, то вони не зберігають своїх значень між активізаціями блоків. Це особливо важливо пам'ятати стосовно функцій. Під час виклику функції її локальні змінні створюються, а при виході з неї руйнуються. Це означає, що локальні змінні не зберігають своїх значень між викликами функцій[1].

*Локальні змінні не зберігають своїх значень між активізаціями.*

Локальні змінні зберігаються в стеку, якщо при цьому не задано іншого способу зберігання. Оскільки стек – динамічно змінна область пам'яті, локальні змінні не можуть в загальному випадку зберігати свої значення між викликами функцій.

Як було зазначено вище, незважаючи на те, що локальні змінні зазвичай оголошуються на початку свого блоку, це не є обов'язковим. Локальні змінні можна оголосити в будь-якому місці блоку, головне, щоб це було зроблено до їх використання. Наприклад, наведений нижче код програми цілком допустимий.

[1] Існує один спосіб обійти це обмеження, який буде розглядатися нижче у цій книзі.

**Код програми 7.4. Демонстрація механізму оголошення змінних**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int a;                   // Оголошуємо одну змінну.
    cout << "Введіть число: "; cin >> a;

    int b;                   // Оголошуємо ще одну змінну.
    cout << "Введіть друге число: "; cin >> b;

    cout << "Добуток дорівнює: " << a*b << endl;

    getch(); return 0;
}
```

У наведеному прикладі змінні a і b не оголошуються доти, доки вони стануть потрібними. Все ж таки більшість програмістів оголошують усі локальні змінні на початку блоку, у якому вони використовуються, але це, як мовиться, питання стилістики (або смаку).

### 7.1.2. Оголошення змінних у ітераційних настановах і настановах вибору

Змінну можна оголосити в розділі ініціалізації циклу **for** або умовних виразах таких настанов як **if**, **switch** або **while**. Змінна, оголошена в одній з цих настанов, має область видимості, яка обмежена блоком коду програми, керованим цією настановою. Наприклад, змінна, оголошена в настанові організації циклу **for**, буде локальною для цього циклу, як це показано в наведеному нижче прикладі:

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    // Змінна i локальна для циклу for.
    for(int i=0; i<10; i++) {
        cout << i << " ";
        cout << "у квадраті дорівнює " << i * i << endl;
    }
    // i = 10; // *** Помилка *** -- i тут невідома!
    getch(); return 0;
}
```

У цьому коді програми змінну i оголошують в розділі ініціалізації циклу **for** і використовують для керування цим циклом. А за межами циклу змінна i невідома.

У загальному випадку, якщо керівна змінна циклу **for** не потрібна за межами цього циклу, то оголошення її усередині **for**-настанови (як показано у





наведеному прикладі) добре тим, що воно обмежує її існування рамками циклу і, тим самим, запобігає випадковому використанню у якомусь іншому місці програми. Професійні програмісти часто оголошують керівну змінну циклу усередині **for**-настанови. Але, якщо змінна потрібна коду програми поза циклом, її не можна оголошувати в настанові **for**.

***Нео! хідноапам'ятати!а*** *Твердження про те, що змінна, оголошена в розділі ініціалізації циклу* ***for****, є локальною стосовно цього циклу або не є такою, змінилося з часом (йдеться про час, протягом якого розвивалася мова C++). Спочатку така змінна була доступна після виходу з циклу* ***for****. Проте стандарт C++ обмежує область видимості цієї змінної рамками циклу* ***for****. Але необхідно мати на увазі, що різні компілятори і тепер порізному "дивляться" на цю ситуацію.*

Якщо Ваш компілятор повністю дотримується стандарту мови програмування C++, то Ви можете також оголосити змінну в умовному виразі настанов **if**, **switch** або **while**. Наприклад, у наведеному нижче коді програми

```
if(int x = 20) {
    cout << "Це значення змінної x: ";
    cout << x;
}
```

визначається тип змінної x, якій присвоюється число 20. Оскільки цей вираз оцінюється як істинний, настанова **cout** буде виконана. Область видимості змінних, оголошених в умовному виразі настанови, обмежується блоком коду програми, керованим цією настановою. Отже, у цьому випадку змінна x невідома за межами настанови **if**. Правду кажучи, далеко не всі програмісти вважають оголошення змінних в умовному виразі настанов ознакою хорошого стилю програмування, і тому такий прийом у цьому посібнику більше не повториться.

### 7.1.3. Формальні параметри

Як уже зазначалося вище, якщо функція використовує аргументи, то вона повинна оголосити змінні, які прийматимуть значення цих аргументів. Ці змінні називаються *формальними параметрами* функції. Якщо не рахувати отримання значень аргументів під час виклику функції, то поведінка формальних параметрів нічим не відрізняється від поведінки будь-яких інших локальних змінних усередині функції. Область видимості параметра обмежується рамками його функції. Програміст повинен гарантувати, що тип оголошуваних ним формальних параметрів збігається з типом аргументів, що передаються функції.

***Нео! хідноапам'ятати!*** *Незважаючи на те, що формальні параметри виконують спеціальне завдання отримання значень аргументів, їх можна використовувати подібно до будь-яких інших локальних змінних. Наприклад, параметру усередині функції можна присвоїти яке-небудь нове значення.*

### 7.1.4. Глобальні змінні

Глобальні змінні у багатьох аспектах протилежні до локальних. Вони відомі впродовж всієї програми, їх можна використовувати в будь-якому її місці, і вони зберігають свої значення у процесі виконання всього коду програми. Отже, їх область видимості розширюється до обсягу всієї програми. Глобальна змінна створюється шляхом її оголошення поза будь-якою функцією. Завдяки їх глобальності доступ до цих змінних можна отримати з будь-якого виразу, незалежно від функції, у якій цей вираз знаходиться.

Якщо глобальна і локальна змінні мають однакові імена, то перевага знаходиться на стороні локальної змінної. Іншими словами, локальна змінна приховає глобальну з таким самим іменем. Таким чином, незважаючи на те, що до глобальної змінної теоретично можна отримати доступ з будь-якого коду програми, практично це можливо тільки у випадку, якщо однойменна локальна змінна не перевизначить глобальну.

Використання глобальних змінних продемонстровано у наведеному нижче коді програми. Як бачимо, змінні count і num_right оголошені поза всіма функціями, отже, вони глобальні. Із звичайних практичних міркувань краще оголошувати глобальні змінні ближче до початку програми. Але формально вони просто мають бути оголошені до їх першого використання. Пропонована для перегляду програма – тільки простий тренажер з виконання арифметичного додавання. Спочатку користувачу запропоновано вказати кількість вправ. Для виконання кожної вправи викликається функція drill(), яка генерує два випадкові числа в діапазоні від 0 до 99. Користувачу пропонується додати ці числа, а потім перевіряється відповідь. На кожну вправу дається три спроби. В кінці програма відображає кількість правильних відповідей. Зверніть особливу увагу на глобальні змінні, що використовуються у цьому коді програми.

**Код програми 7.5. Демонстрація механізму роботи програми-тренажера з виконання операції додавання**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void drill();        // Попереднє оголошення функції
int count;           // Змінні count і num_right – глобальні.
int num_right;

int main()
{
      cout << "Скільки практичних вправ: "; cin >> count;

      num_right = 0;
      do {
            drill();
            count--;
      } while(count);
      cout << "Ви дали " << num_right << " правильних відповідей" << endl;

      getch(); return 0;
}

void drill()         // Визначення функції
{
      int count; /* Ця змінна count -- локальна і ніяк
                    не пов'язана з однойменною глобальною. */
      int a, b, ans;
      // Генеруємо два числа між 0 і 99.
      a = rand() % 100;
      b = rand() % 100;
      // Користувач отримує три спроби дати правильну відповідь.
      for(count = 0; count<3; count++) {
            cout << "Скільки буде " << a << " + " << b << "? "; cin >> ans;
      if(ans == a + b) {
            cout << "Правильно" << endl;
            num_right++;
            return;
      }
      }
      cout << "Ви використовували всі свої спроби" << endl;
      cout << "Відповідь дорівнює " << a + b << endl;
}
```

При уважному вивченні цієї програми Вам стане зрозуміло, що як функція **main**(), так і функція drill() отримують доступ до глобальної змінної num_right. Але із змінною count справа йде дещо складніше. У функції **main**() використовується глобальна змінна count. Проте у функції drill() оголошується локальна змінна count. Тому тут під час використання імені count маємо на увазі саме локальну, а не глобальну змінну count. Необхідно пам'ятати, що, коли у функції глобальна і локальна змінні мають однакові імена, то під час звернення до цього імені маємо на увазі локальну, а не глобальну змінну.

Зберігання глобальних змінних здійснюється в деякій певній області пам'яті, програмою, що спеціально виділяється для цих потреб. Глобальні змінні корисні у тому випадку, коли в декількох функціях програми використовуються одні і ті ж самі дані, або коли змінна повинна зберігати своє значення впродовж виконання всієї програми. Проте без особливої потреби необхідно уникати використання глобальних змінних, і на це є три причини:

- вони займають пам'ять протягом всієї тривалості виконання програми, а не тільки тоді, коли дійсно вони необхідні;
- використання глобальної змінної в "ролі", з якою легко б "справилася" локальна змінна, робить таку функцію менш універсальною, оскільки вона покладається на потребу визначення даних поза цією функцією;
- використання великої кількості глобальних змінних може призвести до появи помилок в роботі програми, оскільки при цьому можливий прояв невідомих і небажаних побічних ефектів.

Основна проблема, характерна для розроблення великих C++-програм, випадкове модифікування значення змінної у якомусь іншому місці програми. Чим більше глобальних змінних у програмі, тим більшою є ймовірність помилки.

## 7.2. Передача покажчиків і масивів як аргументів функціям

Дотепер у прикладах, наведених тут, функціям передавалися значення простих змінних. Але можливі ситуації, коли як аргументи необхідно використовувати покажчики і масиви. Вивченню особливостей передачі таких аргументів функцій і присвячено наступні підрозділи.

### 7.2.1. Виклик функцій з покажчиками

У мові програмування C++ дозволено передавати функції покажчики як аргументи. Для цього достатньо оголосити параметр типу покажчика. Розглянемо такий приклад.

**Код програми 7.6. Демонстрація механізму передачі функції покажчика: початкова версія**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Fun(int *d);        // Попереднє оголошення функції

int main()
{
      int c, *p;
      p = &c;            // Покажчик p тепер містить адресу змінної c.
      Fun(p);
      cout << c;         // Змінна c тепер містить число 100.

      getch(); return 0;
}

void Fun(int *d)         // Визначення функції
{
      *d = 100;  // Змінній, яка адресується покажчиком d, присвоюється число 100.
}
```

Як бачите, у цьому коді програми функція Fun() приймає один параметр: покажчик на цілочисельне значення. У функції **main**() покажчику p присвоюється адреса змінної c. Потім з функції **main**() викликається функція Fun(), а покажчик p передається їй як аргумент. Після того, як параметр-покажчик d набуде значення аргументу p, він (так само, як і p) указуватиме на змінну c, що визначається у функції **main**(). Таким чином, у процесі виконання операції присвоєння *d = 100; змінна c набуває значення 100. Тому програма відобразить на екрані число 100. У загальному випадку наведена тут функція Fun()





присвоює число 100 змінній, адреса якої була передана цій функції як аргумент.

У попередньому прикладі необов'язково було використовувати змінному p. Замість неї під час виклику функції Fun() достатньо використовувати змінну c, якій передував оператором "&" (при цьому, як уже зазначалося вище, генерується адреса змінної c). Після внесення зумовленої зміни попередня програма набуває такого вигляду:

**Код програми 7.7. Демонстрація механізму передачі покажчика функції: виправлена версія**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Fun(int *d);        // Попереднє оголошення функції

int main()
{
      int c;
      Fun(&c);
      cout << c << endl;

      getch(); return 0;
}

void Fun(int *d)         // Визначення функції
{
      *d = 100;  // Змінній, яка адресується покажчиком d, присвоюється число 100.
}
```

Передаючи покажчик функції, необхідно розуміти таке. У процесі виконання деякої операції у функції, яка використовує покажчик, ця операція фактично здійснюється над змінною, яка адресується цим покажчиком. Таким чином, така функція може змінити значення об'єкта, яке адресується її параметром.

### 7.2.2. Виклик функцій з масивами

Якщо масив є аргументом функції, то необхідно розуміти, що під час виклику такої функції їй передається тільки адреса першого елемента масиву, а не повна його копія[1]. Це означає, що оголошення параметра повинно мати тип, сумісний з типом аргумента. Взагалі існує три способи оголосити параметр, який приймає покажчик на масив. По-перше, параметр можна оголосити як масив, тип і розмір якого збігається з типом і розміром масиву, використовуваного під час виклику функції. Цей варіант оголошення параметра-масиву продемонстровано в наведеному нижче прикладі.

[1] Необхідно пам'ятати, що у мові програмування C++ ім'я масиву без індексу є покажчик на перший елемент цього масиву.

**Код програми 7.8. Демонстрація механізму оголошення параметра-масиву**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Display(int num[10]);   // Попереднє оголошення функції

int main()
{
      int t[10];
      for(int i=0; i<10; ++i) t[i] = i;
      Display(t);          // Передаємо функції масив t.
      getch(); return 0;
}

// Функція виводить усі елементи масиву.
void Display(int num[10])    // Визначення функції
{
      for(int i=0; i<10; i++) cout << num[i]<< endl;
}
```

Незважаючи на те, що параметр num оголошений тут як цілочисельний масив, який складається з 10 елементів, C++-компілятор автоматично перетворить його в покажчик на цілочисельне значення. Необхідність цього перетворення пояснюється тим, що ніякий параметр насправді не може прийняти масив цілком. А оскільки буде переданий один тільки покажчик на масив, то функція повинна мати параметр, здатний прийняти цей покажчик.

Другий спосіб оголошення параметра-масиву полягає в його представленні у вигляді безрозмірного масиву, як це показано далі:

```
void Display(int num[])
{
      for(int i=0; i<10; i++) cout << num[i] << endl;
}
```

У цьому записі параметр num оголошується як цілочисельний масив невідомого розміру. Оскільки мова C++ не забезпечує перевірки порушення меж масиву, то реальний розмір масиву не релевантний чинник для подібного параметра (але, безумовно, не для програми в цілому). Цілочисельний масив при такому способі оголошення також автоматично перетвориться C++-компілятором у покажчик на цілочисельне значення.

Нарешті, розглянемо третій спосіб оголошення параметра-масиву. При передачі масиву функції її параметр можна оголосити як покажчик. Якраз цей варіант найчастіше використовують професійні програмісти. Ось приклад:

```
void Display(int *num)
{
      for(int i=0; i<10; i++) cout << num[i]<< endl;
}
```





Можливість такого оголошення параметра (у цьому випадку num) пояснюється тим, що будь-який покажчик (подібно до масиву) можна індексувати за допомогою символів квадратних дужок "[]". Таким чином, всі три способи оголошення параметра-масиву приводяться до однакового результату, який можна виразити одним словом: покажчик.

Проте окремий *елемент* масиву, що використовується як аргумент, обробляється подібно до звичайної змінної. Наприклад, розглянуту вище програму можна було б переписати, не використовуючи передачу цілого масиву.

**Код програми 7.9. Демонстрація механізму оброблення елемента масиву, що використовується як аргумент, подібно до звичайної змінної**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Display(int num);       // Попереднє оголошення функції

int main()
{
      int Array[10], i;
      for(i=0; i<10; ++i) Array[i] =i;
      for(i=0; i<10; i++) Display(Array[i]);
      getch(); return 0;
}

// Функція виводить одне число.
void Display(int num)        // Визначення функції
{
      cout << num << endl;
}
```

Як бачите, параметр, який використовується функцією Display(), має тип **int**. Тут не важливо, що ця функція викликається з використанням елемента масиву, оскільки їй передається тільки один його елемент.

***Нео! хідноа пам'ятати!*** *Коли масив використовується як аргумент функції, то функції передається адреса цього масиву. Це означає, що код функції може потенційно змінити реальний вміст масиву, використовуваного під час виклику функції.*

Наприклад, у наведеному нижче коді програми функція Cube() перетворить значення кожного елемента масиву в куб цього значення. Під час виклику функції Cube() як перший аргумент необхідно передати адресу масиву значень, що підлягають перетворенню, а як друге – його розмір.

**Код програми 7.10. Демонстрація механізму перетворення значень елементів масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void Cube(int *n, int num);  // Попереднє оголошення функції
```

```
int main()
{
        int i, Array[10];

        for(i=0; i<10; i++) Array[i] = i+1;
        cout << "Початковий вміст масиву: ";
        for(i=0; i<10; i++) cout << Array[i] << endl;
        cout << endl;

        Cube(Array, 10); // Обчислюємо куби значень.

        cout << "Змінений вміст: ";
        for(i=0; i<10; i++) cout << Array[i] << endl;

        getch(); return 0;
}

void Cube(int *n, int num)      // Визначення функції
{
        while(num) {
                *n = *n * *n * *n;
                num--;
                n++;
        }
}
```

Результати виконання цієї програми є такими:

```
Початковий вміст масиву: 1 2 3 4 5 6 7 8 9 10
Змінений вміст: 1 8 27 64 125 216 343 512 729 1000
```

Як бачите, після звернення до функції Cube() вміст масиву Array змінився: кожен елемент став таким, що дорівнює кубу початкового значення. Іншими словами, елементи масиву Array були модифіковані настановами, які є складовими тіла функції Cube(), оскільки її параметр п вказує на масив Array.

### 7.2.3. Передача рядків функціям

Як зазначалося вище, рядки у мові програмування С++ – звичайні символьні масиви, які завершуються нульовим символом. Таким чином, при передачі функції рядка реально передається тільки покажчик (типу **char** *) на початок цього рядка. Розглянемо, наприклад, таку програму. У ній визначається функція StrFun(), яка перетворить рядок символів у її прописний еквівалент.

**Код програми 7.11. Демонстрація механізму передачі рядка функції**

```
#include <vcl>
#include <iostream>   // Потокове введення-виведення
#include <conio>      // Консольний режим роботи
#include <cstring>    // Для роботи з рядковими типами даних
#include <cctype>     // Для роботи з символьними аргументами
```





```
using namespace std;          // Використання стандартного простору імен

void StrFun(char *str);       // Попереднє оголошення функції

int main()
{
        char str[80];
        strcpy(str, "Мені подобається мова програмування С++");

        StrFun(str);
        cout << str << endl;  // Відображаємо рядок з використанням
                              // прописного написання символів.
        getch(); return 0;
}

void StrFun(char *str)        // Визначення функції
{
        while(*str) {
                *str = toupper(*str); // Отримуємо прописний еквівалент одного символу.
                str++;                // Переходимо до наступного символу.
        }
}
```

Результати виконання цієї програми є такими:

```
МЕНІ ПОДОБАЄТЬСЯ МОВА ПРОГРАМУВАННЯ С++
```

Зверніть увагу на те, що параметр str функції StrFun() оголошується з використанням типу **char** *. Це дає змогу отримати покажчик на символьний масив, який містить рядок.

Розглянемо ще один приклад передачі рядка функції. Як було зазначено в розд. 5, стандартна бібліотечна функція **strlen**() повертає довжину рядка. У наведеному нижче коді програми показано один з можливих варіантів реалізації цієї функції.

**Код програми 7.12. Демонстрація механізму використання функції strlen()**

```
#include <iostream>   // Потокове введення-виведення
using namespace std;          // Використання стандартного простору імен

int myStrlen(char *str);

int main()
{
        cout << "Довжина рядка ПРИВІТ УСІМ дорівнює: ";
        cout << myStrlen("ПРИВІТ УСІМ");

        getch(); return 0;
}

// Нестандартна реалізація функції strlen().
int myStrlen(char *str)
```

```
{
    for(int i=0; str[i]; i++); // Знаходимо кінець рядка.

    return i;
}
```

Ось як виглядають результати виконання цієї програми.

```
Довжина рядка ПРИВІТ УСІМ дорівнює: 11
```

Як вправу Вам варто було б спробувати самостійно реалізувати інші рядкові функції, наприклад **strcpy**() або **strcpy**(). Цей тест дасть змогу дізнатися, наскільки добре Ви засвоїли такі елементи мови С++, як масиви, рядки і покажчики.

## 7.3. Аргументи основної функції main(): argc і argv

Іноді виникає потреба передати інформацію програмі при її запуску. Як правило, це реалізується шляхом передачі аргументів командного рядка функції **main**(). Аргумент командного рядка є інформацією, що вказується в команді (командному рядку), призначеною для виконання операційною системою, після імені програми[1]. Наприклад, С++-програми можна компілювати шляхом виконання наступної команди:

```
cl prog_name
```

У цьому записі елемент *prog_name* – ім'я програми, яку ми хочемо скомпілювати. Ім'я програми передається С++-компіляторові як аргумент командного рядка.

*Аргумент командного рядка є інформацією, що задається в командному рядку після імені програми.*

У мові програмування С++ для основної функції **main**() визначено два вбудованих, але необов'язкових параметри, **argc** і **argv**, які набувають свої значення від аргументів командного рядка. У конкретному операційному середовищі можуть підтримуватися і інші аргументи (таку інформацію необхідно уточнити по документації, що додається до Вашого компілятора). Розглянемо параметри **argc** і **argv** детальніше.

***Вартоа' нати!*** *Формально для імен параметрів командного рядка можна вибирати будь-які ідентифікатори, проте імена* ***argc*** *і* ***argv*** *використовуються за домовленістю вже протягом декількох років. Тому є сенс не вдаватися до інших імен ідентифікаторів, щоб будь-який програміст, якому доведеться розбиратися у Вашій програмі, зміг швидко ідентифікувати їх як параметри командного рядка.*

Параметр **argc** має цілочисельний тип і призначений для зберігання кількості аргументів командного рядка. Його значення завжди не менше від одиниці, оскільки ім'я програми також є одним з аргументів, що враховуються.

[1] У Windows команда "Run" (Виконати) також використовує командний рядок.





Параметр **argv** є покажчик на масив символьних покажчиків. Кожен покажчик в масиві **argv** посилається на рядок, що містить аргумент командного рядка. Елемент **argv**[0] вказує на ім'я програми; елемент **argv**[1] – на перший аргумент, елемент **argv**[2] – на другий і т.д. Всі аргументи командного рядка передаються програмі як рядки, тому числові аргументи необхідно перетворити у програмі у відповідний внутрішній формат.

Важливо правильно оголосити параметр **argv**, а саме:

```
char *argv[];
```

Доступ до окремих аргументів командного рядка можна отримати шляхом індексації масиву **argv**. Як це зробити, показано у наведеному нижче коді програми. Під час її виконання на екран виводиться вітання ("Привіт"), а за ним Ваше ім'я, яке повинно бути першим аргументом командного рядка.

**Код програми 7.12. Демонстрація механізму доступу до окремих аргументів командного рядка**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main(int argc, char *argv[])
{
    if(argc !=2) {
        cout << "Ви забули ввести своє ім'я" << endl;
        return 1;
    }
    cout << "Привіт, " << argv[1] << endl;

    getch(); return 0;
}
```

Припустимо, що Ви називаєтеся Тимур і що Ви назвали цю програму іменем name. Тоді, якщо запустити цю програму, ввівши команду name Тимур, результат її роботи повинен виглядати так: Привіт, Тимур. Наприклад, Ви працюєте з диском A, і у відповідь на пропозицію введення команди Ви повинні ввести згадану вище команду і отримати такий результат:

```
A>name Тимур
Привіт, Тимур
A>
```

У мові програмування С++ точно не визначено, як мають бути представлені аргументи командного рядка, оскільки середовища виконання (операційні системи) мають тут великі відмінності. Проте найчастіше використовують така домовленість: кожен аргумент командного рядка повинен бути відокремлений пропуском або символом табуляції. Як правило, коми, крапки з комою і подібні до них знаки не є допустимими роздільниками аргументів. Наприклад, рядок

```
один, два і три
```

складається з чотирьох рядкових аргументів, тоді як рядок

```
один, два, три
```

містить тільки два, оскільки кома не є допустимим роздільником.

Якщо необхідно передати як один аргумент командного рядка набір символів, який містить пропуски, то його потрібно помістити в лапки. Наприклад, цей набір символів буде сприйнятий як один аргумент командного рядка:

"це тільки один аргумент"

---

***Вартоа' нати!*** *Представлені тут приклади застосовуються до широкого діапазону середовищ мови програмування C++, але це не означає, що Ваше середовище входить до їх числа.*

Щоб отримати доступ до окремого символу в одному з аргументів командного рядка, під час звернення до масиву **argv** додайте другий індекс. Наприклад, у процесі виконання наведеної нижче програми по-символьно відображаються всі аргументи, з якими вона була викликана.

**Код програми 7.13. Демонстрація по-символьного виведення всіх аргументів командного рядка, з яких вона була викликана**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main(int argc, char *argv[])
{
    for(int t=0; t<argc; ++t) {
        int i=0;
        while(argv[t][i]) {
            cout << argv[t][i];
            ++i;
        }
        cout << " ";
    }

    getch(); return 0;
}
```

Неважко здогадатися, що перший індекс масиву **argv** дає змогу отримати доступ до відповідного аргументу командного рядка, а другий – до конкретного символу цього рядкового аргументу.

Звичайно аргументи **argc** і **argv** використовуються для введення у програму початкових параметрів, початкових значень, імен файлів або варіантів (режимів) роботи програми. У мові програмування C++ можна ввести стільки аргументів командного рядка, скільки допускає операційна система. Використання аргументів командного рядка додає програмі професійний вигляд і дає змогу використовувати її в командному файлі (виконуваному текстовому файлі, що містить одну або декілька команд).

### 7.3.1. Передача програмі числових аргументів командного рядка

Як було зазначено вище, при передачі програмі числових даних як аргументів командного рядка, ці дані приймаються в рядковій формі. У програмі повинно бути передбачено їх перетворення у відповідний внутрішній формат





за допомогою однієї із стандартних бібліотечних функцій, які підтримуються мовою C++. Наприклад, у процесі виконання наведеної нижче програми виводиться сума двох чисел, які вказуються в командному рядку після імені програми. Для перетворення аргументів командного рядка у внутрішнє представлення тут використовується стандартна бібліотечна функція **atof**(). Вона перетворить число з рядкового формату в значення типу **double**.

**Код програми 7.14. Демонстрація механізму знаходження суми двох числових аргументів командного рядка**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен

int main(int argc, char *argv[])
{
    double a, b;
    if(argc !=3) {
        cout << "Використання: add число число \n";
        return 1;
    }
    a = atof(argv[1]);
    b = atof(argv[2]);
    cout << a + b;

    getch(); return 0;
}
```

Щоб додати два числа, використовується командний рядок такого вигляду (припускаючи, що ця програма має ім'я add).

C>add 100.2 231

### 7.3.2. Перетворення числових рядків у числа

Стандартна бібліотека C++ містить декілька функцій, які дають змогу перетворити рядкове представлення числа в його внутрішній формат. Для цього використовуються такі функції, як **atoi**(), **atol**() і **atof**(). Вони перетворять рядок в цілочисельне значення (типу **int**), довге ціле (типу **long**) і значення з плинною крапкою (типу **double**) відповідно. Використання цих функцій (для їх виклику необхідно приєднати до програми заголовний файл <**cstdlib**>) продемонстровано у наведеному нижче коді програми.

**Код програми 7.15. Демонстрація механізму використання функцій atoi(), atol() і atof()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен

int main()
{
    int c;
```

```
    long d;
    double f;

    c = atoi("100");
    d = atol("100000");
    f = atof("-0.123");
    cout << c << " " << d << " " << f;
    cout << endl;
    getch(); return 0;
}
```

Результати виконання цієї програми є такими:

```
100 100000 -0.123
```

Функції перетворення рядків корисні не тільки при передачі числових даних програмі через аргументи командного рядка, але і у деяких інших ситуаціях.

## 7.4. Механізм використання настанови return у функціях

Дотепер (починаючи з розд. 2) ми використовували настанову **return** без детальних пояснень. Нагадаємо, що настанова **return** здійснює дві важливі операції. По-перше, вона забезпечує негайне повернення керування до ініціатора виклику функції. По-друге, її можна використовувати для передачі значення, що повертається функцією. Саме цим двом операціям і присвячено даний розділ.

### 7.4.1. Завершення роботи функції

Як зазначалося вище, керування від функції передається ініціатору її виклику в двох ситуаціях: або внаслідок виявлення фігурної дужки, що закривається, або у процесі виконання настанови **return**. Настанову **return** можна використовувати з певним заданим значенням або без нього. Але, якщо в оголошенні функції вказано тип значення (тобто не тип **void**), що повертається, то функція повинна повертати значення цього типу. Тільки **void**-функції можуть використовувати настанову **return** без будь-якого значення.

Для **void**-функцій настанова **return** в основному використовується як елемент програмного керування. Наприклад, у наведеній нижче функції виводиться результат зведення числа в позитивний цілочисельний степінь. Якщо ж показник степеня виявиться негативним, настанова **return** забезпечить вихід з функції, перш ніж буде зроблена спроба обчислити такий вираз. У цьому випадку настанова **return** діє як керівний елемент, тобто запобігає небажаному виконанню певної частини функції.

```
void power(int base, int exp)
{
    if(exp<0) return; /* Щоб не допустити зведення числа в негативний
                         степінь, тут здійснюється повернення у функцію, яка
                         викликає, і ігнорується решта частини функції. */
    int c = 1;
    for(; exp; exp--) c = base * c;
    cout << "Результат дорівнює: " << c;
}
```

Функція може містити декілька настанов **return**. Функція буде завершена у процесі виконання хоч би одного з них. Наприклад, такий фрагмент коду програми є абсолютно правомірним.

```
void Fun()
{
    //...
    switch(c) {
        case 'a': return;
        case 'b': //...
        case 'c': return;
    }
    if(count<100) return; //...
}
```

Проте необхідно мати на увазі, що дуже велика кількість настанов **return** може погіршити ясність алгоритму і ввести в оману тих, хто буде у ньому розбиратися. Декілька настанов **return** варто використовувати тільки у тому випадку, якщо вони сприяють ясності функції.

### 7.4.2. Повернення значень з функції

Кожна функція, окрім типу **void**, повертає яке-небудь значення. Це значення безпосередньо задається за допомогою настанови **return**. Іншими словами, будь-яку не **void**-функцію можна використовувати як операнд у виразі. Отже, кожний з наступних виразів допустимо у мові програмування C++:

```
x = power(y);

if(max(x, y)) > 100) cout << "більше";

switch(fabs(x)) {
```

Незважаючи на те, що всі не **void**-функції повертають значення, вони необов'язково мають бути використані у програмі. Найпоширеніше запитання щодо значень, які повертаються функціями, звучить так: "Оскільки функція повертає певне значення, то хіба я не повинен (повинна) присвоїти це значення будь-якій змінній?". Відповідь: ні, це необов'язково. Якщо значення, що повертається функцією, не бере участі в операції присвоєння, воно просто відкидається (втрачається).

Розглянемо наведену нижче програму, у якій використовується стандартна бібліотечна функція **abs**().

**Код програми 7.16. Демонстрація механізму використання стандартної бібліотечної функції abs()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
```

```
using namespace std;          // Використання стандартного простору імен

int main()
{
      int c = abs(-10);       // рядок 1
      cout << abs(-23);       // рядок 2
      abs(100);               // рядок 3

      getch(); return 0;
}
```

Функція **abs**() повертає абсолютне значення свого цілочисельного аргументу. Вона використовує заголовок <**cstdlib**>. У рядку 1 значення, що повертається функцією **abs**(), присвоюється змінній c. У рядку 2 значення, що повертається функцією **abs**(), нічому не присвоюється, але використовується настановою **cout**. Нарешті, в рядку 3 значення, що повертається функцією **abs**(), втрачається, оскільки не присвоюється ніякій іншій змінній і не використовується як частина виразу.

Якщо функція, тип якої є відмінним від типу **void**, завершується внаслідок виявлення фігурної дужки, що закривається, то значення, яке вона повертає, не визначене (тобто невідоме). Через особливості формального синтаксису C++ не **void**-функція не зобов'язана виконувати настанову **return**. Це може відбутися у тому випадку, якщо кінець функції буде досягнуто до виявлення настанови **return**. Але, оскільки функція оголошена як така, що повертає значення, значення буде таки повернено, навіть якщо це просто "сміття". У загальному випадку не кожна зі створюваних Вами **void**-функцій повинна повертати значення за допомогою безпосередньо виконуваної настанови **return**.

Вище згадувалося, що **void**-функція може мати декілька настанов **return**. Те саме стосується і функцій, які повертають значення. Наприклад, представлена у наведеному нижче коді програми функція ReturnFun() використовує дві настанови **return**, які дають змогу спростити алгоритм її роботи. Ця функція виконує пошук заданого підрядка в заданому рядку. Вона повертає індекс першого виявленого входження заданого підрядка або значення -1, якщо заданий підрядок не було знайдено. Наприклад, якщо в рядку "Я люблю C++" необхідно відшукати підрядок "люблю", то функція ReturnFun() поверне число 2 (яке є індекс символу "л" у рядку "Я люблю C++").

**Код програми 7.17. Демонстрація механізму використання двох настанов return, які значно спрощують алгоритм роботи програми**

```
#include <iostream>           // Потокове введення-виведення
using namespace std;          // Використання стандартного простору імен

int ReturnFun(char *sub, char *str);

int main()
{
      int index = ReturnFun("три", "один два три чотири");
```





```
      cout << "Індекс дорівнює " << index; // Індекс дорівнює 9.

      getch(); return 0;
}

// Функція повертає індекс шуканого підрядка або -1, якщо його не було знайдено.
int ReturnFun(char *sub, char *str)
{
      char *p, *p2;

      for(int t=0; str[t]; t++) {
            p = &str[t]; // Встановлення покажчиків
            p2 = sub;
            while(*p2 && *p2==*p) { // перевірка збігу
                  p++; p2++;
            }
            // Якщо досягнуто кінець p2-рядка (тобто підрядка), то його було знайдено.
            if(!*p2) return t; // Повертаємо індекс підрядка.
      }

      return -1; // Підрядок не був виявлений.
}
```

Результати виконання цієї програми є такими:

```
Індекс дорівнює 9
```

Оскільки шуканий підрядок існує в заданому рядку, здійснюється перша настанова **return**. Для прикладу змінимо програму так, щоб нею виконувався пошук підрядка, який не є частиною заданого рядка. У цьому випадку функція ReturnFun() повинна повернути значення -1 (завдяки другій настанові **return**).

Функцію можна оголосити так, щоб вона повертала значення будь-якого типу даних, дійсного у мові C++ (за винятком масиву: функція не може повернути масив). Спосіб оголошення типу значення, що повертається функцією, аналогічний тому, який використовують для оголошення змінних: імені функції повинен передувати специфікатору типу даних. Застосування специфікатора типу даних повідомляє компілятор, значення якого типу даних повинна повернути функція. Указаний в оголошенні функції тип повинен бути сумісним з типом даних, які використовується в настанові **return**. Інакше компілятор відреагує повідомленням про помилку.

### 7.4.3. Функції, які не повертають значень (void-функції)

Як Ви помітили, функції, які не повертають значень, оголошуються з вказанням типу **void**. Це ключове слово не допускає їх використання у виразах і захищає від невірного застосування. У наведеному нижче прикладі функція print_vertical() виводить аргумент командного рядка у вертикальному напрямі (вниз) по лівому краю екрана. Оскільки ця функція не повертає ніякого значення, в її оголошенні використано ключове слово **void**.

**Код програми 7.18. Демонстрація механізму виведення аргументу командного рядка у вертикальному напрямі (вниз) по лівому краю екрана**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void print_vertical(char *str);

int main(int argc, char *argv[])
{
    if(argc==2) print_vertical(argv[1]);

    getch(); return 0;
}

void print_vertical(char *str)
{
    while(*str) cout << *str++ << endl;
}
```

Оскільки print_vertical() оголошена як **void**-функція, то її не можна використовувати у виразі. Наприклад, наступна настанова неправильна і тому не буде компільована:

```
x = print_vertical("Привіт!"); // помилка
```

***Вартоа' нати!а****У перших версіях мови С не було передбачено типу* ***void****. Таким чином, у старих С-програмах функції, які не повертають значень, за замовчуванням мали тип* ***int****. Якщо Вам доведеться натрапити на такі функції у процесі перекладу старих С-програм "на рейки" С++, просто оголосіть їх з використанням ключового слова* ***void****, зробивши їх* ***void****-функціями.*

### 7.4.4. Повернення покажчиків з функції

Функції можуть повертати покажчики. Покажчики повертаються подібно до значень будь-яких інших типів даних і не створюють при цьому особливих проблем. Але, оскільки покажчик є одним з найскладніших (або небезпечних) засобів мови програмування С++, то є сенс присвятити йому окремий підрозділ.

Щоб повернути покажчик, функція повинна оголосити його тип як тип значення, що повертається. Наприклад, нижче оголошується тип значення, що повертається функцією Fun(), яка має повертати покажчик на ціле число:

```
int *Fun();
```

Якщо функція повертає покажчик, то значення, що використовується в її настанові **return**, також повинно бути покажчиком[1].

У наведеному нижче коді програми продемонстровано механізм використання покажчика як типу значення, що повертається. Це нова версія наведеної вище функції ReturnFun(), тільки тепер вона повертає не індекс знайденого підрядка, а покажчик на неї. Якщо заданий підрядок не знайдений, повертається нульовий покажчик.

[1] Як і для всіх функцій, **return**-значення повинне бути сумісним з типом значення, що повертається.





**Код програми 7.19. Демонстрація нової версії функції ReturnFun(), яка повертає покажчик на підрядок**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

char *ReturnFun(char *sub, char *str);

int main()
{
    char *substr;

    substr = ReturnFun("три", "один два три чотири");
    cout << "Знайдений підрядок: " << substr;
    getch(); return 0;
}
// Функція повертає покажчик на шуканий підрядок або нуль,
// якщо такий не буде знайдено.
char *ReturnFun(char *sub, char *str)
{
    char *p, *p2, *start;

    for(int t=0; str[t]; t++) {
        p = &str[t]; // Встановлення покажчиків
        start = p;
        p2 = sub;
        while(*p2 && *p2==*p) { // Перевірка збігу
            p++;
            p2++;
        }

        // Якщо досягнуто кінець p2-підрядка, то цей підрядок буде знайдено.
        if(!*p2) return start; // Повертаємо покажчик на початок знайденого підрядка.
    }
    getch(); return 0; // Підрядок не знайдено
}
```

У процесі виконання цієї версії програми отримано такий результат:

```
Знайдений підрядок: три чотири
```

У цьому випадку, коли підрядок "три" було знайдено в рядку "один два три чотири", функція ReturnFun() повернула покажчик на початок шуканого підрядка "три", який у функції **main**() був присвоєний змінній substr. Таким чином, під час виведення значення substr на екрані відобразився залишок рядка, тобто " три чотири".

Багато підтримуваних С++ бібліотечних функцій, призначених для оброблення рядків, повертають покажчики на символи. Наприклад, функція **strcpy**() повертає покажчик на перший аргумент.

### 7.4.5. Прототипи функцій

Дотепер у прикладах програм, наведених тут, прототипи функцій використовувалися без жодних пояснень. Тепер настав час поговорити про них детально. У мові програмування C++ всі функції мають бути оголошені до їх використання. Переважно це реалізується за допомогою прототипу функції. Прототипи містять три види інформації про функцію: тип значення, що повертається нею; тип її параметрів; кількість параметрів.

Прототипи дають змогу компіляторові виконати такі три операції.

- вони повідомляють компілятор, програмний код якого типу необхідно генерувати під час виклику функції. Відмінності в типах параметрів і значенні, що повертається функцією, забезпечують різне оброблення компілятором;
- вони дають змогу C++ виявити неприпустимі перетворення типів аргументів, що використовуються під час виклику функції, в тип, який було вказано в оголошенні її параметрів, і повідомити про них;
- вони дають змогу компіляторові виявити відмінності між кількістю аргументів, що використовуються під час виклику функції, і кількістю параметрів, заданих у визначенні функції.

Загальна форма прототипу функції аналогічна її визначенню за винятком того, що в прототипі не представлено тіла функції.

*type func_name*(*type parm_name1, type parm_name2,., type parm_nameN*);

Використання імен параметрів у прототипі необов'язкове, але дає змогу компіляторові ідентифікувати будь-який збіг типів під час виникнення помилки, тому краще імена параметрів все ж таки помістити в прототип функції.

Щоб краще зрозуміти корисність прототипів функцій, розглянемо наведену нижче програму. Якщо Ви спробуєте її скомпілювати, то отримаєте від компілятора повідомлення про помилку, оскільки у цьому коді програми робиться спроба викликати функцію sqr_Fun() з цілочисельним аргументом, а не з покажчиком на цілочисельне значення (згідно з прототипом функції). Помилка полягає в неприпустимості перетворення цілочисельного значення в покажчик.

**Код програми 7.20. Демонстрація механізму використання прототипу функції, яка дає змогу здійснити строгий контроль типів даних**

```
void sqr_Fun(int *c);            // Попереднє оголошення прототипу функції

int main()
{
      int x = 10;
      sqr_Fun(x);                // *** Помилка *** -- невідповідність типів!
      getch(); return 0;
}

void sqr_Fun(int *c)             // Визначення прототипу функції
{
      *c = *c * *c;
}
```





***Вартоа' нати!а****Незважаючи на те, що мова програмування С допускає прототипи, однак їх використання не є обов'язковим. Йдеться про те, що в перших версіях мови С вони не застосовувалися. Тому у процесі перекладу старого С-коду в С++-код перед компіляцією програми необхідно забезпечити наявність прототипів абсолютно для всіх функцій.*

### 7.4.6. Заголовки у C++-програмах

На початку цього навчального посібника Ви дізналися про існування стандартних заголовків у C++-програмах, які містять інформацію, необхідну для Ваших програм. Хоча все зазначене вище є істинною правдою, це ще не вся правда. Заголовки у C++-програмах містять прототипи стандартних бібліотечних функцій, а також різні значення і визначення, що використовуються цими функціями. Подібно до функцій, які створюються програмістами, стандартні бібліотечні функції також повинні "заявити про себе" у формі прототипів до їх використання. Тому будь-яка програма, у якій використовується бібліотечна функція, має містити заголовок з прототипом цієї функції.

Щоб дізнатися, який заголовок необхідний для тієї або іншої бібліотечної функції, необхідно звернутися до довідкової настанови, що додається до Вашого компілятора. Окрім опису кожної функції, там повинно бути вказано ім'я заголовка, який необхідно приєднати до програми для використання вибраної функції.

#### Порівняння старого і нового стилів оголошення параметрів функцій

Якщо Вам доводилося коли-небудь розбиратися в старому С-коді програми, то Ви, можливо, звернули увагу на незвичайне (з погляду сучасного програміста) оголошення параметрів функції. Цей старий стиль оголошення параметрів, який іноді називають *класичним форматом*, застарілий, але на нього дотепер можна натрапити у програмах раннього періоду. У мові програмування C++ (і оновленому С-коді програми) використовується нова форма оголошень параметрів функцій. Але, якщо Вам доведеться працювати із старими С-програмами і, особливо, якщо знадобиться переводити їх в C++-код, то Вам буде корисно розуміти і форму оголошення параметрів, "витриману" в старому стилі.

Оголошення параметра функції, згідно зі старим стилем, складається з двох частин: переліку параметрів, поміщених у круглі дужки, які наводяться після імені функції, і власне оголошення параметрів, яке повинно знаходитися між закритою круглою дужкою і відкритою фігурною дужкою функції. Наприклад, це "нове" оголошення (тобто за новим стилем)

```
float Fun(int a, int b, char ch)
{...
```

виглядатиме з використанням старого стилю дещо по-іншому.

```
float Fun(a, b, ch)
int a, b;
char ch;
{...
```

Зверніть увагу на те, що в класичній формі після того, яка вказано ім'я типу, в переліку може знаходитися декілька параметрів. У новій формі це оголошення не допускається.

У загальному випадку, щоб перетворити оголошення параметрів із старого стилю в новий (C++-стиль), достатньо внести оголошення типів параметрів у круглі дужки, наступні за іменем функції. При цьому кожен параметр необхідно оголосити окремо, з власним специфікатором типу.

### 7.4.7. Організація рекурсивних функцій

***Рекурсія*** – остання тема, яку ми розглянемо у цьому розділі. *Рекурсія*, яку іноді називають циклічним визначенням, є процес визначення чого-небудь на власній основі. В області програмування під рекурсією розуміють процес виклику функцією самої себе. Функцію, яка викликає саму себе, називають рекурсивною.

Класичним прикладом рекурсії є обчислення факторіалу від числа за допомогою функції factr(). Факторіал числа *N* є добуток всіх цілих чисел від 1 до *N*. Наприклад, факторіал числа 3 дорівнює 1·2·3, або 6. Рекурсивний спосіб обчислення факторіалу від числа продемонстровано у наведеному нижче коді програми. Для порівняння сюди ж включений і його нерекурсивний (ітеративний) еквівалент.

**Код програми 7.21. Демонстрація механізму роботи рекурсивного способу обчислення факторіалу**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int factr(int n), fact(int n);

int main()
{
      // Використання рекурсивної версії.
      cout << "Факторіал числа 4 дорівнює " << factr(4) << endl;
      // Використання ітеративної версії.
      cout << "Факторіал числа 4 дорівнює " << fact(4) << endl;
      getch(); return 0;
}

int factr(int n)   // Рекурсивна версія функції.
{
      int rezult;
      if(n == 1) return (1);
      rezult = factr(n-1)*n;
      return (rezult);
}

int fact(int n)    // Ітеративна версія функції.
{
```





```
      int rezult = 1;
      for(int t=1; t<=n; t++) rezult = rezult*(t);
      return (rezult);
}
```

Нерекурсивна версія функції fact() достатньо проста і не вимагає розширених пояснень. У ній використовується цикл, у якому організовано множення послідовних чисел, починаючи з 1 і закінчуючи числом, заданим як параметр: на кожній ітерації циклу поточне значення керованої змінної циклу множиться на поточне значення добутку, отримане внаслідок виконання попередньої ітерації циклу.

Рекурсивна функція factr() є дещо складнішою. Якщо вона викликається з аргументом, що дорівнює 1, то відразу повертає значення 1. В іншому випадку вона повертає добуток factr(n-1)*n. Для обчислення цього виразу викликається метод factr() з аргументом n-1. Цей процес повторюється доти, доки аргумент не стане таким, що дорівнює 1, після чого викликані раніше методи почнуть повертати значення. Наприклад, під час обчислення факторіалу від числа 2 перше звернення до методу factr() приведе до другого звернення до того ж методу, але з аргументом, що дорівнює 1. Другий виклик методу factr() поверне значення 1, яке буде помножене на 2 (початкове значення параметра n). Можливо, Вам буде цікаво вставити у функцію factr() настанову **cout**, щоб показати ієрархічний рівень кожного виклику і проміжні результати.

Коли функція викликає саму себе, в системному стеку виділяється пам'ять для нових локальних змінних і параметрів, і код функції із самого початку здійснюється з цими новими змінними. Рекурсивний виклик не створює нової копії функції. Новими є тільки аргументи. При поверненні кожного рекурсивного виклику із стека витягуються старі локальні змінні та параметри, а виконання функції поновлюється з "внутрішньої" точки її виклику. Про рекурсивні функції можна сказати, що вони "висуваються" і "засуваються".

***Вартоа' нати!*** *Здебільшого використання рекурсивних функцій не дає значного скорочення об'єму коду програми. Окрім цього, рекурсивні версії багатьох процедур виконуються повільніше, ніж їх ітеративні еквіваленти, через додаткові витрати системних ресурсів, пов'язаних з багатократними викликами функцій. Дуже велика кількість рекурсивних звернень до функції може викликати переповнювання стека. Оскільки локальні змінні і параметри зберігаються в системному стеку і кожен новий виклик створює нову копію цих змінних, може настати момент, коли пам'ять стека буде вичерпана. У цьому випадку можуть бути зруйновані інші ("ні у чому не винні") дані. Але, якщо рекурсія побудована коректно, про це навряд чи варто хвилюватися.*

Основна перевага рекурсії полягає у тому, що деякі типи алгоритмів рекурсивно реалізуються простіше, ніж їх ітеративні еквіваленти. Наприклад, алгоритм сортування Quicksort достатньо важко реалізувати ітеративним способом. Окрім цього, деякі завдання (особливо ті, які пов'язані з штучним інтелектом) просто створені для рекурсивних вирішень. Нарешті, у деяких програмістів процес мислення організований так, що їм простіше думати рекурсивно, ніж ітеративно.

Під час написання рекурсивної функції необхідно включити в неї настанову перевірки умови (наприклад, **if**-настанову), яка б забезпечувала вихід з функції без виконання рекурсивного виклику. Якщо цього не зробити, то, викликавши одного разу таку функцію, з неї вже не можна буде повернутися. Під час роботи з рекурсією це найпоширеніший тип помилки. Тому у процесі розроблення програм з рекурсивними функціями не варто ігнорувати настанову **cout**, щоб бути в курсі того, що відбувається в конкретній функції, і мати можливість перервати її роботу у разі виявлення помилки.

Розглянемо ще один приклад рекурсивної функції. Функція **reverse**() використовує рекурсію для відображення свого рядкового аргументу в зворотному порядку.

**Код програми 7.22. Демонстрація механізму відображення рядка в зворотному порядку за допомогою рекурсії**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void reverse(char *s);
int main()
{
      char str[] = "Це тест";
      reverse(str);
      getch(); return 0;
}

// Виведення рядка в зворотному порядку.
void reverse(char *s)
{
      if(*s) reverse(s+1);
      else return;
      cout << *s;
}
```

Функція **reverse**() перевіряє, чи не переданий їй як параметр покажчик на нуль, яким завершується рядок. Якщо ні, то функція **reverse**() викликає саму себе з покажчиком на наступний символ у рядку. Цей "закручений" процес повторюється доти, доки тієї ж самої функції не буде передано покажчик на нуль. Коли, нарешті, виявиться символ кінця рядка, розпочнеться процес "розкручування", тобто викликані раніше функції почнуть повертати значення, і кожне повернення супроводжуватиметься "довиконанням" методу, тобто відображенням символу s. Внаслідок цього початковий рядок посимвольно відобразиться в зворотному порядку.

Розроблення рекурсивних функцій часто викликає труднощі у програмістів-початківців. Але з приходом досвіду використання рекурсії стає для багатьох звичайною практикою.





# Розділ 8. ВИКОРИСТАННЯ ЗАСОБІВ ПРОГРАМУВАННЯ ДЛЯ РОЗШИРЕННЯ МОЖЛИВОСТЕЙ C++-ФУНКЦІЙ

У цьому розділі продовжимо вивчення деяких особливостей застосування C++-функцій, а саме розглянемо три засоби програмування: посилання, перевизначення функцій і використання аргументів за замовчуванням. Вони значною мірою розширюють можливості розроблених функцій користувача. Як буде показано далі, посилання – неявний покажчик. Перевизначення функцій є властивістю, яка дає змогу одну і ту саму функцію реалізувати декількома способами, причому у кожному випадку можливе виконання окремого завдання. Тому є всі підстави вважати перевизначення функцій одним із шляхів підтримки поліморфізму мови C++. Використовуючи можливість задавання аргументів за замовчуванням, можна визначити значення для параметра, яке буде автоматично застосоване у випадку, якщо відповідний аргумент не задано.

Оскільки до параметрів функцій часто застосовуються посилання (це основна причина їх існування), почнемо цей розділ з короткого перегляду способів передачі аргументів функціям.

## 8.1. Способи передачі аргументів функціям

Щоб зрозуміти походження посилання, необхідно знати теорію процесу передачі аргументів функціям. У загальному випадку в мовах програмування, як правило, передбачається два способи, які дають змогу передавати аргументи в підпрограми (функції, методи, процедури). Перший називається *викликом за значенням* (call-by-value). У цьому випадку значення аргументу копіюється у формальний параметр підпрограми. Значить зміни, внесені в параметри підпрограми, не впливають на аргументи, що використовуються під час її виклику.

*Під час виклику функції за значенням передається значення аргументу.*

Другий спосіб передачі аргументу підпрограмі називається *викликом за посиланням* (call-by-reference). У цьому випадку в параметр копіюється адреса аргументу (а не його значення). У межах викликуваної підпрограми цю адресу використовують для доступу до реального аргументу, що задається під час її виклику. Це означає, що зміни, внесені в параметр, нададуть дію на аргумент, що використовується під час виклику підпрограми.

*Під час виклику функції з використанням механізму посилання зразу ж передається адреса аргументу.*

### 8.1.1. Механізм передачі аргументів у мові програмування C++

За замовчуванням для передачі аргументів у мові програмування C++ використовується метод виклику за значенням. Це означає, що в загальному випадку код функції не може змінити аргументи, що використовуються під час виклику функції. В усіх програмах цього навчального посібника, представлених дотепер, використовувався метод виклику за значенням.

Розглянемо таку функцію.

**Код програми 8.1. Демонстрація механізму передачі аргументів у мові програмування C++**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int sqr_Fun(int x);

int main()
{
    int t=10;
    cout << "t^2=" << sqr_Fun(t) << "; t=" << t;

    getch(); return 0;
}

int sqr_Fun(int x)
{
    x = x*x;
    return x;
}
```

У наведеному прикладі значення аргументу 10, що передається функції sqr_Fun(), копіюється в параметр x. У процесі її виконання присвоєння x = x*x змінюється тільки локальній змінній x. Змінна t, використовувана під час виклику функції sqr_Fun(), як і раніше, матиме значення 10 і на неї ніяк не вплинуть операції, що виконуються у цій функції. Отже, після запуску цієї програми на екран монітора буде виведено такий результат: t^2=100; t=10.

***Нео! Відноапам'ятати!*** *За замовчуванням функції передається копія аргументу. Те, що відбувається усередині функції, ніяк не відображається на значенні змінної, що використовується під час виклику функції.*

### 8.1.2. Механізм використання покажчика для забезпечення виклику функції за посиланням

Незважаючи на те, що як C++-узгодження про передачу параметрів за замовчуванням діє виклик функції за значенням, існує можливість "вручну" замінити його викликом за посиланням. У цьому випадку функції передаватиметься адреса аргументу (тобто покажчик на аргумент). Це дасть змогу внутрішньому коду функції змінити значення аргументу, яке зберігається поза функцією. Приклад такого "дистанційного" керування значеннями змінних





було показано в попередньому розділі під час перегляду можливості виклику функції з покажчиками (у прикладі коду програми функції передавався покажчик на цілочисельну змінну). Як уже зазначалося вище, покажчики передаються функціям подібно до значень будь-якого іншого типу. Безумовно, для цього необхідно оголосити параметри з типом покажчиків.

Щоб зрозуміти, як передача покажчика дає змогу вручну забезпечити виклик за посиланням, розглянемо таку версію функції swap(). Вона змінює значення двох змінних, на які вказують її аргументи.

```
void swap(int *x, int *y)
{
    int tmp;
    tmp = *x;          // Тимчасово зберігаємо значення, розташоване за адресою x.
    *x = *y;           // Поміщаємо значення, що зберігається
                       // за адресою y, за адресою x.
    *y = tmp;          // Поміщаємо значення, яке раніше зберігалося
                       // за адресою x, за адресою y.
}
```

Тут параметри *x і *y означають змінні, що адресуються покажчиками x і y, які просто є адресами аргументів, що використовуються під час виклику функції swap(). Отже, у процесі виконання цієї функції буде зроблено реальний обмін вмістом змінних, що використовуються під час її виклику.

Оскільки функція swap() чекає, щоб отримати два покажчики, то програміст повинен пам'ятати, що функцію swap() необхідно викликати з адресами змінних, значення яких він хочете обміняти. Коректний виклик цієї функції продемонстровано у наведеному нижче коді програми.

**Код програми 8.2. Демонстрація механізму використання покажчика для забезпечення виклику за посиланням**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

// Оголошуємо функцію swap(), яка використовує покажчики.
void swap(int *x, int *y);

int main()
{
    int c = 10, d = 20;
    cout << "Початкові значення змінних c та d: ";
    cout << c << " " << d << endl;

    swap(&d, &c);          // Викликаємо функцію swap() з адресами змінних c та d.

    cout << "Значення змінних c та d після обміну: ";
    cout << c << " " << d << endl;

    getch(); return 0;
}
```

```
void swap(int *x, int *y)          // Обмін аргументами.
{
      int tmp;

      tmp = *x;  // Тимчасово зберігаємо значення за адресою x.

      *x = *y;      // Поміщаємо значення, збережене раніше за адресою y, за адресою x.

      *y = tmp;  // Поміщаємо значення, яке раніше зберігалося
                        // за адресою x, за адресою y.
}
```

Результати виконання цієї програми є такими:

```
Початкові значення змінних c та d: 10 20
Значення змінних c та d після обміну: 20 10
```

У наведеному прикладі змінній c було присвоєне початкове значення 10, а змінній d – 20. Потім була викликана функція swap() з адресами змінних c та d. Для отримання адрес тут використовується унарний оператор "&". Отже, функції swap() під час виклику були передані адреси змінних c та d, а не їх значення. Після завершення роботи функції swap() змінні c та d обмінялися своїми значеннями.

## 8.2. Поняття про посилальні параметри

Незважаючи на можливість "вручну" організувати виклик за посиланням за допомогою оператора отримання адреси, такий підхід не завжди є зручним. По-перше, він вимушує програміста виконувати всі операції з використанням покажчиків. По-друге, викликаючи функцію, програміст повинен не забути передати їй адреси аргументів, а не їхнє значення. На щастя, у мові програмування C++ можна зорієнтувати компілятор на автоматичне використання виклику за посиланням (замість виклику за значенням) для одного або декількох параметрів конкретної функції. Така можливість реалізується за допомогою *посилального параметра* (reference parameter). Під час використання посилального параметра функції автоматично передається адреса (а не значення) аргументу. У процесі виконання коду функції, а саме у процесі виконання операцій над посилальним параметром, забезпечується його автоматичне перейменування, і тому програмісту не потрібно використовувати оператори, що працюють з покажчиками.

Посилальний параметр оголошується за допомогою символу "&", який повинен передувати імені параметра в попередньому оголошенні функції. Посилальний параметр автоматично отримує адресу відповідного аргументу. Операції, що виконуються над посилальним параметром, роблять вплив на аргумент, що використовується під час виклику самої функції, а не на сам посилальний параметр.





### 8.2.1. Механізм дії посилальних параметрів

Щоб краще зрозуміти механізм дії посилальних параметрів, розглянемо спершу простий приклад. У наведеному нижче коді програми функція Fun() приймає один посилальний параметр типу **int**.

**Код програми 8.3. Демонстрація механізму дії посилального параметра**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Fun(int &c);

int main()
{
      int num = 1;
      cout << "Старе значення змінної num: " << num << endl;

      Fun(num);          // Передаємо адресу змінної num функції Fun().
      cout << "Нове значення змінної num: " << num << endl;
      getch(); return 0;
}

void Fun(int &c)
{
      c = 10;     // Модифікування аргументу, що задається під час його виклику.
}
```

Внаслідок виконання ця програма відображає на екрані такий результат:

```
Старе значення змінної num: 1
Нове значення змінної num: 10
```

Зверніть особливу увагу на визначення функції Fun():

```
void Fun(int &c)
{
      c = 10;     // Модифікування аргументу, що задається під час його виклику.
}
```

Отже, розглянемо оголошення параметра c. Його імені передує символ "&", який "перетворює" змінну c на посилальний параметр[1]. Настанова

```
c = 10;
```

(у цьому випадку вона одна утворює тіло функції) *не* присвоює змінній c значення 10. Насправді значення 10 присвоюється змінній, на яку посилається змінна c (у нашій програмі нею є змінна num). Зверніть увагу на те, що у цій настанові не використовують оператор "*", який є необхідним під час роботи з покажчиками. Застосовуючи посилальний параметр, Ви тим самим повідомляєте C++-компілятор про передачу адреси (тобто покажчика), і компілятор автоматично перейменовує його за Вас. Понад це, якби Ви спробували "допомогти" компіляторові, використавши оператора "*", то відразу ж отри-

[1] Це оголошення також використовується в прототипі функції.

мали б повідомлення про помилку (і справді "жодна добра справа не залишається безкарною").

Оскільки змінна c була оголошена як посилальний параметр, компілятор автоматично передає функції Fun() адресу будь-якого аргументу, з яким викликається ця функція. Таким чином, у функції **main**() настанова

```
Fun(num);              // Передаємо адресу змінної num функції Fun().
```

передає функції Fun() адресу змінної num (а не її значення). Зверніть увагу на те, що під час виклику функції Fun() не потрібно передувати змінній num оператором "&"[1]. Оскільки функція Fun() отримує адресу змінної num у формі посилання, вона може модифікувати значення цієї змінної.

Щоб проілюструвати реальне застосування посилальних параметрів (і тим самим продемонструвати їх перевагу), перепишемо нашу стару знайому функцію swap() з використанням посилань. У наведеному нижче коді програми зверніть увагу на те, як функція swap() спочатку оголошується, а потім у функції **main**() викликається.

**Код програми 8.4. Демонстрація реального застосування посилальних параметрів**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

// Оголошуємо функцію swap() з використанням посилальних параметрів.
void swap(int &x, int &y);

int main()
{
      int c = 10, d = 20;

      cout << " Початкові значення змінних c та d: ";
      cout << c << " " << d << endl;
      swap(d, c);
      cout << " Значення змінних c та d після обміну: ";
      cout << c << " " << d << endl;

      getch(); return 0;
}

// Обмін аргументами.
void swap(int &x, int &y)
{
      int tmp;

      tmp = x;   // Зберігаємо значення, розташоване за адресою x.
      x = y;     // Поміщаємо значення, що зберігається за адресою y за адресою x.
      y = tmp;   // Поміщаємо значення, яке раніше зберігалося
                 // за адресою x, за адресою y.
}
```

[1] Понад це, це було б помилкою.





У цьому коді програми функція swap() визначається з розрахунку на виклик за посиланням, а не на виклик за значенням. Тому вона може виконати обмін значеннями двох аргументів, з якими вона викликається. Знову ж таки, зверніть увагу на те, що оголошення x і y посилальними параметрами позбавляє Вас від потреби використання оператора "*" при організації обміну значеннями. Як уже згадувалося вище, така "нав'язливість" з Вашого боку стала б причиною помилки. Тому запам'ятаєте, що компілятор автоматично генерує адреси аргументів, що використовуються під час виклику функції swap(), і автоматично перейменовує посилання x і y.

Отже, підведемо деякі підсумки. Після створення посилальний параметр автоматично посилається (тобто опосередковано вказує) на аргумент, що використовується під час виклику функції. Понад це, під час виклику функції не потрібно застосовувати до аргументу оператор "&". Окрім цього, в тілі функції посилальний параметр використовується безпосередньо, тобто без використання оператора "*". Всі операції, що містять посилальний параметр, автоматично виконуються над аргументом, що використовується під час виклику функції.

***Нео! хідноапам'ятати!*** *Привласнюючи певне значення посиланню, Ви насправді присвоюєте це значення змінній, на яку вказує це посилання. Тому, застосовуючи посилання як аргумент функції, під час виклику функції Ви насправді використовуєте таку змінну.*

### 8.2.2. Варіанти оголошень посилальних параметрів

У виданій в 1986 р. книзі "*Язык программирования C++*" (у якій був вперше описаний синтаксис мови C++) Б'ярн Страуструп представив стиль оголошення посилальних параметрів, який було схвалено іншими програмістами. Відповідно до цього стилю оператор "&" зв'язується з іменем типу, а не з іменем змінної. Наприклад, ось як виглядає ще один спосіб запису прототипу функції swap().

```
void swap(int& x, int& y);
```

Неважко помітити, що у цьому оголошенні символ "&" прилягає впритул до імені типу **int**, а не до імені змінної x.

Деякі програмісти визначають в такому стилі і покажчики, пов'язуючи символ "*" з типом, а не із змінною, як у наведеному прикладі:

```
float* p;
```

Наведені оголошення відображають бажання деяких програмістів мати у мові програмування C++ окремий тип посилання або покажчика. Але йдеться про те, що таке зв'язування символу "&" або "*" з типом (а не з іменем змінної) не поширюється на весь перелік змінних, які наводяться в оголошенні, що може призвести до плутанини. Наприклад, в наступному оголошенні створюється один покажчик (а не два) на цілочисельну змінну:

```
int* a, b;
```

У цьому записі b оголошується як цілочисельна змінна (а не як покажчик на цілочисельну змінну), оскільки, як визначено синтаксисом мови C++, використаний в оголошенні символ "*" або "&" пов'язується з конкретною змінною, якій він передує, а не з типом, за яким його знаходять.

Важливо розуміти, що для C++-компілятора абсолютно байдуже, як саме Ви напишете оголошення: **int** *p або **int*** p. Таким чином, якщо Ви вважаєте за краще пов'язувати символ "*" або "&" з типом, а не змінною, то вчиняйте так, як Вам зручно. Але, щоб уникнути надалі будь-яких непорозумінь, у цьому посібнику ми пов'язуватимемо символ "*" або "&" з іменем змінної, а не з іменем типу.

***Вартоа' нати!а****У мові C посилання не підтримуються. Тому єдиний спосіб забезпечити у мові C виклик за посиланням полягає у використанні покажчиків, як це було показано вище (див. першу версію функції swap()). Перетворюючи C-код в C++-код, Вам варто замість параметрів-покажчиків використовувати, де це можливо, посилання.*

### 8.2.3. Механізм повернення посилань

Функція може повертати посилання. У програмуванні мовою C++ передбачено декілька застосувань для посилальних значень, що повертаються функціями. Зараз ми продемонструємо тільки деякі з них, а інші розглянемо нижче у цьому посібнику, коли познайомимося з перевизначенням операторів.

Якщо функція повертає посилання, це означає, що вона повертає неявний покажчик на значення, що передається нею настановою **return**. Цей факт відкриває вражаючі можливості: функцію, виявляється, можна використовувати в лівій частині настанови присвоєння! Наприклад, розглянемо таку просту програму.

**Код програми 8.5. Демонстрація механізму повернення посилання**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

double &Fun();
double num = 100.0;

int main()
{
      double val;

      cout << Fun() << endl;   // Відображаємо значення num.
      val = Fun();             // Присвоюємо значення num змінною val.

      cout << val << endl;     // Відображаємо значення val.
      Fun() = 99.1;            // Змінюємо значення num.
      cout << Fun() << endl;   // Відображаємо нове значення num.

      getch(); return 0;
}
```





```
double &Fun()
{
      return num; // Повертаємо посилання на num.
}
```

Ось як виглядають результати виконання цієї програми:

```
100
100
99.1
```

Розглянемо цю програму ґрунтовніше. Судячи з прототипу функції Fun(), вона повинна повертати посилання на **double**-значення. За оголошенням функції Fun() знаходиться оголошення глобальної змінної num, яка ініціалізувалася значенням 100. У процесі виконання такої настанови виводиться початкове значення змінної num:

```
cout << Fun() << endl; // Відображаємо значення num.
```

Після виклику функція Fun() повертає посилання на змінну num. Оскільки функція Fun() оголошена з "зобов'язанням" повернути посилання, то у процесі виконання рядка

```
return num; // Повертаємо посилання на num.
```

автоматично повертається посилання на глобальну змінну num. Це посилання потім використовується настановою **cout** для відображення значення num.

У процесі виконання рядка

```
val = Fun(); // Присвоюємо значення num змінній val.
```

посилання на змінну num, що повертається функцією Fun(), використовують для присвоєння значення num змінній val.

А ось найцікавіший рядок у програмі

```
Fun() = 99.1;            // Змінюємо значення num.
```

У процесі виконання цієї настанови присвоєння значення змінній num дорівнює числу 99,1. І ось чому: оскільки функція Fun() повертає посилання на змінну num, то це посилання і є приймачем настанови присвоєння. Таким чином, значення 99,1 присвоюється змінній num опосередковано, через посилання на неї, яке повертає функція Fun().

Нарешті, у процесі виконання рядка

```
cout << Fun() << endl; // Відображаємо нове значення num.
```

відображається нове значення змінної num (після того, як посилання на змінну num буде повернено внаслідок виклику функції Fun() у настанові **cout**).

Наведемо ще один приклад програми, у якій як значення, що повертається функцією, використовується посилання (або значення посилального типу).

**Код програми 8.6. Демонстрація механізму повернення функцією значення посилального типу**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен
```

```
double &ChangeFun(int c);              // Функція повертає посилання.
double Array[] = {1.1, 2.2, 3.3, 4.4, 5.5};

int main()
{
      cout << "Ось початкові значення: ";
      for(int i=0; i<5; i++) cout << Array[i] << " ";
      cout << endl;

      ChangeFun(1) = 5298.23;          // Змінюємо значення 2-го елемента масиву.
      ChangeFun(3) = -98.8;            // Змінюємо значення 4-го елемента масиву.

      cout << "Ось змінені значення: ";
      for(int i=0; i<5; i++) cout << Array[i] << " "; cout << endl;

      getch(); return 0;
}

double &ChangeFun(int i)
{
      return Array[i];    // Повертаємо посилання на і-й елемент.
}
```

Ця програма змінює значення другого і четвертого елементів масиву Array. Результати її виконання є такими:

```
Ось початкові значення: 1.1 2.2 3.3 4.4 5.5
Ось змінені значення: 1.1 5298.23 3.3 -98.8 5.5
```

Давайте з'ясуємо, як вони були отримані. Функція ChangeFun() оголошена як та, що повертає посилання на значення типу **double**. Кажучи конкретніше, вона повертає посилання на елемент масиву Array, який задано їй як параметр і. Отож, у процесі виконання такої настанови функції **main**()

```
ChangeFun(1) = 5298.23;              // Змінюємо значення 2-го елемента масиву.
```

функція ChangeFun() повертає посилання на елемент Array[1]. Через це посилання елементові масиву Array[1] тепер присвоюється значення 5298,23. Аналогічні дії відбуваються у процесі виконання і цієї настанови:

```
ChangeFun(3) = -98.8;          // Змінюємо значення 4-го елемента масиву.
```

Оскільки функція ChangeFun() повертає посилання на конкретний елемент масиву Array, то її можна використовувати в лівій частині настанови для присвоєння нового значення відповідному елементу масиву.

Проте, організовуючи повернення функцією посилання, необхідно поклопотатися про те, щоб об'єкт, на який вона посилається, не виходив за межі діючої області видимості. Наприклад, розглянемо таку функцію:

```
// Тут помилка: не можна повертати посилання на локальну змінну.
int &Fun()
{
      int c = 10;
      return c;
}
```





Внаслідок завершення роботи функції Fun() локальна змінна c вийде за межі області видимості. Отже, посилання на змінну c, що повертається функцією Fun(), буде невизначеною. Насправді деякі компілятори не скомпілюють функцію Fun() у такому вигляді, і саме з цієї причини. Проте проблема такого роду може бути створена опосередковано, тому потрібно уважно поставитися до того, на який об'єкт повертатиме посилання Ваша функція.

### 8.2.4. Створення обмеженого (безпечного) масиву

Посилального типу як тип значення, що повертається функцією, можна з успіхом застосувати для створення обмеженого (безпечного) масиву. Як уже зазначалося вище, у процесі виконання C++-коду програми перевірка порушення меж під час індексування масивів не передбачена. Це означає, що може відбутися вихід за межі області пам'яті, виділеної для масиву. Іншими словами, може бути задано індекс, що перевищує розмір масиву. Проте шляхом створення *обмеженого*, або *безпечного* масиву виходу за його межі можна запобігти. Під час роботи з таким масивом будь-який індекс, що виходить за встановлені межі, не допускається для індексування елементів масиву. Один із способів створення безпечного масиву продемонстровано у наведеному нижче коді програми.

**Код програми 8.7. Демонстрація механізму організації безпечного масиву**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

int &put(int i);             // Поміщаємо значення в масив.
int get(int i);              // Зчитуємо значення з масиву.

int Array[10], error = -1;

int main()
{
      put(0) = 10;           // Поміщаємо значення в масив.
      put(1) = 20;
      put(9) = 30;

      cout << get(0) << endl;
      cout << get(1) << endl;
      cout << get(9) << endl;

      // А зараз спеціально генеруємо помилку.
      put(12) = 1; // Індекс за межами масиву.

      getch(); return 0;
}

int &put(int i)    // Функція занесення значення в масив.
```

```
{
        if(i>=0 && i<10)
                return Array[i]; // Повертаємо посилання на і-й елемент.
        else {
                cout << "Помилка порушення меж масиву!" << endl;
                return error; // Повертаємо посилання на error.
        }
}

int get(int i)      // Функція зчитування значення з масиву.
{
        if(i>=0 && i<10)
                return Array[i]; // Повертаємо значення і-го елемента,
        else {
                cout << "Помилка порушення меж масиву!" << endl;
                return error; // Повертаємо значення змінної error.
        }
}
```

Результат, отриманий у процесі виконання цієї програми, має такий вигляд:

```
10 20 30 Помилка порушення меж масиву!
```

У цьому коді програми створюється безпечний масив, призначений для зберігання десяти цілочисельних значень. Щоб помістити в нього значення, використовується функція put(), а щоб прочитати потрібний елемент масиву, викличте функцію get(). Під час використання обох функцій індекс елемента, що Вас цікавить, задається у вигляді аргументу. Як видно з коду програми, функції get() і put() не допускають виходу за межі області пам'яті, виділеної для масиву. Зверніть увагу на те, що функція put() повертає посилання на заданий елемент і тому законно використовується в лівій частині настанови присвоєння.

Незважаючи те, що метод реалізації безпечного масиву, який було представлено в попередній програмі, цілком коректний, проте можливий вдаліший варіант. Як буде показано далі у цьому посібнику (під час перегляду теми перевизначення операторів), програміст може створити власний безпечний масив, під час роботи з яким достатньо використовувати стандартну систему позначень.

### 8.2.5. Поняття про незалежні посилання

Поняття посилання включено у мову програмування C++ в основному для підтримки способу передачі параметрів "за посиланням" і для використання як посилального типу значення, що повертається функцією. Без огляду на це, можна оголосити незалежну змінну посилального типу, яка і називається *незалежним посиланням*. Проте, заради справедливості, необхідно зазначити, що ці незалежні посилальні змінні використовуються досить рідко, оскільки вони можуть "збити з пантелику" Вашу програму. Зробивши (для





очищення совісті) ці зауваження, ми все ж таки можемо приділити незалежним посиланням певну увагу.

*Незалежне посилання – просто ще одна назва для змінних дещо іншого типу.*

Незалежне посилання повинно вказувати на певний об'єкт. Отже, незалежне посилання повинно ініціалізуватися під час її оголошення. У загальному випадку це означає, що їй буде присвоєно адресу раніше оголошеної змінної. Після цього ім'я такої посилальної змінної можна застосовувати скрізь, де може бути використано змінну, на яку вона посилається. І справді, між посиланням і змінною, на яку вона посилається, практично немає ніякої різниці. Розглянемо, наприклад, таку програму:

**Код програми 8.8. Демонстрація механізму використання незалежного посилання**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
        int d, f;
        int &c = d;                  // Незалежне посилання

        d = 10;
        cout << d << " " << c;       // Виводиться: 10 10

        f = 121;
        c = f;              // Копіює в змінну d значення змінної f, а не адресу змінної f.
        cout << endl << d;           // Виводиться значення: 121
        getch(); return 0;
}
```

Внаслідок виконання ця програма виводить такі результати:

```
10 10
121
```

Адреса, яку містить посилальна змінна, є фіксованою і її не можна змінити. Отже, у процесі виконання настанови c = f в змінну d (що адресується посиланням c) копіюється значення змінної f, а не її адреса. Як ще один приклад зазначимо, що після виконання настанови c++ посилальна змінна c не стане містити нову адресу, як це можна було б припустити. У цьому випадку на 1 збільшиться вміст змінної d.

Як було відзначено вище, незалежні посилання краще не використовувати, оскільки найчастіше ним можна знайти заміну, а їх неакуратне застосування може спотворити Ваш програмний код. Погодьтеся: наявність двох імен для однієї і тієї ж самої змінної, по суті, вже створює ситуацію, що потенційно породжує непорозуміння.

### 8.2.6. Врахування обмежень під час використання посилань

На застосування посилальних змінних накладаються такі обмеження:

- не можна посилатися на посилальну змінну;
- не можна створювати масиви посилань;
- не можна створювати покажчик на посилання, тобто не можна до посилання застосовувати оператор "&";
- посилання не дозволено використовувати для бітових полів структур[1].

### 8.2.7. Механізм перевизначення функцій

У цьому розділі ми дізнаємося про одну з найдивовижніших можливостей мови програмування C++ – перевизначення функцій. У мові C++ декілька функцій можуть мати однакові імена, але за умови, що їх параметри будуть різними. Таку особливість мови програмування C++ називають *перевизначенням функцій* (function overloading), а імена функцій, які в ній задіяно, перевизначеними (overloaded) функціями. Перевизначення функцій – один із способів реалізації поліморфізму у мові програмування C++.

***Перевизначення функцій*** *– механізм програмування, який дає змогу двом спорідненим функціям мати однакові імена.*

Розглянемо простий приклад перевизначення функцій.

**Код програми 8.9. Демонстрація механізму "триразового" перевизначення функції Fun()**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

void Fun(int c);             // Один цілочисельний параметр
void Fun(int c, int d);      // Два цілочисельні параметри
void Fun(double f);          // Один параметр типу double

int main()
{
    Fun(10);                 // Виклик функції Fun(int)
    Fun(10, 20);             // Виклик функції Fun(int, int)
    Fun(12.23);              // Виклик функції Fun(double)

    getch(); return 0;
}

void Fun(int c)
{
    cout << "У функції Fun(int) індекс c дорівнює " << c << endl;
}

void Fun(int c, int d)
{
    cout << "У функції Fun(int, int) індекс c дорівнює " << c;
    cout << ", d дорівнює " << d << endl;
}

void Fun(double f)
{
    cout << "У функції Fun(double) індекс f дорівнює " << f << endl;
}
```

Внаслідок виконання ця програма відображає такі результати:

```
У функції Fun(int) індекс c дорівнює 10
У функції Fun(int, int) індекс c дорівнює 10, d дорівнює 20
У функції Fun(double) індекс f дорівнює 12.23
```

Як бачимо, функція Fun() перевантажується три рази. Перша версія приймає один цілочисельний параметр, друга – два цілочисельні параметри, а третя – один **double**-параметр. Оскільки переліки параметрів для всіх трьох версій функцій є різними, то компілятор володіє достатньою інформацією, щоб викликати правильну версію кожної функції. У загальному випадку для створення перевизначення певної функції достатньо оголосити різні її версії.

Для визначення того, яку версію перевизначеної функції викликати, компілятор використовує тип і/або кількість аргументів. Таким чином, перевизначені функції повинні відрізнятися типами і/або кількістю параметрів. Хоча перевизначені методи можуть відрізнятися і типами значень, що повертаються, цього виду інформації недостатньо для мови C++, щоб в усіх випадках компілятор міг вирішити, яку саме функцію потрібно викликати.

Щоб краще зрозуміти виграш від перевизначення функцій, розглянемо три функції із стандартної бібліотеки: **abs**(), **labs**() і **fabs**(). Вони були вперше визначені у мові C, а потім заради сумісності включено у стандарт мови C++. Функція **abs**() повертає абсолютне значення (модуль) цілого числа, функція **labs**() повертає модуль довгого цілочисельного значення (типу **long**), а **fabs**() – модуль значення з плинною крапкою (типу **double**). Оскільки мова C не підтримує перевизначення функцій, то кожна функція повинна мати власне ім'я, хоча всі три функції виконують, по суті, одну і ту саму дію. Це робить ситуацію складнішою, ніж вона є насправді. Іншими словами, при одних і тих самих діях програмісту необхідно пам'ятати імена всіх трьох (у цьому випадку) функцій замість одного. Але у мові програмування C++, як це показано в наведеному нижче прикладі, можна використовувати тільки одне ім'я для всіх трьох функцій.

**Код програми 8.10. Демонстрація механізму створення функцій myAbs() – перевизначеної версії функції abs()**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

// Функція myAbs() перевантажується трьома способами
int myAbs(int c);
```

[1] Бітові поля розглядаються нижче у цьому навчальному посібнику (див. розд. 10.4).

```
double myAbs(double dzm);
long myAbs(long g);

int main()
{
        cout << myAbs(-10) << endl;
        cout << myAbs(-11.0) << endl;
        cout << myAbs(-9L) << endl;

        getch(); return 0;
}
int myAbs(int c)
{
        cout << "Використання int-функції myAbs(): ";
        if(c<0) return -c;
        else return c;
}

double myAbs(double dzm)
{
        cout << "Використання double-функції myAbs(): ";
        if(dzm<0.0) return -dzm;
        else return dzm;
}

long myAbs(long g)
{
        cout << "Використання long-функції myAbs(): ";
        if(g<0) return -g;
        else return g;
}
```

Результати виконання цієї програми є такими:

```
Використання int-функції myAbs(): 10
Використання double-функції myAbs(): 11
Використання long-функції myAbs(): 9
```

Внаслідок виконання ця програма створює три схожі, але все таки різні між собою, функції, які викликаються з використанням "загального" (одного на всіх) імені myAbs. Кожна з них повертає абсолютне значення свого аргументу. В усіх ситуаціях виклику компілятор "знає", яку саме функцію йому використовувати. Для ухвалення рішення йому достатньо "поглянути" на тип аргументу, що передається функції. Принципова значущість перевизначення полягає у тому, що вона дає змогу звертатися до взаємопов'язаних функцій за допомогою одного, загального для всіх, імені. Отже, ім'я myAbs представляє загальну *дію, яка виконується* в усіх випадках. Компіляторові залишається правильно вибрати конкретну версію функції при конкретних обставинах. Завдяки поліморфізму програмісту потрібно пам'ятати не троє різних імен, а тільки одне. Попри простоту наведеного прикладу, він дає змогу зрозуміти,





наскільки механізм перевизначення функцій здатний спростити процес програмування.

Кожна версія перевизначеної функції може виконувати будь-які дії. Іншими словами, не існує правила, яке б зобов'язувало програміста пов'язувати перевизначені функції загальними діями. Проте з погляду стилістики перевизначення функцій все-таки передбачає певну "спорідненість" його версій. Таким чином, незважаючи на те, що одне і те саме ім'я функції можна використовувати для перевизначення не взаємопов'язаних між собою загальними діями функцій, цього робити не варто. Наприклад, у принципі можна використовувати ім'я sqr для створення функції, яка повертає квадрат цілого числа, і функції, яка повертає значення квадратного кореня з дійсного числа (типу **double**). Але, оскільки ці операції фундаментально різні між собою, застосування механізму перевизначення методів у цьому випадку зводить нанівець його первинну мету[1]. На практиці перевантажувати функції має сенс тільки для тісно пов'язаних операцій.

### 8.2.8. Поняття про ключове слово overload

Ще на початку розроблення мови програмування C++ перевизначені функції необхідно було безпосередньо оголошувати такими за допомогою ключового слова **keyword**. Це ключове слово більше не потрібне у стандарті мови програмування C++. Насправді стандартом мови C++ воно навіть не включене в перелік ключових слів. Проте час від часу воно може траплятися у будь-якому C++-коді програми, особливо в старих книгах і статтях.

Загальна форма використання ключового слова **overload** є така:

**overload** *func_name*;

У цьому записі елемент *func_name* є іменем перевантажуваної функції. Ця настанова повинна передувати оголошенням перевизначених функцій[2]. Наприклад, якщо функція Counter() є перевизначеною, то у програму може бути поміщено такий рядок:

**overload** Counter;

Якщо Вам трапиться **overload**-оголошення під час роботи із старими програмами, то їх можна просто видалити: вони більше не потрібні. Оскільки ключове слово **overload** є анахронізмом, то його не варто використовувати в нових C++-програмах. Насправді більшість компіляторів його просто не сприймуть.

## 8.3. Механізм передачі аргументів функції за замовчуванням

У мові програмування C++ ми можемо надати параметру функції певне значення, яке буде автоматично використане, якщо під час виклику функції не задається аргумент, що відповідає цьому параметру. Аргументи, що пере-

[1] Такий стиль програмування, напевно, підійшов би тільки для того, щоб ввести в оману конкурента.
[2] У загальному випадку воно трапляється на початку програми.

даються функції за замовчуванням, можна використовувати для спрощення звернення до складних функцій, а також як "скорочену форму" перевизначення функцій.

Завдання аргументів, що передаються функції за замовчуванням, синтаксично є аналогічним ініціалізації змінних. Розглянемо наведений нижче приклад, у якому оголошується функція myFunc(), що приймає один аргумент типу **double** з діючим за замовчуванням значенням 0.0 і один символьний аргумент з діючим за замовчуванням значенням 'X'.

```
void myFunc(double num = 0.0, char ch = 'X')
{
    . . .
}
```

Після такого оголошення функцію myFunc() можна викликати одним з трьох таких способів:

```
myFunc(198.234, 'A');   // Передаємо безпосередньо задані значення.
myFunc(10.1);           // Передаємо для параметра num значення 10.1,
                        // а для параметра ch дозволяємо застосувати
                        // аргумент, що задається за замовчуванням ('X').
myFunc();               // Для обох параметрів num і ch дозволяємо застосувати
                        // аргументи, що задаються за замовчуванням.
```

Під час першого виклику параметру num передається значення 198.234, а параметру ch – символ 'A'. Під час другого виклику параметру num передається значення 10.1, а параметру ch за замовчуванням встановлюється значення, що дорівнює символу 'X'. Нарешті, внаслідок третього виклику як параметр num, так і параметр ch за замовчуванням встановлюються такими значення, що задаються в оголошенні функції.

Внесення у мову програмування C++ можливості передачі аргументів за замовчуванням дає змогу програмістам спрощувати код програм. Щоб передбачити максимально можливу кількість ситуацій і забезпечити їх коректне оброблення, функції часто оголошуються з великою кількістю параметрів, ніж це потрібно в найбільш поширених випадках. Тому завдяки застосуванню аргументів за замовчуванням програмісту потрібно вказувати не всі аргументи (що використовуються в загальному випадку), а тільки ті, які мають сенс для певної ситуації.

---

*Аргумент, що передається функції за замовчуванням, є значенням, яке буде автоматично передано параметру функції у випадку, якщо аргумент, що відповідає цьому параметру, безпосередньо не задано.*

### 8.3.1. Можливі випадки передачі аргументів функції за замовчуванням

Наскільки корисна можливість передачі аргументів функції за замовчуванням, показано на прикладі функції **clrscr**(), представленої у наведеному нижче коді програми. Функція **clrscr**() очищає екран шляхом виведення послідовності символів нового рядка (це не найефективніший спосіб, але він надзвичайно підходить для цього прикладу). Оскільки в найбільш часто використовуваному режимі представлення відеозображень на екран дисплея виводиться 25 рядків тексту, то як аргумент за замовчуванням використовується значення 25. Але оскільки в інших відеорежимах на екрані може відображатися більше або менше ніж 25 рядків, то аргумент, що діє за замовчуванням, можна перевизначити, безпосередньо вказавши потрібне значення.





**Код програми 8.11. Демонстрація механізму передачі аргументів функції за замовчуванням**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

void clrscr(int size=25);              // Очищаємо 25 рядків.

int main()
{
    for(int i=0; i<30; i++) cout << i << endl;
    clrscr();           // Очищаємо 25 рядків.

    for(int i=0; i<30; i++) cout << i << endl;
    clrscr(10);         // Очищаємо 10 рядків.
    getch(); return 0;
}

void clrscr(int size)
{
    for(; size; size--) cout << endl;
}
```

Як видно з коду цієї програми, якщо значення, що діє за замовчуванням, відповідає ситуації, під час виклику функції **clrscr**() аргумент вказувати не потрібно. Але в інших випадках аргумент, що діє за замовчуванням, можна перевизначити і передати параметру size потрібне значення.

При створенні функцій, що мають значення аргументів, які передаються функції за замовчуванням, потрібно пам'ятати дві речі. Ці значення за замовчуванням мають бути задані тільки один раз, причому під час першого оголошення функції у файлі. У попередньому прикладі аргумент за замовчуванням було задано у прототипі функції **clrscr**(). Під час спроби визначити нові (або навіть ті ж) значення аргументів, що передаються функції за замовчуванням, у визначенні функції **clrscr**() компілятор відобразить повідомлення про помилку і не скомпілює Вашої програми.

Хоча аргументи, які передаються функції, за замовчуванням мають бути визначені тільки один раз, проте для кожної версії перевизначеної функції для передачі за замовчуванням можна задавати різні аргументи. Таким чином, різні версії перевизначеної функції можуть мати різні значення аргументів, що діють за замовчуванням.

Важливо розуміти, що всі параметри, які приймають значення за замовчуванням, мають бути розташовані праворуч від інших. Наприклад, у наведеному нижче прототипі функції міститься помилка:

```
void Fun(int a = 1, int b); // Неправильно!
```

Якщо Ви почали визначати параметри, які приймають значення за замовчуванням, то не можна після них вказувати параметри, що задаються під час виклику функції тільки безпосередньо. Тому наведене нижче оголошення є також неправильним і тому воно не буде скомпільовано:

```
int myFunc(float f, char *str, int c=10, int d); // Неправильно!
```

Оскільки для параметра c визначено значення за замовчуванням, то для параметра d також потрібно задати значення за замовчуванням.

### 8.3.2. Порівняння можливості передачі аргументів функції за замовчуванням з її перевизначенням

Як згадувалося на початку цього розділу, одним із застосувань передачі аргументів функції за замовчуванням є "скороченою формою" перевизначення функцій. Щоб зрозуміти це, уявіть, що Вам потрібно створити дві "адаптовані" версії стандартної функції **strcpy**(). Одна версія повинна приєднувати весь вміст одного рядка до кінця іншої. Друга ж приймає третій аргумент, який задає кількість символів, що конкатенуються (приєднуються). Іншими словами, ця версія повинна конкатенувати тільки задану кількість символів одного рядка до кінця іншої.

Допустимо, що Ви назвали свої функції іменем myStrcpy() і запропонували такий варіант їх прототипів:

```
void myStrcpy(char *s1, char *s2, int len);
void myStrcpy(char *s1, char *s2);
```

Перша версія повинна скопіювати len символів з рядка s2 до кінця рядка s1. Друга версія копіює весь рядок, який адресується покажчиком s2, у кінець рядка, який адресується покажчиком s1, тобто діє подібно до стандартної функції **strcpy**().

Попри те, що для досягнення поставленої мети можна реалізувати дві версії функції myStrcpy(), є простіший спосіб розв'язання цієї задачі. Використовуючи можливість передачі аргументів функції за замовчуванням, можна створити тільки одну і ту саму функцію myStrcpy(), яка замінить обидві задумані її версії. Реалізація цієї ідеї продемонстрована у наведеному нижче коді програми.

**Код програми 8.12. Демонстрація механізму передачі аргументів функції за замовчуванням з її перевизначенням**

```
#include <iostream>          // Потокове введення-виведення
#include <cstring>           // Робота з рядковими типами даних
using namespace std;         // Використання стандартного простору імен

void myStrcpy(char *s1, char *s2, int len = -1);

int main()
{
    char str1[80] = "Це тест.";
    char str2[80] = "0123456789";
```





```
    myStrcpy(str1, str2, 5); // Приєднуємо 5 символів.
    cout << str1 << endl;
    strcpy(str1, "Це тест."); // Відновлюємо str1.

    myStrcpy(str1, str2); // Приєднуємо весь рядок.
    cout << str1 << endl;

    getch(); return 0;
}
// Призначена для користувача версія функції strcpy().
void myStrcpy(char *s1, char *s2, int len)
{
    while(*s1) s1++;  // Знаходимо кінець рядка s1.

    if(len == -1) len = strlen(s2);

    while(*s2 && len) {
        *s1 = *s2;          // Копіюємо символи.
        s1++; s2++; len--;
    }
    *sl = '\0';             // Завершуємо рядок s1 нульовим символом.
}
```

У цьому коді програми функція myStrcpy() приєднує len символів рядка, який адресується покажчиком s2, до кінця рядка, який адресується покажчиком s1. Але, якщо значення len дорівнює -1, як і у разі дозволу передачі цього аргументу функції за замовчуванням, то функція myStrcpy() приєднає до рядка s1 весь рядок, який адресується покажчиком s2. Іншими словами, якщо значення len дорівнює -1, то функція myStrcpy() діє подібно до стандартної функції **strcpy**(). Якщо використовувати для параметра len можливість передачі аргументу функції за замовчуванням, то обидві операції можна об'єднати в одній функції.

Цей приклад дав змогу продемонструвати, як аргументи, що передаються функції за замовчуванням, забезпечують основу для скороченої форми оголошення перевизначених функцій.

### 8.3.3. Механізм використання аргументів, що передаються функції за замовчуванням

Хоча передача аргументів функції за замовчуванням, є дуже потужним засобом програмування (за умови їх коректного використання), з ними можуть іноді виникати проблеми. Їх призначення дає змогу функції ефективно виконувати свою роботу, забезпечуючи при всій простоті цього механізму значну гнучкість. У цьому сенсі всі передані аргументи функції за замовчуванням повинні відображати спосіб найбільш загального використання функції або альтернативного її застосування. Якщо не існує деякого єдиного значення, яке звичайно присвоюється тому або іншому параметру, то і немає се-

нсу оголошувати відповідний аргумент за замовчуванням. Насправді оголошення аргументів, що передаються функції за замовчуванням, при недостатній для цього підставі деструктуризує програмний код, оскільки такі аргументи здатні збити з пантелику будь-кого, кому доведеться розбиратися в такій програмі. Нарешті, основним принципом використання аргументів за замовчуванням повинен бути, як у лікарів, принцип "не нашкодити". Іншими словами, випадкове використання аргументу функції за замовчуванням не повинно призвести до незворотних негативних наслідків. Адже такий аргумент можна просто забути вказати під час виклику певної функції, і, якщо це трапиться, то подібний промах не повинен викликати, наприклад, втрату важливих даних!

## 8.4. Перевизначення функцій і неоднозначності, що при цьому виникають

Перш ніж завершити цей розділ, ми повинні дослідити вид помилок, що є унікальними для мови програмування C++: неоднозначності. Можливі ситуації, у яких компілятор не здатний зробити вибір між двома (або більше) коректно перевизначеними функціями. Такі ситуації називають *неоднозначними*. Настанови, що створюють неоднозначності, є помилковими, а програми, які їх містять, не будуть скомпільовані.

*Неоднозначності виникають тоді, коли компілятор не може визначити відмінність між двома перевизначеними функціями.*

Основною причиною неоднозначності у мові програмування C++ є автоматичне перетворення типів. У мові програмування C++ робиться спроба автоматично перетворити тип аргументів, що використовуються для виклику функції, в тип параметрів, визначених функцією. Розглянемо такий приклад:

```
int myFunc(double d);
. . .
cout << myFunc('c'); // Помилки немає, здійснюється перетворення типів
```

Як зазначено в коментарі, помилки тут немає, оскільки мова C++ автоматично перетворить символ 'c' в його **double**-еквівалент. Власне кажучи, у мові програмування C++ заборонено достатньо мало видів перетворень типів. Незважаючи на те, що автоматичне перетворення типів – дуже зручно, воно, проте, є головною причиною неоднозначності. Розглянемо таку програму.

**Код програми 8.13. Демонстрація появи неоднозначності внаслідок перевизначення функцій**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

float myFunc(float c)        { return c; }
double myFunc(double c)      { return -c;}
```





```
int main()
{       // Неоднозначності немає, викликається функція myFunc(double)
    cout << myFunc(10.1) << " ";
    cout << myFunc(10);      // Виникнення неоднозначності.
    getch(); return 0;
}
```

У цьому коді програми завдяки перевизначенню функція myFunc() може приймати аргументи або типу **float**, або типу **double**. У процесі виконання рядка коду програми

```
cout << myFunc(10.1) << " ";
```

не виникає ніякої неоднозначності: компілятор мови C++ "впевнено" забезпечує виклик функції myFunc(**double**), оскільки, якщо не задано безпосередньо інше, всі літерали з плинною крапкою у мові програмування C++ автоматично отримують тип **double**. Але під час виклику функції myFunc() з аргументом, що дорівнює цілому числу 10, у програму вноситься неоднозначність, оскільки компіляторові невідомо, в який тип йому необхідно перетворити цей аргумент: **float** або **double**. Обидва перетворення є допустимими. У такій неоднозначній ситуації буде видано повідомлення про помилку, і програма не буде компільована. На прикладі попередньої програми хотілося б наголосити, що неоднозначність в ній викликана не перевизначенням функції myFunc(), оголошеної двічі для прийому **double**- і **float**-аргументу, а використанням при конкретному виклику функції myFunc() аргументу невизначеного для перетворення типу. Іншими словами, помилка полягає не в перевизначенні функції myFunc(), а в конкретному її виклику.

А ось ще один приклад неоднозначності, викликаної автоматичним перетворенням типів у мові програмування C++.

**Код програми 8.14. Демонстрація появи помилки, спричиненої неоднозначністю**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

char myFunc(unsigned char ch)    { return ch-1; }
char myFunc(char ch)             { return ch+1; }

int main()
{
    cout << myFunc('c');         // Тут викликається myFunc(char).
    cout << myFunc(88) << " ";   // Вноситься неоднозначність.

    getch(); return 0;
}
```

У мові програмування C++ типи **unsigned char** і **char** не є істотно неоднозначними, позаяк це різні типи. Але під час виклику функції myFunc() з цілочисельним аргументом 88 компілятор "не знає", яку функцію йому виконати,

тобто в значення якого типу йому необхідно перетворити число 88: типу **char** або типу **unsigned char**? Обидва перетворення тут цілком допускаються.

Неоднозначність може бути також викликана використанням в перевизначених функціях аргументів, що передаються функції за замовчуванням. Для прикладу розглянемо таку програму.

**Код програми 8.15. Демонстрація ще одного прикладу неоднозначності**

```
#include <iostream>            // Потокове введення-виведення
#include <conio>               // Для консольного режиму роботи
using namespace std;           // Використання стандартного простору імен

int myFunc(int c)              { return c; }
int myFunc(int c, int d=1)     { return c*d; }

int main()
{
      cout << myFunc(4, 5) << " ";     // Неоднозначності немає
      cout << myFunc(10);              // Виникнення неоднозначності

      getch(); return 0;
}
```

У цьому коді програми в першому зверненні до функції myFunc() задається два аргументи, тому у компілятора немає ніяких сумнівів у виборі потрібної функції, а саме myFunc(**int** c, **int** d), тобто ніякої неоднозначності у цьому випадку не виникає. Але під час другого звернення до функції myFunc() ми отримуємо неоднозначність, оскільки компілятор "не знає", чи то йому викликати версію функції myFunc(), яка приймає один аргумент, чи то використовувати можливість передачі аргументу функції за замовчуванням до версії, яка приймає два аргументи.

Програмуючи мовою C++, Вам ще не раз доведеться наштовхнутися на помилки неоднозначності, які, шкода, але дуже легко проникають у програми, і тільки досвід та практика допоможуть Вам виправляти їх.





# Розділ 9. C++-СПЕЦИФІКАТОРИ ТА СПЕЦІАЛЬНІ ОПЕРАТОРИ

Перш ніж переходити до перегляду складніших засобів мови програмування C++, є сенс ґрунтовніше познайомитися з C++-специфікаторами типів даних і класів пам'яті, а також з деякими спеціальними операторами. Окрім вже розглянутих вище типів даних, у мові програмування C++ визначено ще й інші. Одні з них складаються з модифікаторів, що додаються до вже відомих типів, інші містять перерахунки, а треті використовують ключове слово **typedef**. Мова C++ також підтримує ряд операторів, які значно розширюють область дії мови і дають змогу розв'язувати задачі програмування у надзвичайно широкому діапазоні. Йдеться про порозрядні оператори, оператори зсуву, а також оператори "**?**" і **sizeof**. Окрім цього, у цьому розділі розглядаються такі спеціальні оператори, як **new** і **delete**. Вони призначені для підтримки C++-системи динамічного розподілу пам'яті.

## 9.1. Специфікатори типів даних

У мові програмування C++ визначено два специфікатори типів даних – **const** і **volatile**, котрі роблять вплив на те, як можна отримати доступ до змінних або модифікувати їх. Офіційно їх іменують ***cv-специфікаторами*** і вони мають передувати базовому типу під час оголошення змінної.

*Специфікатори типів даних **const** і **volatile** керують доступом до змінної.*

### 9.1.1. Застосування специфікатора типу даних const

Змінні, які оголошено з використанням специфікатора типу даних **const**, не можуть змінити свої значення у процесі виконання програми. Проте будь-якій **const**-змінній можна присвоїти деяке початкове значення. Наприклад, у процесі виконання такої настанови

```
const double val = 3.2;
```

створюється **double**-змінна val, яка містить значення 3.2, і це значення у процесі виконання програми змінити вже не можна. Але цю змінну можна використовувати в інших виразах. Будь-яка **const**-змінна набуває значення або під час ініціалізації, що задається безпосередньо, або під час використання апаратно-залежних засобів. Застосування специфікатора типу даних **const** до оголошення змінної гарантує, що вона не буде модифікована іншими частинами Вашої програми.

*Специфікатор типу даних **const** запобігає модифікації змінної у процесі виконання програми.*

Специфікатор типу даних **const** має ряд важливих застосувань. Можливо, частіше за все його використовують для створення **const**-параметрів типу покажчика. Такий параметр-покажчик захищає об'єкт, на який він посилається, від модифікації з боку функції. Іншими словами, якщо параметр-покажчик передує ключовому слову **const**, то ніяка настанова цієї функції не може модифікувати змінну, яка адресується цим параметром. Наприклад, функція **code**() у наведеній нижче короткій програмі зсовує кожну букву в повідомленні на одну алфавітну позицію (тобто замість букви "А" ставиться буква "Б" і т.д.), відображаючи таким чином повідомлення в закодованому вигляді. Використання специфікатора типу даних **const** в оголошенні параметра не дає змоги коду функції модифікувати об'єкт, на який вказує цей параметр.

**Код програми 9.1. Демонстрація зсуву букв у повідомленні на одну алфавітну позицію**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void code(const char *str);
int main()
{
      code("Це тест.");      // Зсув букви на одну алфавітну позицію
      getch(); return 0;
}

/* Використання специфікатора типу даних const гарантує, що str
не може змінити аргумент, на який він вказує. */
void code(const char *str)
{
      while(*str) {
            cout << (char) (*str+1);
            str++;
      }
}
```

Оскільки параметр str оголошується як **const**-покажчик, то у функції **code**() немає ніякої можливості внести зміни в рядок, який адресується покажчиком str. Але, якщо Ви спробуєте написати функцію **code**() так, як це показано в наведеному нижче прикладі, то обов'язково отримаєте повідомлення про помилку і програма не буде компільована.

```
// Демонстрація роботи неправильного коду програми
void code(const char *str)
{
      while(*str) {
            *str = *str + 1; // Помилка, аргумент модифікувати не можна
            cout << (char) *str;
            str++;
      }
}
```





Оскільки параметр str є **const**-покажчиком, то його не можна використовувати для модифікації об'єкта, на який він посилається.

Специфікатор типу даних **const** можна також використовувати для посилальних параметрів, щоб не допустити у функції модифікацію змінних, на які посилаються ці параметри. Наприклад, наведений нижче код програми є некоректним, оскільки функція Fun() намагається модифікувати змінну, на яку посилається параметр c.

**Код програми 9.2. Демонстрація неможливості модифікувати const-посилання**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

void Fun(const int &c);

int main()
{
      int f = 10;
      Fun(f);

      getch(); return 0;
}

// Використання посилального const-параметра.
void Fun(const int &c)
{
      c = 100; // Помилка, не можна модифікувати const-посилання.
      cout << c;
}
```

Специфікатор типу даних **const** ще можна використовувати для підтвердження того, що Ваша програма не змінює значення певної змінної. Пригадайте, що змінна типу **const** може бути модифікована зовнішніми пристроями, тобто її значення може бути встановлено будь-яким апаратним пристроєм (наприклад, давачем). Оголосивши змінну за допомогою специфікатора типу даних **const**, можна довести, що будь-які зміни, яким піддається ця змінна, викликані виключно зовнішніми діями.

Нарешті, специфікатор типу даних **const** використовують для створення іменованих констант. Часто у програмах багато разів застосовується одне і те саме значення для різних потреб. Наприклад, необхідно оголосити декілька різних масивів так, щоб усі вони мали однаковий розмір. Коли потрібно використовувати таке "магічне число", то є сенс реалізувати його у вигляді **const**-змінної. Потім замість реального значення можна використовувати ім'я цієї змінної, а якщо це значення доведеться згодом змінити, то Ви зміните його тільки в одному місці програми. Наведений нижче код програми дає змогу реалізувати застосування специфікатора типу даних **const** "на смак".

```
// Використання специфікатора типу даних const для створення іменованих констант
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен
```

```
const int size = 10;

int main()
{
    int A1[size], A2[size], A3[size];
    //...
}
```

Якщо у наведеному прикладі знадобиться використовувати новий розмір для масивів, то Вам потрібно буде змінити тільки оголошення змінної size і перекомпілювати програму. Як наслідок, всі три масиви автоматично отримають новий розмір.

### 9.1.2. Застосування специфікатора типу даних volatile

Специфікатор типу даних **volatile** повідомляє компіляторові про те, що значення відповідної змінної може бути змінене у програмі опосередковано. Наприклад, адреса певної глобальної змінної може передаватися керованій змінній перериваннями підпрограмі трактування, яка обновляє цю змінну з приходом кожного імпульсу сигналу часу. У такій ситуації вміст змінної змінюється без використання безпосередньо заданих настанов програми. Існують вагомі підстави для того, щоби повідомити компілятор про зовнішні чинники зміни змінної. Йдеться про те, що C++-компіляторові дозволяється автоматично оптимізувати певні вирази в припущенні, що вміст тієї або іншої змінної залишається незмінним, якщо воно не знаходиться в лівій частині настанови присвоєння. Але, якщо деякі чинники (зовнішні стосовно програми) змінять значення цього поля, то таке припущення виявиться неправильним, внаслідок чого можуть виникнути проблеми.

---

*Специфікатор типу даних **volatile** інформує компілятор про те, що ця змінна може бути змінена зовнішніми (стосовно програми) чинниками.*

Наприклад, у наведеному нижче коді програми припустимо, що змінна chas обновляється кожну мілісекунду годинниковим механізмом комп'ютера. Але, оскільки змінна chas не оголошена з використанням специфікатора типу даних **volatile**, цей фрагмент коду програми може іноді працювати неналежним чином. Зверніть особливу увагу на рядки, позначені буквами "А" і "Б":

```
int chas, timerClass;
// ...

timerClass = chas; // Рядок А
//... Виконання будь-яких дій.

cout << "Пройдений час " << chas-timerClass; // Рядок Б
```

У цьому коді програми змінна chas набуває свого значення, коли вона присвоюється змінній timerClass у рядку А. Але, оскільки змінна chas не оголошена з використанням специфікатора типу даних **volatile**, то компілятор має можливість оптимізувати цей код, причому у такий спосіб, при якому значення змінної chas, можливо, не буде запитано в настанові **cout** (рядок Б),





якщо між рядками А і Б не буде жодного проміжного присвоєння значення змінної chas. Іншими словами, у рядку Б компілятор може просто ще раз використовувати значення, яке отримала змінна chas у рядку А. Але, якщо між моментами виконання рядків А і Б надійдуть чергові імпульси сигналу часу, то значення змінної chas обов'язково зміниться, а рядок Б у цьому випадку не відобразить коректного результату.

Для вирішення цього питання необхідно визначити змінну chas з ключовим словом **volatile**:

```
volatile int chas;
```

Тепер значення змінної chas запитуватиметься під час кожного її використанні. І хоча на перший погляд це може видатися дивним, проте специфікатори типів даних **const** і **volatile** можна використовувати разом. Наприклад, наведене нижче оголошення абсолютно допустиме. Воно створює **const**-покажчик на **volatile**-об'єкт:

```
const volatile unsigned char *port = ( const volatile char *) 0x2112;
```

У наведеному вище прикладі для перетворення цілочисельного літерала 0x2112 в **const**-покажчик на **volatile**-символ необхідно застосувати операцію приведення типів.

## 9.2. Поняття про специфікатори класів пам'яті

Мова програмування C++ підтримує п'ять специфікаторів класів пам'яті:
**auto**, **extern**, **register**, **static**, **mutable**

За допомогою цих ключових слів компілятор отримує інформацію про те, як повинна зберігатися змінна. Специфікатор класів пам'яті необхідно вказувати на початку оголошення змінної.

---

*Специфікатори класів пам'яті визначають, як повинна зберігатися змінна.*

Специфікатор **mutable** застосовується тільки до об'єктів класів, про які йтиметься попереду. Решту специфікаторів ми розглянемо у цьому розділі.

### 9.2.1. Застосування специфікатора класу пам'яті auto

Специфікатор класу пам'яті **auto** оголошує локальну змінну. Але він використовується досить рідко (можливо, Вам ніколи і не доведеться застосувати його), оскільки локальні змінні є "автоматичними" за замовчуванням. Навряд чи Вам трапиться це ключове слово і в чужих програмах.

---

*Рідко використовуваний специфікатор класу пам'яті **auto** оголошує локальну змінну.*

### 9.2.2. Застосування специфікатора класу пам'яті extern

Всі програми, які ми розглядали дотепер, мали достатньо скромний розмір. Реальні ж комп'ютерні програми є значно більшими. У міру збільшення розміру файлу, що містить програму, тривалість її компілювання стає іноді дратівливо довгою. У цьому випадку необхідно поділити програму на декілька окремих файлів. Після цього невеликі зміни, що вносяться в один файл, не призведуть до перекомпілювання всієї програми. У процесі розроблення великих проектів такий багатофайловий підхід може заощадити істотний час. Реалізувати цей підхід дає змогу ключове слово **extern**.

*Специфікатор класу пам'яті* ***extern*** *оголошує змінну, але не виділяє для неї області пам'яті.*

У програмах, які складаються з двох або більше файлів, кожен файл повинен "знати" імена і типи глобальних змінних, що використовуються програмою в цілому. Проте не можна просто оголосити копії глобальних змінних у кожному файлі. Йдеться про те, що у мові програмування C++ програма може містити тільки одну копію кожної глобальної змінної. Отже, якщо Ви спробуєте оголосити необхідні глобальні змінні у кожному файлі, то виникнуть проблеми. Коли компонувальник спробує скомпонувати ці файли, то він виявить дубльовані глобальні змінні, і компонування програми не відбудеться. Щоб вийти з цього скрутного становища, достатньо оголосити всі глобальні змінні в одному файлі, а в інших використовувати extern-оголошення, як це показано в табл. 9.1.

**Табл. 9.1. Механізм використання глобальних змінних у окремо компільованих модулях**

| Файл F1 | Файл F2 |
|---|---|
| **int** x, y;<br>**char** ch;<br><br>**int main**()<br>{<br>//...<br>}<br>**void** func1()<br>{<br>x = 123;<br>} | **extern int** x, y;<br>**extern char** ch;<br><br>**void** func22()<br>{<br>x = y/10;<br>}<br><br>**void** func23()<br>{<br>y = 10;<br>} |

У файлі F1 оголошуються та визначаються змінні x, y і ch. У файлі F2 використовується скопійований з файлу F1 перелік глобальних змінних, до оголошення яких додано ключове слово **extern**. Специфікатор класу пам'яті **extern** робить змінну відомою для модуля, але насправді не створює її. Іншими словами, ключове слово **extern** надає компіляторові інформацію про тип і ім'я глобальних змінних, повторно не виділяючи для них пам'яті. Під час компонування цих двох модулів усі посилання на ці зовнішні змінні будуть визначені.





***Нео! хідноапам'ятати!*** *Дотепер ми не уточнювали, у чому полягає відмінність між оголошенням і визначенням змінної, але тут це надзвичайно важливо.* ***Під ч с оголоше- - я ' мі- - ої їй присвоюєтьсяаі, 'яаіатип,ааа' аа допо, огоюави' наченняадляа' , інноїавиділяєтьсяатількиапа, 'ять****. Здебільшого оголошення змінних одночасно є їх визначеннями. Якщо імені змінної передує специфікатор класу пам'яті* ***extern****, то можна оголосити змінну, не визначаючи її.*

Існує ще одне застосування для ключового слова **extern**, яке не пов'язане з багатофайловими проектами. Не секрет, що багато часу йде на оголошення глобальних змінних, які, як правило, наводяться на початку програми, але це не завжди є обов'язковим. Якщо функція використовує глобальну змінну, яка визначається нижче (у тому ж файлі), то в тілі функції її можна специфікувати як зовнішню (за допомогою ключового слова **extern**). Внаслідок виявлення визначення цієї змінної компілятор обчислить відповідні посилання на неї.

Для розуміння сказаного розглянемо такий приклад. Зверніть увагу на те, що глобальні змінні first і last оголошуються не перед, а після основної функції **main**().

**Код програми 9.3. Демонстрація механізму застосування специфікатора класу пам'яті extern**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    extern int first, last;      // Використання глобальних змінних.
    cout << first << " " << last << endl;
    getch(); return 0;
}

// Глобальне визначення змінних first і last.
int first = 10, last = 20;
```

У процесі виконання цієї програми на екран буде виведено числа 10 20, оскільки глобальні змінні first і last, що використовуються в настанові **cout**, ініціалізувалися цими значеннями. Оскільки **extern**-оголошення у функції **main**() повідомляє компіляторові про те, що змінні first і last оголошуються десь у іншому місці (у цьому випадку нижче, але у тому ж файлі), програму можна скомпілювати без помилок, незважаючи на те, що змінні first і last використовуються до їх визначення.

Важливо розуміти, що **extern**-оголошення змінних, показані в попередній програмі, необхідні тут тільки з тієї причини, що змінні first і last не були визначені до їх використання у функції **main**(). Якби їх визначення компілятор виявив раніше від початку визначення функції **main**(), то потреби в **extern**-настанові не було б.

***Нео! хідноапа, 'ятати!*** *Якщо компілятор виявляє змінну, яка не була оголошена в поточному блоці, то він перевіряє, чи не збігається вона з*

*якою-небудь із змінних, оголошених усередині інших блоків. Якщо ні, то компілятор проглядає раніше оголошені глобальні змінні. Якщо виявляється збіг їх імен, то компілятор припускає, що посилання було саме на цю глобальну змінну. Специфікатор класу пам'яті* ***extern*** *є необхідним тільки у тому випадку, якщо виникає бажання використовувати змінну, яка оголошується або нижче у тому ж файлі, або в іншому.*

***Вартоа' нати!*** *Незважаючи на те, що специфікатор класу пам'яті* ***extern*** *оголошує, але не визначає змінну, то існує один виняток з цього правила. Якщо в* ***extern****-визначенні змінна ініціалізувалася, то таке* ***extern****-оголошення стає визначенням. Це дуже важливий момент, оскільки будь-який об'єкт може мати декілька оголошень, але тільки одне визначення.*

### 9.2.3. Статичні змінні

Статичні змінні типу **static** – змінні "довготривалого" зберігання, тобто вони зберігають свої значення у межах своєї функції або файлу. Від глобальних вони відрізняються тим, що за рамками своєї функції або файлу вони невідомі. Оскільки специфікатор класу пам'яті **static** по-різному визначає "долю" локальних і глобальних змінних, то ми розглянемо їх окремо.

#### Локальні static-змінні

Якщо до локальної змінної застосовано модифікатор **static**, то для неї виділяють постійну область пам'яті практично так само, як і для глобальної змінної. Це дає змогу статичній змінній підтримувати її значення між викликами функцій. Іншими словами, на відміну від звичайної локальної змінної, значення static-змінної не втрачається при виході з функції. Ключова відмінність між статичною локальною і глобальною змінними полягає у тому, що статична локальна змінна відома тільки блоку, у якому вона оголошена. Таким чином, статичну локальну змінну деякою мірою можна назвати глобальною змінною, яка має обмежену область видимості.

*Локальна* ***static****-змінна підтримує своє значення між викликами функції.*

Щоб оголосити статичну змінну, достатньо, щоб її типу передувало ключове слово **static**. Наприклад, у процесі виконання цієї настанови змінна *count* оголошується статичною:

```
static int count;
```

Статичній змінній можна присвоїти початкове значення. Наприклад, у цій настанові змінній *count* присвоюється початкове значення 200:

```
static int count = 200;
```

Локальні **static**-змінні ініціалізувалися тільки одного разу, на початку виконання програми, а не під час кожного вході у функцію, у якій вони оголошені.

Можливість використання статичних локальних змінних важлива для створення незалежних функцій, оскільки існують такі типи функцій, які повинні зберігати їх значення між викликами. Якби статичні змінні не були передбачені мовою програмування C++, то довелося б використовувати замість них глобальні, що відкрило б шлях для усяких побічних ефектів.





Розглянемо приклад використання модифікатора **static**-змінної. Вона слугує для зберігання поточного середнього значення від чисел, що вводяться користувачем.

**Код програми 9.4. Демонстрація механізму використання локальних static-змінних**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int stVal(int c); // Обчислення поточного середнього значення.

int main()
{
      int num;
      do {
            cout << "Введіть число (-1 означає вихід): "; cin >> num;
            if(num != -1)
            cout << "Поточне середнє дорівнює: " << stVal(num);
            cout << endl;
      } while(num > -1);

      getch(); return 0;
}

int stVal(int c) // Обчислення поточного середнього значення.
{
      static int sum = 0, count = 0;
      sum = sum + c;
      count++;
      return sum / count;
}
```

У цьому коді програми обидві локальні змінні sum і *count* оголошено статичними, тобто вони були ініціалізовані значенням 0.

***Нео! хідноапа, 'ятати!*** *Для статичних змінних ініціалізація здійснюється тільки один раз (під час першого виконання функції), а не під час кожного вході у функцію.*

У цьому коді програми функцію stVal() використовують для обчислення поточного середнього значення від чисел, що вводяться користувачем. Оскільки обидві змінні sum і *count* є статичними, вони підтримують свої значення між викликами функції stVal(), що дає змогу нам отримати правильний результат обчислень. Щоб переконатися в потребі модифікатора **static**, спробуйте видалити його з програми. Після цього програма не працюватиме коректно, оскільки проміжна сума втрачатиметься під час кожного виході з функції stVal().

### Глобальні static-змінні

Якщо модифікатор **static** застосовується до глобальної змінної, то компілятор створить глобальну змінну, яка буде відома тільки для файлу, у якому вона оголошена. Це означає, що, хоча ця змінна є глобальною, проте інші функції в інших файлах не мають про неї "ані найменшого поняття" і не можуть змінити її вміст. Тому вона і не може стати "жертвою" несанкціонованих змін. Отже, для особливих ситуацій, коли локальна статичність виявляється безсилою, то можна створити невеликий файл, який міститиме тільки функції, які використовують глобальні **static**-змінні, окремо скомпілювати цей файл і працювати з ним, не побоюючись шкоди від побічних ефектів "загальної глобальності".

*Глобальна **static**-змінна відома тільки для файлу, у якому вона оголошена.*

Розглянемо приклад, який є переробленою версією програми (з попереднього підрозділу), що обчислює поточне середнє значення. Ця версія складається з двох файлів і використовує глобальні **static**-змінні для зберігання значень проміжної суми і лічильника чисел, що вводяться.

**Код програми 9.5. Демонстрація механізму використання глобальних static-змінних**

```
// ------------------Перший файл------------------
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int stVal(int c);            // Обчислення поточного середнього значення.
void Reset();                // Очищення глобальних змінних.

int main()
{
      int num;
      do {
            cout << "Введіть число (-1 для виходу, -2 для скидання): "; cin >> num;
            if(num==-2) {
                  Reset();
                  continue;
            }
            if Fun(num != -1)
                  cout << "Середнє значення дорівнює: " << stVal(num);
            cout << endl;
      } while(num != -1);

      getch(); return 0;
}
// ------------------Другий файл------------------

static int sum=0, count = 0;

int stVal(int c)   // Обчислення поточного середнього значення.
{
      sum = sum + c; count++;
      return sum / count;
}

void Reset()      // Очищення глобальних змінних.
{
      sum = 0; count = 0;
}
```

У цій версії програми змінні sum і *count* є глобально статичними, тобто їх глобальність обмежена другим файлом. Отже, вони використовуються функціями stVal() і Reset(), причому обидві вони розташовані в другому файлі. Цей варіант програми дає змогу скидати нагромаджену суму (шляхом встановлення в початкове положення змінних sum і *count*), щоб можна було усереднити інший набір чисел. Але жодна з функцій, розташованих поза другим файлом, не може отримати доступ до цих змінних. Працюючи з цією програмою, можна очистити попередні нагромадження, ввівши число -2. У цьому випадку буде викликано функцію Reset(). Окрім цього, спробуйте отримати з першого файлу доступ до будь-якої із змінних sum або *count*[1].

Отже, ім'я локальної **static**-змінної відоме тільки функції або блоку коду програми, у якому вона оголошена, а ім'я глобальної **static**-змінної – тільки файлової, у якому вона "мешкає". По суті, модифікатор **static** дає змогу змінним існувати так, що про них знають тільки функції, які використовують їх, тим самим "тримаючи у шорах" і обмежуючи можливості негативних побічних ефектів. Змінні типу **static** дають змогу програмісту "приховувати" одні частини своєї програми від інших. Це може виявитися просто суперперевагою, коли Вам доведеться розробляти дуже велику і складну програму.

***Вартоа' нати!а****Попри те, що глобальні **static**-змінні, як і раніше, допускаються і широко використовуються в C++-програмі, стандарт мови C++ заперечує проти їх застосування. Для керування доступом до глобальних змінних рекомендуємо інший метод, який полягає у використанні просторів імен.*

### 9.2.4. Регістрові змінні

Зазвичай найчастіше використовується специфікатор класу пам'яті **register**. Для компілятора модифікатор **register** означає розпорядження забезпечити таке зберігання відповідної змінної, щоб доступ до неї можна було отримати максимально швидко. Зазвичай змінна у цьому випадку зберігатиметься або в регістрі центрального процесора (ЦП), або в кеші (швидкодіючій буферній пам'яті невеликої місткості). Ймовірно, Ви знаєте, що доступ до регістрів ЦП (або до кеш-пам'яті) принципово швидший, ніж доступ до осно-

[1] Ви отримаєте повідомлення про помилку.

вної пам'яті комп'ютера. Таким чином, змінну, що зберігається в регістрі, буде обслужене набагато швидше, ніж змінну, що зберігається, наприклад, в оперативній пам'яті (ОЗП). Оскільки швидкість, з якою до змінних можна отримати доступ, визначає, по суті, швидкість виконання Вашої програми, то для отримання задовільних результатів програмування важливо розумно використовувати специфікатор класу пам'яті **register**.

*Специфікатор класу пам'яті **register** у визначенні змінної означає вимогу оптимізувати код для отримання максимально можливої швидкості доступу до неї.*

Формально специфікатор класу пам'яті **register** є тільки запитом, який компілятор має право проігнорувати. Це легко пояснити: адже кількість регістрів (або пристроїв пам'яті з малим часом вибірки) обмежена, причому для різних середовищ вона може бути різною. Тому, якщо компілятор вичерпає пам'ять швидкого доступу, то він зберігатиме **register**-змінні звичайним способом. У загальному випадку незадоволений **register**-запит не зашкодить, але, звичайно ж, і не дає ніяких переваг зберігання в регістровій пам'яті.

Оскільки насправді тільки для обмеженої кількості змінних можна забезпечити швидкий доступ, то важливо ретельно вибрати, до яких з них застосувати модифікатор **register**[1]. Як правило, ніж частіше до змінної потрібен доступ, тим більша вигода буде отримана внаслідок оптимізації коду програми за допомогою специфікатора класу пам'яті **register**. Тому оголошувати регістровими має сенс змінні параметри циклу, або змінні, до яких здійснюється доступ в тілі самого циклу. На прикладі наведеної нижче функції показано, як **register**-змінна типу **int** використовують для керування циклом. Ця функція обчислює результат виразу $\mathbf{m}^{\mathbf{e}}$ для цілочисельних значень із збереженням знаку початкового числа (тобто при **m** = -2 і **e** = 2 результат буде дорівнювати -4).

```
int signed_pwr(register int m, register int e)
{
    register int tmp;
    int znak;

    if(m < 0) znak = -1;
    else znak = 1;

    tmp = 1;
    for( ; e; e--) tmp = tmp * m;

    return tmp * znak;
}
```

У наведеному прикладі змінні m, e і tmp оголошені як регістрові, оскільки всі вони використовуються в тілі циклу, і тому до них часто здійснюється доступ. Проте змінна znak оголошена без специфікатора класу пам'яті **register**, оскільки вона не є частиною циклу і використовується рідше.

[1] Тільки правильний вибір може підвищити швидкодію програми.





### 9.2.5. Походження модифікатора register

Модифікатор **register** був вперше визначений у мові C. Спочатку він застосовувався тільки до змінних типу **int** і **char** або до покажчиків і примушував зберігати змінні цього типу в регістрі цифрового процесора, а не в ОЗП, де зберігаються звичайні змінні. Це означало, що операції з регістровими змінними могли виконуватися набагато швидше, ніж операції з іншими (що зберігаються в пам'яті), оскільки для запиту або модифікації їх значень не потрібен був доступ до пам'яті.

Після стандартизації мови C було ухвалене рішення розширити визначення специфікатора класу пам'яті **register**. Згідно з ANSI-стандартом мови C, модифікатор **register** можна застосовувати до будь-якого типу даних. Його використання стало означати для компілятора вимогу зробити доступ до змінної типу **register** максимально швидким. Для ситуацій, що містять символи і цілочисельні значення, це, як і раніше, означає занесення їх у регістри ЦП, тому традиційне визначення все ще є в силі. Оскільки мова C++ побудована на ANSI-стандарті мови C, то він також підтримує розширене визначення специфікатора класу пам'яті **register**.

Як було зазначено вище, точна кількість **register**-змінних, які реально будуть оптимізовані в будь-якій одній функції, визначається як типом процесора, так і конкретною реалізацією мови програмування C++, яку Ви використовуєте. У загальному випадку можна розраховувати принаймні на дві. Проте не варто турбуватися про те, що Ви могли оголосити дуже багато **register**-змінних, оскільки мова C++ автоматично перетворить регістрові змінні на нерегістрові, коли їх ліміт буде вичерпаний. Це гарантує перенесність C++-коду програми у межах широкого діапазону процесорів.

Щоб показати вплив, що надається **register**-змінними на швидкодію програми, у наведеному нижче прикладі вимірюється час виконання двох циклів **for**, які відрізняються один від одного тільки типом змінних, що ним керують. У програмі використовується стандартна бібліотечна C++-функція **clock**(), яка повертає кількість імпульсів сигналу часу системного годинника, підрахованого з початку виконання цієї програми. Програма повинна містити заголовок <**ctime**>.

**Код програми 9.6. Демонстрація впливу використання register-змінної на швидкість виконання програми**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
#include <ctime>             // Використання системного часу і дати
using namespace std;         // Використання стандартного простору імен

unsigned int i;  // Не register-змінна
unsigned int delay;

int main()
{
```

```
        register unsigned int j;
        long start, end;
        start = clock();
        for(delay=0; delay<50; delay++)
                for(i=0; i < 64000000; i++);
        end = clock();
        cout << "Кількість тиків для не register-циклу: ";
        cout << end-start << endl;

        start = clock();
        for(delay=0; delay<50; delay++)
                for(j=0; j < 64000000; j++);
        end = clock();
        cout << "Кількість тиків для register-циклу: ";
        cout << end-start << endl;

        getch(); return 0;
}
```

У процесі виконання цієї програми Ви переконаєтеся у тому, що цикл з "регістровим" керуванням здійснюється приблизно в два рази швидше, ніж цикл з керуванням "нерегістровим". Якщо Ви не побачили сподіваної різниці, це може означати, що Ваш компілятор оптимізує всі змінні. Спробуйте виконати декілька розрахунків і переконайтеся в тому, що ця різниця на якомусь із них стає очевидною.

***Нео! Хідноапа, 'ятати!*** *Під час написання цього навчального посібника було використано середовище C++Builder 6, яке ігнорує ключове слово* ***register****. C++Builder 6 застосовує оптимізацію "як вважає за потрібне". Тому Ви можете не помітити впливу специфікатора класу пам'яті* ***register*** *на виконання попередньої програми. Проте ключове слово* ***register*** *все ще приймається компілятором без повідомлення про помилку. Воно просто не спонукає до ніякої дії.*

## 9.3. Поняття про порозрядні оператори

Оскільки мова C++ спрямована на те, щоб відкрити повний доступ до апаратних засобів комп'ютера, важливо, щоб оператор мав можливість безпосередньо впливати на окремі біти у межах байта або машинного слова. Саме тому мова програмування C++ і містить *порозрядні оператори*, призначені для тестування, встановлення або зсуву реальних бітів у байтах або словах, які відповідають символьним або цілочисельним C++-типам даних. Порозрядні оператори не використовуються для операндів типу **bool**, **float**, **double**, **long double**, **void** або інших ще складніших типів даних.

*Порозрядні оператори обробляють окремі біти.*

Порозрядні оператори (вони перераховано у табл. 9.1) дуже часто використовують для вирішення широкого кола завдань програмування на системному рівні, наприклад, під час пошуку інформації про стан пристрою або його спрацювання. Тепер розглянемо кожного оператора цієї групи окремо.





**Табл. 9.1. Порозрядні оператори оброблення окремих бітів**

| Оператор | Значення |
|---|---|
| & | Порозрядне І (AND) |
| \| | Порозрядне АБО (OR) |
| ^ | Порозрядне виключення АБО (XOR) |
| >> | Зсув вправо |
| << | Зсув вліво |
| ~ | Доповнення до І (унарний оператор НЕ) |

### 9.3.1. Порозрядні оператори І, АБО, що виключає АБО і НЕ

Порозрядні оператори І, АБО, що виключають АБО і НЕ (що позначаються символами &, |, ^ і ~ відповідно) виконують ті самі операції, що і їх логічні еквіваленти (тобто вони діють згідно з тією ж таблицею істинності). Відмінність полягає тільки у тому, що порозрядні операції працюють на побітовій основі. У наведеній нижче таблиці показано результат виконання кожної порозрядної операції для всіх можливих поєднань операндів (нулів і одиниць).

**Табл. 9.1. Результат виконання порозрядних операцій поєднань операндів**

| p | q | p & q | p \| a | p ^ a | ~p |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 1 | 1 |
| 1 | 1 | 1 | 1 | 0 | 0 |

Як видно з цієї таблиці, результат застосування оператора XOR (що виключає АБО) буде дорівнювати значенню ІСТИНА (1) тільки у тому випадку, якщо істинний (дорівнює значенню 1) тільки один з операндів; інакше результат приймає значення ФАЛЬШ (0).

Порозрядний оператор І можна представити як спосіб придушення бітової інформації. Це означає, що 0 в будь-якому операнді забезпечить установку в 0 відповідного біта результату. Ось приклад:

```
  1101 0011
& 1010 1010
  1000 0010
```

Наведений нижче код програми зчитує символи з клавіатури і перетворює будь-який рядковий символ в його прописний еквівалент шляхом встановлення шостого біта, що дорівнює значенню 0. Набір символів ASCII визначено так, що рядкові букви мають майже такий самий код, що і прописні, за винятком того, що код перших відрізняється від коду програми інших точно на 32. Отже, як це показано у наведеному нижче коді програми, щоб з рядкової букви зробити прописну, достатньо очистити її шостий біт.

**Код програми 9.7. Демонстрація механізму перетворення рядкових букв у прописні**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char ch;
    do {
        cin >> ch;
        // Ця настанова онулює 6-й біт.
        ch = ch & 223; // В змінної ch тепер прописна буква.

        cout << ch;
    } while(ch !='Q');

    getch(); return 0;
}
```

Значення 223, що використовується в настанові порозрядного I, є десятковим представленням двійкового числа 1101 1111. Отже, ця операція I залишає всі біти в змінній ch незайманими, за винятком шостого (він скидається в нуль).

Оператор I також корисно використовувати, якщо потрібно визначити, чи встановлено біт, який Вас зацікавив, тобто чи дорівнює він значенню 1 чи ні. Наприклад, у процесі виконання такої настанови можна дізнатися, чи встановлено 4-й біт у змінній status:

```
if(status & 8) cout << "Біт 4 встановлено";
```

Щоб зрозуміти, чому для тестування четвертого біта використовують число 8, пригадаймо, що в двійковій системі числення число 8 представляється як 0000 1000, тобто в числі 8 встановлено тільки четвертий розряд. Тому умовний вираз у настанові **if** дасть значення ІСТИНА тільки у тому випадку, якщо четвертий біт змінної status також встановлено (дорівнює 1). Цікаве використання цього методу показано на прикладі функції DispBit(). Вона відображає в двійковому форматі конфігурацію бітів свого аргументу. Будемо використовувати функцію DispBit() нижче у цьому підрозділі для дослідження можливостей інших порозрядних операцій:

```
// Відображення конфігурації бітів у байті
void DispBit(unsigned u)
{
    register int t;

    for(t=128; t>0; t = t/2)
        if(u & t) cout << "1 ";
        else cout << "0 ";
    cout << endl;
}
```





Функція DispBit(), використовуючи порозрядний оператор I, послідовно тестує кожен біт молодшого байта змінної u, щоб визначити, встановлений він чи скинутий. Якщо він встановлений, то відображається цифра 1, інакше цифра 0. Заради інтересу спробуйте розширити цю функцію так, щоб вона відображала всі біти змінної I, а не тільки її молодший байт.

Порозрядний оператор АБО, на противагу порозрядному оператору I, зручно використовувати для встановлення потрібних бітів у одиницю. У процесі виконання операції АБО наявність в будь-якому операнді біта, що дорівнює 1, означає, що внаслідок відповідний біт також буде дорівнювати одиниці. Ось приклад.

```
  1101 0011
| 1010 1010
  1111 1011
```

Можна використовувати оператор АБО для перетворення розглянутої вище програми (яка перетворить рядкові символи в їх прописні еквіваленти) в її "протилежність", тобто тепер, як це показано далі, вона перетворюватиме прописні букви у рядкові.

**Код програми 9.8. Демонстрація механізму перетворення прописних букв у рядкові**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    char ch;
    do {
        cout << "Введіть прописну букву: "; cin >> ch;

        // Ця настанова робить букву рядковою, встановлюючи її 6-й біт.
        ch = ch | 32;
        cout << endl << "Рядкова буква: " << ch << endl;
    } while(ch != 'q');

    getch(); return 0;
}
```

Встановлення шостого біта перетворює прописну букву на її рядковий еквівалент.

Порозрядне виключення АБО (XOR) встановлює в одиницю біт результату тільки у тому випадку, якщо відповідні біти операндів відрізняються один від одного, тобто вони є не однаковими. Ось приклад:

```
  0111 1111
^ 1011 1001
  1100 0110
```

Унарний оператор НЕ (або оператор доповнення до 1) інвертує стан усіх бітів свого операнда. Наприклад, якщо цілочисельне значення (що зберігається в змінній A), є двійковим кодом 1001 0110, то внаслідок виконання операції ~A отримаємо двійковий код 0110 1001.

У наведеному нижче коді програми продемонстровано механізм використання оператора НЕ шляхом відображення деякого числа і його доповнення до 1 у двійковому коді за допомогою наведеної вище функції DispBit().

**Код програми 9.9. Демонстрація механізму використання оператора НЕ**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void DispBit(unsigned u);

int main()
{
      unsigned u;
      cout << "Введіть число між 0 і 255: "; cin >> u;
      cout << "Початкове число в двійковому коді: ";
      DispBit(u);

      cout << "Його доповнення до одиниці: ";
      DispBit(~u);
      getch(); return 0;
}

// Відображення бітів, з яких складається байт.
void DispBit(unsigned u)
{
      register int t;

      for(t=128; t>0; t = t/2)
            if(u & t) cout << "1 ";
            else cout << "0 ";
      cout << endl;
}
```

Ось як виглядають результати виконання цієї програми.

```
Введіть число між 0 і 255: 99
Початкове число в двійковому коді: 01100011
Його доповнення до одиниці: 10011100
```

***Вартоа' нати!*** *Не плутайте логічні та порозрядні оператори. Вони виконують різні дії. Оператори &, | і ~ застосовуються безпосередньо до кожного біта значення окремо. Еквівалентні логічні оператори обробляють як операнди значення ІСТИНА/ФАЛЬШ (не нуль/нуль). Тому порозрядні оператори не можна використовувати замість їх логічних еквівалентів у умовних виразах. Наприклад, якщо значення x дорівнює 7, то вираз x && 8 має значення ІСТИНА, тоді як вираз x & 8 дає значення ФАЛЬШ.*





***Нео! хідноапа, 'ятати!*** *Оператор відношення або логічний оператор завжди генерує результат, який має значення ІСТИНА або ФАЛЬШ, тоді як аналогічний порозрядний оператор генерує значення, що отримується згідно з таблицею істинності конкретної операції.*

### 9.3.2. Оператори зсуву

Оператори зсуву, ">>" і "<<", зсувають усі біти в значенні вправо або вліво. Загальний формат використання оператора зсуву вправо виглядає так:

*значення >> кількість_бітів*,

а оператор зсуву вліво використовують так:

*значення << кількість_бітів*.

У цьому записі елемент *кількість_бітів* вказує, на скільки позицій потрібно зсунути *значення*. Під час кожного зсування вліво всі біти, що складають значення, зсуваються вліво на одну позицію, а у молодший розряд записують нуль. Під час кожного зсування вправо всі біти зсуваються, відповідно, вправо. Якщо зсуву вправо піддається значення без знаку, то в старший розряд записують нуль. Якщо ж зсуву вправо піддається значення зі знаком, то значення знакового розряду зберігається. Як Ви пам'ятаєте, негативні цілі числа представляються встановленням старшого розряду числа, що дорівнює одиниці. Таким чином, якщо зсовуване значення є негативним, то під час кожного зсування вправо в старший розряд записують одиницю, а якщо позитивне – нуль. Не забувайте, зсув, що здійснюється операторами зсуву, не є циклічним, тобто при зсуванні як вправо, так і вліво крайні біти втрачаються, і вміст втраченого біта дізнатися неможливо.

*Оператори зсуву призначені для зсуву бітів у рамках цілочисельного значення.*

Оператори зсуву працюють тільки зі значеннями цілочисельних типів, наприклад, символами, цілими числами і довгими цілими числами. Вони не застосовуються до значень з плинною крапкою.

Побітові операції зсуву можуть виявитися вельми корисними для декодування вхідної інформації, що отримується від зовнішніх пристроїв (наприклад, цифроаналогових перетворювачів), і оброблення інформації про стан пристроїв. Порозрядні оператори зсуву можна також використовувати для виконання прискорених операцій множення і ділення цілих чисел. За допомогою зсуву вліво можна ефективно множити на два, а зсув вправо дає змогу не менш ефективно ділити два.

Наведений нижче код програми наочно ілюструє результат використання операторів порозрядного зсуву.

**Код програми 9.10. Демонстрація механізму виконання операторів порозрядного зсуву**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
```

```
using namespace std;          // Використання стандартного простору імен

void DispBit(unsigned u);

int main()
{
      int c=1, t;
      for(t=0; t<8; t++) {
            DispBit(c);
            c = c << 1;
      }
      cout << endl;

      for(t=0; t<8; t++) {
            c = c >> 1;
            DispBit(c);
      }

      getch(); return 0;
}

// Відображення бітів, з яких складається байт.
void DispBit(unsigned u)
{
      register int t;

      for(t=128; t>0; t=t/2)
            if(u & t) cout << "1 ";
            else cout << "0 ";
      cout << endl;
}
```

Результати виконання цієї програми є такими:

```
0 0 0 0 0 0 0 1
0 0 0 0 0 0 1 0
0 0 0 0 0 1 0 0
0 0 0 0 1 0 0 0
0 0 0 1 0 0 0 0
0 0 1 0 0 0 0 0
0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0

1 0 0 0 0 0 0 0
0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0
0 0 0 1 0 0 0 0
0 0 0 0 1 0 0 0
0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 1
```





## 9.4. Поняття про спеціальні оператори розширення можливостей мови С++

Вище у цьому посібнику Ви вже познайомилися з більшістю операторів, які не є унікальними у мові С++. Але, на відміну від інших мов, у мові програмування С++ передбачено і інші спеціальні оператори, які значно розширюють можливості мови і підвищують її гнучкість. Цим операторам і присвячено частину підрозділу, що залишилася.

### 9.4.1. Перерахунки – списки іменованих цілочисельних констант

У мові програмування С++ можна визначити список іменованих цілочисельних констант. Такий список називається *перерахунком* (enumeration). Ці константи можна потім використовувати скрізь, де допускаються цілочисельні значення (наприклад, в цілочисельних виразах). Перерахунки визначаються за допомогою ключового слова **enum**, а формат їх визначення має такий вигляд:

**enum** *type_name* { *перелік_перерахунку* } *перелік_змінних*;

Під елементом *перелік_перерахунку* розуміють список розділених між собою комами імен, які представляють значення перерахунку. Елемент *перелік_змінних* є необов'язковим, оскільки змінні можна оголосити пізніше, використовуючи ім'я типу перерахунку. У наведеному нижче прикладі визначається перерахунок apple і дві змінні типу apple з іменами red і yellow:

```
enum apple {Jonathan Golden_Del, Red_Del, Winesap, Cortland, McIntosh} red, yellow;
```

Визначивши перерахунок, можна оголосити інші змінні цього типу, використовуючи ім'я перерахунку. Наприклад, за допомогою такої настанови оголошується одна змінна fruit перерахунку apple:

```
apple fruit;
```

Цю настанову можна записати і так:

```
enum apple fruit;
```

Проте використання ключового слова **enum** тут є зайвим. У мові програмування С (яка також підтримує перерахунки) обов'язковою була друга форма, тому в деяких програмах Ви можете натрапити на подібний запис.

---

*Ключове слово* ***enum*** *оголошує перерахунок.*

З урахуванням попередніх оголошень наведені нижче типи настанов абсолютно допускаються:

```
fruit = Winesap;
if(fruit==Red_Del) cout << "Red Delicious" << endl;
```

Важливо розуміти, що кожен символ перерахунку означає ціле число, причому кожне наступне число (представлене ідентифікатором) є на одиницю більшим від попереднього. За замовчуванням значення першого символу перерахунку дорівнює нулю, отже, значення другого – одиниці і т.д. Тому у процесі виконання цієї настанови

```
cout << Jonathan << " " << Cortland;
```

на екран буде виведено числа 0 4.

Хоча перерахункові константи автоматично перетворяться в цілочисельні, зворотне перетворення автоматично не здійснюється. Наприклад, така настанова є некоректною

```
fruit = 1; // помилка
```

Ця настанова спричинить під час компілювання помилку, оскільки автоматичного перетворення цілочисельних значень в значення типу apple не існує. Відкоректувати попередню настанову можна за допомогою операції приведення типів:

```
fruit = (apple) 1; // Тепер все гаразд, але стиль не досконалий.
```

Тепер змінна fruit міститиме значення Golden_Del, оскільки ця apple-константа зв'язується із значенням 1. Як зазначено в коментарі, хоча ця настанова стала коректною, її стиль написання надто недосконалий, тобто його можна вибачити тільки за виняткових обставин.

Використовуючи ініціалізацію, можна вказати значення однієї або декількох перерахункових констант. Це робиться так: після відповідного елемента перерахунку ставиться знак рівності і потрібне ціле число. Під час використання ініціалізації наступному (після того, що ініціалізувало) елементу перерахунку присвоюється значення, що на одиницю перевищує попереднє значення ініціалізації. Наприклад, у процесі виконання такої настанови константі Winesap присвоюється значення 10:

```
enum apple {Jonathan Golden_Del, Red_Del, Winesap=10, Cortland, McIntosh};
```

Тепер всі символи перерахунку apple мають такі значення:

| | |
|---|---|
| Jonathan | 0 |
| Golden_Del | 1 |
| Red_Del | 2 |
| Winesap | 10 |
| Cortland | 11 |
| McIntosh | 12 |

Часто відносно перерахунків помилково вважають, що символи перерахунку можна вводити і виводити як рядки. Наприклад, такий фрагмент коду програми виконаний не буде:

```
// Слово "McIntosh" на екран таким чином не потрапить.
fruit = McIntosh;
cout << fruit;
```

Не забувайте, що символ McIntosh – просто ім'я для певного цілочисельного значення, а не рядок. Отже, у процесі виконання попереднього коду програми на екрані відобразиться числове значення константи McIntosh, а не рядок "McIntosh". Звичайно, можна створити код введення та виведення символів перерахунку у вигляді рядків, але він виходить дещо громіздким. Ось, наприклад, як можна відобразити на екрані назви сортів яблук, пов'язаних із змінною fruit:





```
switch(fruit) {
    case Jonathan   : cout << "Jonathan";
        break;
    case Golden_Del: cout << "Golden Delicious";
        break;
    case Red_Del    : cout << "Red Delicious";
        break;
    case Winesap    : cout << "Winesap";
        break;
    case Cortland   : cout << "Cortland";
        break;
    case McIntosh: cout << "McIntosh";
        break;
}
```

Іноді для перекладу значення перерахунку у відповідний рядок можна оголосити масив рядків і використовувати значення перерахунку як індекс. Наведений нижче код програми виводить назви трьох сортів яблук:

**Код програми 9.11. Демонстрація механізму перекладання значення перерахунку у відповідний рядок**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

enum apple {Jonathan, Golden_Del, Red_Del, Winesap, Cortland, McIntosh};

// Масив рядків, пов'язаних з перерахунком apple.
char name[][20] = {
    "Jonathan",
    "Golden Delicious",
    "Red Delicious",
    "Winesap",
    "Cortland",
    "McIntosh"
    };

int main()
{
    apple fruit;
    fruit = Jonathan; cout << name[fruit] << endl;
    fruit = Winesap; cout << name[fruit] << endl;
    fruit = McIntosh; cout << name[fruit] << endl;
    getch(); return 0;
}
```

Результати виконання цієї програми є такими:

```
Jonathan
Winesap
McIntosh
```

Використаний у цьому коді програми метод перетворення значення перерахунку в рядок можна застосувати до перерахунку будь-якого типу, якщо він не містить ініціалізацій. Для належного індексування елементів масиву рядків перерахункові константи повинні починатися з нуля, бути строго впорядкованими за збільшенням, і кожна наступна константа повинна бути більшою за попередню точно на одиницю.

Через те, що значення перерахунку необхідно вручну перетворювати в зручні для сприйняття людиною рядки, то вони, в основному, використовуються там, де таке перетворення не потрібне. Для прикладу розгляньте перерахунок, що використовують для визначення таблиці символів компілятора.

### 9.4.2. Створення нових імен для наявних типів даних

У мові програмування C++ дозволено визначати нові імена типів даних за допомогою ключового слова **typedef**. Під час використання **typedef**-імені новий тип даних не створюється, а тільки визначається нове ім'я для вже наявного типу даних. Завдяки **typedef**-іменам можна зробити машинозалежні програми більш переносними: для цього іноді достатньо змінити **typedef**-настанови. Цей засіб також дає змогу поліпшити читабельність коду програми, оскільки для стандартних типів даних за допомогою нього можна використовувати описові імена. Загальний формат запису настанови **typedef** є таким:

**typedef** *тип нове_ім'я*;

У цьому записі елемент *тип* означає будь-який допустимий тип даних, а елемент *нове_ім'я* – нове ім'я для цього типу. При цьому зауважте: нове ім'я визначається Вами як доповнення до наявного імені типу, а не для його заміни.

Наприклад, за допомогою такої настанови можна створити нове ім'я для типу **float**:

**typedef float** balance;

Ця настанова є розпорядженням компіляторові розпізнавати ідентифікатор balance як ще одне ім'я для типу **float**. Після цієї настанови можна створювати **float**-змінні з використанням імені balance:

balance over_due;

У цьому записі оголошена змінна з плинною крапкою over_due типу balance, який є стандартним типом **float**, але таким, що має іншу назву.

### 9.4.3. Оператор "знак запитання"

Одним з найчудовіших операторів C++ є оператор "?", який можна використовувати як заміну **if**-**else**-настанов, що вживаються в такому загальному форматі:

```
if(умова)
    змінна = вираз1;
else
    змінна = вираз2;
```





У цьому записі значення, що присвоюється змінній, залежить від результату обчислення елемента *умова*, що керує настановою **if**.

Оператор "?" називається *тернарним*, оскільки він працює з трьома операндами. Ось його загальний формат запису:

*Вираз1 ? Вираз2 : Вираз3*;

Всі елементи тут є виразами. Зверніть увагу на використання і розташування двокрапки.

Значення ?-виразу визначається таким чином. Обчислюється *Вираз1*. Якщо він виявляється істинним, то обчислюється *Вираз2*, і результат його обчислення стає значенням всього ?-виразу. Якщо результат обчислення елемента *Вираз1* виявляється помилковим, то значення всього ?-виразу стає результатом обчислення елемента *Вираз3*. Розглянемо такий приклад:

```
while(something) {
    x = count > 0 ? 0 : 1;
    //...
}
```

У цьому записі змінній x присвоюватиметься значення 0 доти, доки значення змінної *count* не стане меншим або дорівнюватиме нулю. Аналогічний програмний код (але з використанням **if**-**else**-настанови) виглядав би так:

```
while(something) {
    if(count > 0) x = 0;
    else x = 1;
    //...
}
```

А ось ще один приклад практичного застосування оператора "?". Наведений нижче код програми ділить два числа, але не допускає ділення на нуль.

**Код програми 9.12. Демонстрація механізму використання оператора "?" для запобігання ділення на нуль**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int DivZero();

int main()
{
    int c, d, result;
    cout << "Введіть ділене і дільник: "; cin >> c >> d;

    // Ця настанова не допустить виникнення помилки ділення на нуль.
    result = d ? c/d : DivZero();

    cout << "Результат: " << result;
    getch(); return 0;
}
```

```
int DivZero() {
    cout << "Не можна ділити на нуль" << endl;

    getch(); return 0;
}
```

У цьому коді програми, якщо значення змінної d не дорівнює нулю, то здійснюється ділення значення змінної c на значення змінної d, а результат присвоюється змінній result. Інакше викликається обробник помилки ділення на нуль DivZero(), і змінній result присвоюється нульове значення.

### 9.4.4. Складені оператори присвоєння

У мові програмування C++ передбачено спеціальні складені оператори присвоєння, у яких об'єднано присвоєння з ще однією операцією. Почнемо з прикладу і розглянемо таку настанову:

```
x = x + 10;
```

Використовуючи складений оператор присвоєння, його можна записати у такому вигляді:

```
x += 10;
```

Пара операторів += слугує вказанням компіляторові присвоїти змінній x суму поточного значення змінної x і числа 10. Цей приклад є ілюстрацією того, що складені оператори присвоєння спрощують програмування певних настанов присвоєння. Окрім цього, вони дають змогу компіляторові згенерувати ефективніший код.

Складені версії операторів присвоєння існують для всіх бінарних операторів (тобто для всіх операторів, які працюють з двома операндами). Таким чином, при такому загальному форматі бінарних операторів присвоєння

*змінна = змінна op вираз*;

загальна форма запису їх складених версій має такий вигляд:

*змінна op = вираз*;

У цьому записі елемент *op* означає конкретний арифметичний або логічний оператор, що об'єднується з оператором присвоєння.

А ось ще один приклад. Настанова

```
x = x – 100;
```

аналогічна такій:

```
x –= 100;
```

Обидві ці настанови присвоюють змінній x її колишнє значення, зменшене на 100. Складені оператори присвоєння можна часто натрапити у професійно написаних C++-програмах, тому кожен C++-програміст повинен бути з ними ознайомлений.





### 9.4.5. Оператор "кома"

Не менш цікавим, ніж описані вище оператори, є такий оператор мови програмування C++, як "кома". Ви вже бачили декілька прикладів його використання в циклі **for**, де за допомогою "коми" було організовано ініціалізацію відразу декількох змінних. Але оператор "кома" також може складати частину виразу. Його призначення у цьому випадку – зв'язати певним чином декілька виразів. Значення перерахунку виразів, розділених між собою комами, визначається у цьому випадку значенням крайнього справа виразу. Значення інших виразів відкидаються. Отже, значення виразу справа стає значенням всього виразу-списку. Наприклад, у процесі виконання цієї настанови

```
var = (count=19, incr=10, count+1);
```

змінній count спочатку присвоюється число 19, змінною incr – число 10, а потім до значення змінної count додається одиниця, після чого змінній var присвоюється значення крайнього справа виразу, тобто count+1, яке дорівнює 20. Круглі дужки тут обов'язкові, оскільки оператор "кома" має нижчий пріоритет, ніж оператор присвоєння.

Щоб зрозуміти призначення оператора "кома", спробуємо виконати таку програму:

**Код програми 9.13. Демонстрація механізму використання оператора "кома"**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int c, d = 10;

    c = (d++, d+100, 999+d);
    cout << c;

    getch(); return 0;
}
```

Ця програма виводить на екран число 1010. І ось чому: спочатку змінній d присвоюється число 10, потім змінна d інкрементується до 11. Після цього обчислюється вираз d+100, який ніде не застосовується. Нарешті, здійснюється додавання значення змінної d (вона, як і раніше, дорівнює 11) з числом 999, що в результаті дає число 1010.

По суті, призначення оператора "кома" – забезпечити виконання заданої послідовності операцій. Якщо ця послідовність використовується в правій частині настанови присвоєння, то змінній, вказаній у її лівій частині, присвоюється значення останнього виразу із переліку виразів, розділених між собою комами. Оператор "кома" за його функціональним навантаженням можна порівняти із сполучником "і", що використовується у фразі: "зроби це, і те, і інше...".

### 9.4.6. Організація декількох присвоєнь "в одному"

Мова програмування C++ дає змогу застосувати дуже зручний метод одночасного присвоєння багатьом змінним одного і того ж значення. Йдеться про об'єднання відразу декількох присвоєнь в одній настанові. Наприклад, у процесі виконання цієї настанови змінним count, incr і index буде присвоєне число 10:

```
count = incr = index = 10;
```

Цей формат присвоєння декільком змінним загального значення можна часто натрапити в професійно написаних програмах.

### 9.4.7. Механізм використання ключового слова sizeof

Іноді корисно знати розмір (у байтах) одного з типів даних. Оскільки розміри вбудованих C++-типів даних в різних обчислювальних середовищах можуть бути різними, а знання розміру змінної в усіх ситуаціях має важливе значення, то для вирішення цього питання до мови програмування C++ залучено оператор (що діє під час компілювання програми), який використовується в двох таких форматах:

**sizeof**(*type*)
**sizeof** ***value***

---

*Оператор* ***sizeof*** *під час компілювання програми отримує розмір типу або значення.*

Перша версія повертає розмір заданого типу даних, а друга – розмір заданого значення. Якщо виникає потреба дізнатися розмір певного типу даних (наприклад, **int**), помістіть назву цього типу в круглі дужки. Якщо ж Вас цікавить розмір області пам'яті, яке займає конкретне значення, то можна обійтися без круглих дужок, хоча при бажанні їх можна використовувати.

Щоб зрозуміти, як працює оператор **sizeof**, випробуйте таку коротку програму. Для багатьох 32-розрядних середовищ вона повинна відобразити значення 1, 4, 4 і 8.

**Код програми 9.14. Демонстрація механізму використання оператора sizeof**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
      char ch;
      int c;
      cout << sizeof ch << endl;          // розмір типу char
      cout << sizeof c << endl;           // розмір типу int
      cout << sizeof(float) << endl;      // розмір типу float
      cout << sizeof(double) << endl;     // розмір типу double
      getch(); return 0;
}
```





Як було зазначено вище, оператор **sizeof** діє під час компілювання програми. Вся інформація, необхідна для обчислення розміру вказаної змінної або заданого типу даних, відома вже під час компілювання.

Оператора **sizeof** можна застосувати до будь-якого типу даних. Наприклад, у разі застосування його до масиву він повертає кількість байтів, які займає масив. Розглянемо такий фрагмент коду програми:

```
int Array[4];
cout << sizeof Array;          // Буде виведене число 16.
```

Для 4-байтних значень типу **int** у процесі виконання цього фрагмента коду програми на екрані відобразиться число 16 (яке виходить внаслідок множення 4 байтів на 4 елементи масиву).

Оператор **sizeof** в основному використовується під час написання коду програми, який залежить від розміру C++-типів даних. Пам'ятайте: оскільки розміри типів даних у мові програмування C++ визначаються конкретною реалізацією, не варто покладатися на розміри типів, визначених в реалізації, у якій Ви працюєте в цей момент.

## 9.5. C++-система динамічного розподілу пам'яті

Для C++-програми існує два основні способи зберігання інформації в основній пам'яті комп'ютера. *Перший* полягає у використанні змінних. Область пам'яті, що надається змінним, закріплюється за ними під час компілювання і не може бути змінена у процесі виконання програми. *Другий спосіб* полягає у використанні C++-системи динамічного розподілу пам'яті. У цьому випадку пам'ять для даних виділяється в міру потреби з розділу вільної пам'яті, який розташований між Вашою програмою (і її постійною областю зберігання) та стеком. Цей розділ називається *"купою"* (heap). Розташування програми в пам'яті схематично показано на рис. 9.1.

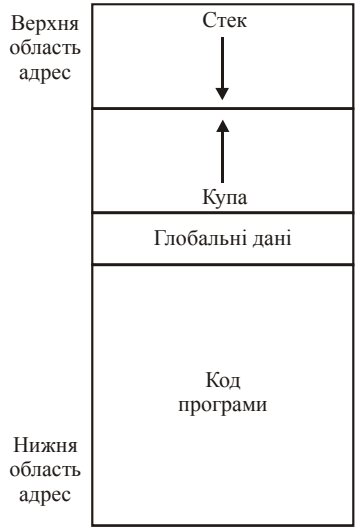



**Рис. 9.1. Механізм використання пам'яті у мові програмування C++**

---

*Система динамічного розподілу пам'яті – засіб отримання програмою деякої області пам'яті під час її виконання.*

Динамічне виділення області пам'яті – отримання програмою пам'яті під час її виконання. Іншими словами, завдяки цій системі програма може створювати змінні у процесі виконання, причому в потрібній (залежно від ситуації) кількості. Ця система динамічного розподілу пам'яті особливо цінна для таких структур даних, як зв'язні

списки і двійкові дерева, які змінюють свій розмір у міру їх використання. Динамічне виділення області пам'яті для тих або інших потреб – важлива складова майже всіх реальних програм.

Щоб задовольнити запит на динамічне виділення області пам'яті, використовують так звану "сукупність". Неважко припустити, що в деяких надзвичайних ситуаціях вільна пам'ять "сукупності" може вичерпатися. Отже, незважаючи на те, що динамічний розподіл пам'яті (порівняно з фіксованим) забезпечує велику гнучкість, але і у цьому випадку він має свої межі використання.

### 9.5.1. Оператори динамічного розподілу пам'яті

Мова C++ містить два оператори – **new** і **delete**, які виконують функції з виділення та звільнення пам'яті. Їх загальний формат має такий вигляд:

*змінна-покажчик* = **new** *тип_змінної;*
**delete** змінна-покажчик;

У цьому записі елемент *змінна-покажчик* є покажчиком на значення, тип якого задано елементом *тип_змінної*. Оператор **new** виділяє область пам'яті, достатню для зберігання значення заданого типу, і повертає покажчик на цю область пам'яті. За допомогою оператора **new** можна виділити пам'ять для значень будь-якого допустимого типу.

---

*Оператор* ***new*** *дає змогу динамічно виділити область пам'яті.*

Оператор **delete** звільняє область пам'яті, яка адресується заданим покажчиком. Після звільнення ця пам'ять може бути знову виділена для інших потреб під час подальшого **new**-запиту на виділення області пам'яті.

---

*Оператор* ***delete*** *звільняє раніше виділену динамічну пам'ять.*

Оскільки об'єм "сукупності" є скінченим, то вона може коли-небудь вичерпатися. Якщо для задоволення чергового запиту на виділення області пам'яті не існує достатньо вільної пам'яті, то оператор **new** зазнає фіаско, і згенерує виняток. *Виняток* – помилка спеціального типу, яка виникає у процесі виконання програми (у мові програмування C++ передбачена цілу підсистему, призначену для оброблення таких помилок (див. розд. 18)). У загальному випадку Ваша програма повинна обробити подібний виняток і за змогою виконати дію, відповідну конкретній ситуації. Якщо цей виняток не буде оброблено Вашою програмою, то її виконання буде припинено.

Така поведінка оператора **new** у разі неможливості задовольнити запит на виділення області пам'яті визначається стандартом мови C++. На таку реалізацію налаштовані також всі сучасні компілятори, в т.ч. останні версії Visual C++ і C++ Builder. Однак йдеться про те, що деякі старіші компілятори обробляють **new**-настанови дещо по-іншому. Відразу після винайдення мови C++ оператор **new** під час невдалого виконання повертав нульовий покажчик. Пізніше його реалізація була змінена так, щоб у разі невдачі генерувався виняток, як це було описано вище. Оскільки у цьому посібнику ми дотримуємося стандарту мови C++, то в усіх представлених тут прикладах передбачається саме генерування винятку. Якщо ж Ви використаєте старіший компілятор, то зверніться до документації, що додається до нього, і уточніть, як реалізований оператор **new** (спробуйте внести в приклади відповідні зміни).





Оскільки винятки розглядатимемо нижче після теми класів і об'єктів, то поки що не варто відволікатися на оброблення винятків, що генеруються у разі невдалого виконання оператора **new**. Окрім цього, жоден з прикладів у цьому і подальших розділах не повинен викликати невдалого виконання оператора **new**, оскільки в цих програмах запитується тільки декілька байтів. Але, якщо така ситуація все ж виникне, то у гіршому випадку це призведе до завершення роботи програми. Пізніше, при розгляду теми щодо оброблення винятків, можна дізнатися, як обробити виняток, який було згенеровано оператором **new**.

Розглянемо приклад програми, яка ілюструє використання операторів **new** і **delete**.

**Код програми 9.15. Демонстрація механізму динамічного розподілу пам'яті**

```
#include <iostream>          // Потокове введення-виведення
using namespace std;         // Використання стандартного простору імен

int main()
{
    int *p;
    p = new int;             // Виділяємо пам'ять для int-значення.
    *p = 20;                 // Поміщаємо в цю область пам'яті значення 20.
    cout << *p;              // Переконуємося (шляхом виведення на екран) в
                             // роботоздатності цього коду програми.
    delete p;                // Звільняємо пам'ять.

    getch(); return 0;
}
```

Ця програма присвоює покажчику p адресу (узятої з "сукупності") області пам'яті, яка матиме розмір, достатній для зберігання цілочисельного значення. Потім у цю область пам'яті поміщається число 20, після чого на екрані монітора з'являється її вміст. Нарешті, динамічно виділена пам'ять звільняється.

Завдяки такому способу організації динамічного виділення області пам'яті оператор **delete** необхідно використовувати тільки з тим покажчиком на пам'ять, який було повернуто внаслідок **new**-запиту на виділення області пам'яті. Використання оператора **delete** з іншим типом адреси може викликати серйозні проблеми.

### 9.5.2. Ініціалізація динамічно виділеної пам'яті

Використовуючи оператор **new**, динамічно виділену пам'ять можна ініціалізувати. Для цього після імені типу необхідно задати початкове значення,

уклавши його в круглі дужки. Наприклад, у наведеному нижче коді програми область пам'яті, яка адресується покажчиком p, ініціалізується значенням 99.

**Код програми 9.16. Демонстрація механізму ініціалізації динамічно виділеної пам'яті**

```
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Для консольного режиму роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      int *p;

      p = new int (99);      // Ініціалізується пам'ять числом 99.
      cout << *p;            // На екран виводиться число 99.
      delete p;

      getch(); return 0;
}
```

### 9.5.3. Динамічне виділення області пам'яті для масивів

За допомогою оператора **new** можна динамічно виділяти пам'ять і для масивів. Ось як виглядає загальний формат операції динамічного виділення області пам'яті для одновимірного масиву:

*змінна-покажчик* = **new** *тип* [*розмір*];

У цьому записі елемент *розмір* задає кількість елементів у масиві.

Щоб звільнити пам'ять, виділену для динамічно створеного масиву, використовують такий формат оператора **delete**:

**delete** [] *змінна-покажчик*;

У цьому записі елемент *змінна-покажчик* є адресою, що отримується під час виділення області пам'яті для масиву (за допомогою оператора **new**). Квадратні дужки означають для C++-компілятора, що динамічно створений масив видаляється, а вся область пам'яті, виділена для нього, автоматично звільняється.

***Нео! хідноапа, 'ятати!а****Старіші версії C++-компілятора можуть вимагати задавання розміру масиву, що видаляється, оскільки в ранніх версіях мови C++ для звільнення пам'яті, займаної масивом, що видаляється, необхідно було застосовувати такий формат оператора* ***delete****:*

**delete** [*розмір*] *змінна-покажчик*;

*У цьому записі елемент* розмір *задає кількість елементів у масиві. Стандарт мови програмування C++ більше не вимагає вказувати розмір при його видаленні.*

У процесі виконання наведеної нижче програми виділяється пам'ять для 10-елементного масиву типу **double**, який потім заповнюється значеннями від 100 до 109, після чого вміст цього масиву відображається на екрані.





**Код програми 9.17. Демонстрація механізму виділення області пам'яті для масивів**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
      double *p;
      p = new double [10]; // Виділення області пам'яті для 10-ти елементів масиву.

      // Заповнюємо масив значеннями від 100 до 109.
      for(int i=0; i<10; i++) p[i] = 100.00 + i;

      // Відображаємо вміст масиву.
      cout << "Ініціалізований масив <p>" << endl;
      for(int i=0; i<10; i++) cout << p[i] << " ";
      cout << endl;
      delete [] p; // Видалення всього масиву.

      cout << "Видалений масив <p>" << endl;
      for(int i=0; i<10; i++) cout << p[i] << " ";
      cout << endl;
      getch(); return 0;
}
```

***Нео! хідноапа, 'ятати!а****При динамічному виділенні пам'яті для масиву його не можна одночасно ініціалізувати.*

### 9.5.4. Функції виділення та звільнення пам'яті у мові програмування C

Мова програмування C не містить операторів **new** або **delete**. Замість них у мові C використовують бібліотечні функції, призначені для виділення і звільнення пам'яті. З метою сумісності, мова C++, як і раніше, підтримує C-систему динамічного розподілу пам'яті і не дарма: у C++-програмах все ще використовують C-орієнтовані засоби динамічного розподілу пам'яті. Тому їм варто приділити певну увагу.

Ядро C-системи динамічного розподілу пам'яті становлять функції **malloc**() і **free**(). Функція **malloc**() призначена для виділення області пам'яті, а функція **free**() – для її звільнення. Іншими словами, кожного разу, коли за допомогою функції **malloc**() робиться запит, частина вільної пам'яті виділяється відповідно до цього запиту. Під час кожного виклику функції **free**() відповідна область пам'яті повертається системі. Будь-яка програма, яка використовує ці функції, повинна містити заголовок <**cstdlib**>.

Функція **malloc**() має такий прототип:

**void** ***malloc**(size_t *num_bytes*);

У цьому записі *num_bytes* означає кількість байтів запрошуваної пам'яті. Тип size_t є різновид цілого типу без знаку. Функція **malloc**() повертає покажчик типу **void**, який відіграє роль узагальненого покажчика. Щоб з цього узагальненого покажчика отримати покажчик на потрібний Вам тип, необхідно використовувати операцію приведення типів. Внаслідок успішного виклику функція **malloc**() поверне покажчик на перший байт області пам'яті, виділеної з "сукупності". Якщо для задоволення запиту вільної пам'яті в системі недостатньо, функція **malloc**() повертає нульовий покажчик.

Функція **free**() здійснює дію, зворотну дії функції **malloc**() тому, що вона повертає системі раніше виділену нею пам'ять. Після звільнення пам'ять можна знову використовувати подальшим зверненням до функції **malloc**(). Функція **free**() має такий прототип:

**void free**(**void** **ptr*);

У цьому записі параметр *ptr* є покажчиком на пам'ять, раніше виділену за допомогою функції **malloc**(). Ніколи не варто викликати функцію **free**() з недійсним аргументом; це може призвести до руйнування переліку областей пам'яті, що підлягають звільненню.

Використання функцій **malloc**() і **free**() продемонстровано у наведеному нижче коді програми.

**Код програми 9.18. Демонстрація механізму використання функцій malloc() і free()**

```
#include <iostream>          // Потокове введення-виведення
#include <cstdlib>           // Використання бібліотечних функцій
using namespace std;         // Використання стандартного простору імен

int main()
{
    int *c; double *d;

    c = (int *) malloc(sizeof(int));
    if(!c) {
        cout << "Виділити пам'ять не вдалося" << endl;
        return 1;
    }

    d = (double *) malloc(sizeof(double));
    if(!d) {
        cout << "Виділити пам'ять не вдалося" << endl;
        return 1;
    }
    *c = 10; *d = 100.123;
    cout << *c << " " << *d;

    // Звільнення пам'яті.
    free(c); free(d);
    getch(); return 0;
}
```





Хоча функції **malloc**() і **free**() повністю придатні для динамічного розподілу пам'яті, є ряд причин, згідно з якими у мові програмування С++ визначено власні засоби динамічного розподілу пам'яті:

- *по-перше*, оператор **new** автоматично обчислює розмір області пам'яті, що виділяється, для заданого типу, тобто програмісту не потрібно використовувати оператор **sizeof**, а значить, наявна економія в коді програми і трудовитратах програміста. Але важливіше те, що автоматичне обчислення не допускає виділення неправильного об'єму пам'яті;
- *по-друге*, С++-оператор **new** автоматично повертає коректний тип покажчика, що звільняє програміста від потреби використовувати операцію приведення типів даних;
- *по-третє*, використовуючи оператора **new**, можна ініціалізувати об'єкт, для якого виділяється пам'ять.

Нарешті, як буде показано далі у цьому посібнику, програміст може створити власні версії операторів **new** і **delete**.

І останнє. Через можливість несумісності не варто змішувати функції **malloc**() і **free**() з операторами **new** і **delete** в одній програмі.

## 9.6. Зведена таблиця пріоритетів виконання С++-операторів

У табл. 9.3 показано пріоритет виконання всіх С++-операторів (від вищого до найнижчого). Більшість операторів асоційована зліва направо. Але унарні оператори, оператори присвоєння і оператор "?" асоційовані справа наліво. Зверніть увагу на те, що ця таблиця містить декілька операторів, які ми поки що не використовували в наших прикладах, оскільки вони належать до об'єктно-орієнтованого програмування (і описані нижче).

**Табл. 9.3. Пріоритет С++--операторів**

| | |
|---|---|
| Найвищий | () [] -> :: . |
| | ! ~ ++ -- - * & **sizeof new delete typeid** оператори приведення типу |
| | .* ->* |
| | * / % |
| | + |
| | << >> |
| | < <= > >= |
| | == != |
| | & |
| | ^ |
| | \| |
| | && |
| | \|\| |
| | ?: |
| | = += -= *= /= %= >>= <<= &= ^= \|= |
| Нижчий | , |

# Розділ 10. ПОНЯТТЯ ПРО СТРУКТУРИ І ОБ'ЄДНАННЯ ДАНИХ

У мові програмування С++ визначено декілька складених типів даних, тобто типів, які складаються з двох або більше елементів. Такі складені типи даних як масив і перерахунок розглядалися раніше відповідно у розд. 6.4 і 9.4. З двома іншими – структурами і об'єднаннями – ми познайомимося у цьому розділі, а знайомство з ще одним складеним типом – класом – ми відкладемо до розд. 12. І, хоча структуровані та об'єднані типи даних призначено для задоволення різних програмних потреб, обидва вони є зручними засобами керування групами взаємопов'язаних між собою змінних. При цьому важливо розуміти, що створення структури (або об'єднання) означає створення нового *визначеного програмістом* типу даних. Сама можливість створення власних типів даних є ознакою потужності мови С++.

У мові програмування С++ структури та об'єднання мають як об'єктно-орієнтовані, так і не об'єктно-орієнтовані атрибути. У цьому розділі детально проаналізуємо тільки останні. Об'єктно-орієнтовані їх властивості будемо розглядати в наступному розділі після введення таких понять як класи і об'єкти.

## 10.1. Організація роботи зі структурами даних

У програмуванні ***структури даних*** – способи організації даних у комп'ютерах. Часто разом зі структурою даних пов'язується і специфічний перелік операцій, що можуть бути виконаними над даними, організованими в таку структуру. Правильний підбір структур даних є надзвичайно важливим для ефективного функціонування відповідних алгоритмів їх оброблення. Добре побудовані структури даних дають змогу оптимізувати використання машинного часу та пам'яті комп'ютера для виконання найбільш критичних операцій.

Відома формула "Програма = Алгоритми + Структури даних" дуже точно виражає потребу відповідального ставлення до такого підбору. Тому іноді навіть не обраний алгоритм для оброблення масиву даних визначає вибір тої чи іншої структури даних для їх збереження, а навпаки. Підтримка базових структури даних, які використовуються в програмуванні, включена в комплекти стандартних бібліотек мови програмування С++.

### 10.1.1. Основні положення

У мові програмування С++ *структура* представляє колекцію змінних, об'єднаних загальним іменем, яка забезпечує зручний засіб зберігання споріднених даних в одному місці. *Структура* – сукупність різних типів даних, оскільки вони складаються з декількох різних, але логічно взаємопов'язаних





між собою змінних. З цих самих причин структури іноді називають *складеними* або *конгломератними типами даних.*

***Структура*** *– група взаємопов'язаних між собою змінних.*

Перед визначенням структурних змінних, необхідно визначити формат структури. Це робиться за допомогою оголошення структури. Оголошення структури дає змогу компілятору зрозуміти, змінні якого типу вона містить. Змінні, які належать до структури, називаються її *членами.* Члени структури також називають *полями.*

***Член структури*** *– змінна, яка є частиною структури.*

У загальному випадку всі члени структури мають бути логічно пов'язані одна з одною! Наприклад, структури зазвичай використовують для зберігання такої інформації, як поштові адреси, банківські реквізити, елементи книжкової бібліографії і т.ін. Безумовно, відносини між членами структури абсолютно суб'єктивні і визначаються програмістом. Компілятор "нічого про них не знає" (або "не хоче знати").

*Ключове слово* ***struct*** *означає початок оголошення структури.*

Загальний формат оголошення структури має такий вигляд:

```
struct ім'я_типу_структури {
    тип ім'я_члена_1;
    тип ім'я_члена_2;
    тип ім'я_члена_3;
    . . . . . .
    тип ім'я_члена_n;
} структурна_змінна_1, структурна_змінна_2, ..., структурна_змінна_m;
```

Розглянемо деякі приклади оголошення структур. Визначимо структуру, яка може містити інформацію про продукцію, що зберігаються на складі приватної фірми. Один запис інвентарної відомості зазвичай складається з декількох даних, наприклад: назву продукції, вартості та наявної кількості. Тому для керування такою інформацією зручно використовувати саме таку структуру. У наведеному нижче коді програми оголошується структура, яка визначає такі елементи: назву продукції, її вартість, роздрібну ціну, наявну кількість, кількість днів до поновлення запасів. Цих даних часто цілком достатньо для керування складським господарством. Про початок оголошення структури компіляторові повідомляє ключове слово **struct**:

```
struct invStruct {              // Попереднє оголошення типу структури
    char nameProd[40];          // Назва продукції
    double vartProd;            // Вартість продукції
    double rozdrCina;           // Роздрібна ціна
    int nayavKilk;              // Наявна кількість
    int kilkDniv;               // Кількість днів до поновлення запасів
};
```

***Ім'я типу структури*** *– її специфікатор типу даних.*

Зверніть увагу на те, що оголошення структури завершується крапкою з комою, тобто вона може бути настановою. Ім'ям типу структури тут є invStruct. Іншими словами, ім'я invStruct ідентифікує конкретну структуру даних і є її специфікатором типу.

У нашому оголошенні структури насправді не було створено жодної структурної змінної, а визначено тільки формат типу даних. Щоб за допомогою цієї структури визначити реальну структурну змінну (тобто фізичний об'єкт), потрібно записати таку настанову:

```
invStruct InvVidom;
```

Ось тепер визначається структурна змінна типу invStruct з іменем InvVidom.

***Нео! хідната, 'ятати!*** *Визначаючи структуру, визначаємо новий тип даних, але він не буде реалізований доти, доки не буде оголошено структурну змінну того типу, який вже реально існує.*

Під час визначення структурної змінної компілятор мови програмування C++ автоматично виділить об'єм пам'яті, достатній для зберігання всіх членів структури. На рис. 10.1 показано, як змінну InvVidom буде розміщено в пам'яті комп'ютера (у припущенні, що **double**-значення займає 8 байтів, а **int**-значення – 4 байти).

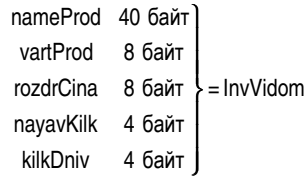



**Рис. 10.1. Розміщення структурної змінної InvVidom у пам'яті комп'ютера**

Одночасно з оголошенням імені типу структури можна визначити одну або декілька структурних змінних:

```
struct invStruct {              // Попереднє оголошення типу структури
    char nameProd[40];          // Назва продукції
    double vartProd;            // Вартість продукції
    double rozdrCina;           // Роздрібна ціна
    int nayavKilk;              // Наявна кількість
    int kilkDniv;               // Кількість днів до поновлення запасів
} InvVidomA, InvVidomB, InvVidomC;  // Визначення структурної змінної
```

Цей код програми оголошує структурний тип invStruct і визначає структурні змінні InvVidomA, InvVidomB і InvVidomC цього типу. Важливо розуміти, що кожна структурна змінна містить власні копії членів структури. Наприклад, поле vartProd структури InvVidomA ізольовано від поля vartProd структури InvVidomB. Отже, зміни, що вносяться в певне поле однієї структурної змінної, ніяк не впливають на вміст такого самого поля іншої структурної змінної.





Якщо для коду програми достатньо тільки однієї структурної змінної, то в оголошенні структури необов'язково міститиме ім'я структурного типу. Для розуміння сказаного розглянемо такий приклад:

```
struct {                        // Попереднє оголошення типу структури
    char nameProd[40];          // Назва продукції
    double vartProd;            // Вартість продукції
    double rozdrCina;           // Роздрібна ціна
    int nayavKilk;              // Наявна кількість
    int kilkDniv;               // Кількість днів до поновлення запасів
} vidom;                        // Визначення структурної змінної
```

Цей код програми визначає одну структурну змінну vidom відповідно до оголошення структури, яка їй передує.

### 10.1.2. Доступ до членів структури

До окремих членів структури доступ здійснюється за допомогою оператора "крапка". Наприклад, у процесі виконання такого коду програми значення 10.39 буде присвоєно полю vartProd структурної змінної InvVidom, яка була оголошена вище:

```
InvVidom.vartProd = 10.39;
```

Щоб звернутися до члена структури, потрібно перед його іменем поставити ім'я структурної змінної та оператор "крапка". Так здійснюється доступ до всіх членів структури. Загальний формат доступу до члена структури записується так:

*ім'я_структурної_змінної.ім'я_члена_структури*

*Оператор "крапка" (.) дає змогу отримати доступ до будь-якого члена відповідної структури.*

Щоб вивести значення поля vartProd структурної змінної InvVidom на екран, необхідно записати таку настанову:

```
cout << InvVidom.vartProd;
```

Аналогічним способом можна використовувати символьний масив InvVidom.nameProd у виклику функції **gets**():

```
gets(InvVidom.nameProd);
```

У цьому записі функції **gets**() буде передано символьний покажчик на початок області пам'яті, відведеної члену nameProd структури InvVidom.

Якщо виникає потреба отримати доступ до окремих елементів масиву структур під назвою, наприклад, InvVidom.nameProd, необхідно використати індексацію масиву. Наприклад, за допомогою цього коду програми можна по-символьно вивести на екран монітора вміст поля масиву структур InvVidom.nameProd:

```
for(int t=0; InvVidom.nameProd[t]; t++) cout << InvVidom.nameProd[t];
cout << endl;
```

### 10.1.3. Поняття про масиви структур

Структури можуть бути елементами масивів, тобто, масиви структур використовуються достатньо часто. Щоб визначити масив структур, необхідно спочатку оголосити структуру, а потім визначити масив елементів цього структурного типу. Наприклад, щоб визначити 100-елементний масив структур типу invStruct (який було оголошено вище), достатньо записати таку настанову:

```
invStruct prodArray[100];
```

Щоб отримати доступ до конкретної структури в масиві структур, необхідно індексувати ім'я структури. Щоб відобразити на екрані вміст члена nayavKilk третьої структури, достатньо використати таку настанову:

```
cout << prodArray[2].nayavKilk;
```

***Нео! хідноапа, 'ятати!*** *Подібно до всіх елементів масиву, у масиві елементів структур їх індексування починається з нуля.*

### 10.1.4. Приклад застосування структури

Щоб продемонструвати застосування структур, розробимо просту програму керування складом, у якій для зберігання інформації про продукцію (інвентарну відомість) використовується масив структур типу invStruct. Різні функції, які визначені у цьому коді програми, взаємодіють із структурою та її членами по-різному.

Запис інвентарної відомості можна зберігати в структурі типу invStruct, організовані у масиві під назвою prodArray:

```
const int rozmir = 100;
struct invStruct {                  // Попереднє оголошення типу структури
        char nameProd[40];          // Назва продукції
        double vartProd;            // Вартість продукції
        double rozdrCina;           // Роздрібна ціна
        int nayavKilk;              // Наявна кількість
        int kilkDniv;               // Кількість днів до поновлення запасів
} prodArray[rozmir];                // Визначення масиву структур
```

Розмір масиву вибрано довільно. При бажанні його можна легко змінити. Зверніть увагу на те, що розмірність масиву задано з використанням **const**-змінної. А оскільки розмір масиву в усій програмі використовується декілька разів, то застосування **const**-змінної для цього є виправданим. Щоб змінити розмір масиву, достатньо змінити значення константної змінної rozmir, а потім перекомпілювати програму. Використання **const**-змінної для визначення "магічного числа", яке часто уживається у програмі, – звичайна практика в професійному C++-коді програми.

Розроблена програма повинна забезпечити виконання таких дій:

- введення інформації про продукцію, що зберігатимуться на складі;
- відображення інвентарної відомості;
- модифікування полів заданого запису.





Передусім напишемо основну функцію **main**(), яка має мати приблизно такий вигляд:

```
int main()
{
        char vybir;
        InitList();   // Ініціалізація елементів масиву структур.
        for( ; ; ) {
                // Отримання команди меню, вибраної користувачем.
                vybir = MenuSelect();
                switch(vybir) {       // Введення запису в інвентарну відомість.
                        case 'e': Enter();
                                break;
                        // Відображення на екрані записів інвентарної відомості.
                        case 'd': Display();
                                break;
                        // Модифікування полів наявного запису відомості.
                        case 'u': UpDate();
                                break;
                        case 'q': return 0;
                }
        }
}
```

Функція **main**() починається з виклику функції InitList(), яка ініціалізує масив структур. Потім організовується цикл, який відображає меню і обробляє команду, вибрану користувачем. Наведемо код функції InitList():

```
void InitList()     // Ініціалізація елементів масиву структур.
{
        // Ім'я нульової довжини означає порожній запис.
        for(int t=0; t<rozmir; t++) *prodArray[t].nameProd = '\0';
}
```

Функція InitList() готує масив структур для подальшого використання, поміщаючи в перший байт поля nameProd нульовий символ. Передбачається, якщо поле nameProd порожнє, то на даний момент структура, у якій воно міститься, просто не використовується.

Функція MenuSelect() відображає команди меню і приймає варіант, вибраний користувачем:

```
int MenuSelect()          // Отримання команди меню, вибраної користувачем.
{
        char ch; cout << endl;
        do {
                cout << "(E)nter" << endl;          // Ввести новий запис.
                cout << "(D)isplay" << endl;        // Відобразити всю відомість.
                cout << "(U)pdate" << endl;         // Змінити поля запису.
                cout << "(Q)uit\n" << endl;         // Вийти з програми.
                cout << "Виберіть команду: "; cin >> ch;
        } while(!strchr("eduq", tolower(ch)));
        return tolower(ch);
}
```

Користувач вибирає із запропонованого меню команду, вводячи потрібну букву. Наприклад, щоб відобразити всю інвентарну відомість, потрібно натиснути букву "D".

Функція MenuSelect() викликає бібліотечну функцію мови C++ **strchr**(), яка має такий прототип:

**char** ***strchr(const char** **str*, **int** *ch*);

Ця функція проглядає рядок, який адресується покажчиком *str*, на предмет входження в неї символу, який зберігається в молодшому байті змінної *ch*. Якщо такий символ виявиться, то функція поверне покажчик на нього. І у цьому випадку значення, що повертається функцією, за визначенням буде ІСТИННИМ. Однак, якщо такого збігу символів не відбудеться, то функція поверне нульовий покажчик, який за визначенням є значення ФАЛЬШ. Так перевірка тут організована для того, щоби виявити, чи є значення, що вводяться користувачем, допустимими командами меню.

Функції Enter() передує виклик функції GetPut(), яка "підказує" користувачу порядок введення даних і приймає їх. Розглянемо програмні коди обох функцій:

```
void Enter()                    // Введення запису в інвентарну відомість.
{
    int i;
    // Знаходимо першу вільну структуру.
    for(i=0; i<rozmir; i++)
        if(!*prodArray[i].nameProd) break;

    // Якщо масив повний, то значення "i" буде дорівнювати rozmir.
    if(i==rozmir) {
        cout << "Перелік повний" << endl;
        return;
    }
    GetPut(i);          // Введення інформації.
}

void GetPut(int i)      // Введення інформації.
{
    cout << "Назва продукції        : ";    cin >> prodArray[i].nameProd;
    cout << "Вартість продукції     : ";    cin >> prodArray[i].vartProd;
    cout << "Роздрібна ціна  : ";    cin >> prodArray[i].rozdrCina;
    cout << "Наявна кількість: ";    cin >> prodArray[i].nayavKilk;
    cout << "Кількість днів до поновлення запасів: ";
    cin >> prodArray[i].kilkDniv;
}
```

Функція Enter() спочатку знаходить порожню структуру. Для цього перевіряється поле nameProd кожного (по черзі) елемента масиву prodArray, починаючи з першого. Якщо поле nameProd виявляється порожнім, то передбачається, що структура, до якої воно належить, ще нічим не зайнята. Якщо ж не знайдено жодної вільної структури при перевірці всього масиву структур, то





значення параметра циклу і, що керує ним, дорівнюватиме його розміру. Це свідчить про те, що масив повний, і в нього вже не можна нічого додати. Якщо ж у масиві знайдеться вільний елемент, то буде викликано функцію GetPut() для отримання інформації про продукцію, що вводиться користувачем. Якщо Вас цікавить, чому код введення даних про нову продукцію не є частиною функції Enter(), то відповідь є такою: функція GetPut() використовується також і функцією UpDate(), про яку мова ще попереду.

Оскільки інформація про продукцію на складі може змінюватися, програма ведення інвентарної відомості повинна давати змогу вносити зміни в її окремі записи. Це реалізується шляхом виклику функції UpDate():

```
void UpDate()   // Модифікування полів наявного запису відомості.
{
    int i; char name[80];

    cout << "Введіть назву продукції: "; cin >> name;

    for(i=0; i<rozmir; i++)
        if(!strcmp(name, prodArray[i].nameProd)) break;

    if(i==rozmir) {
        cout << "Продукцію не знайдено" << endl;
        return 0;
    }

    cout << "Введіть нову назву продукції" << endl;
    GetPut(i);          // Введення інформації.
}
```

Ця функція пропонує користувачу ввести назву продукції, інформацію про який йому потрібно змінити. Потім вона проглядає весь перелік наявних записів і, якщо вказана користувачем продукція у ньому є, то викликається функція GetPut(), яка забезпечує прийняття від користувача нової інформації.

Нам залишилося розглянути функцію Display(). Вона виводить на екран записи інвентарної відомості в повному обсязі. Код функції Display() має такий вигляд:

```
void Display() // Відображення на екрані записів інвентарної відомості.
{
    for(int t=0; t<rozmir; t++)
        if(*prodArray[t].nameProd) {
            cout << "Назва продукції:" << prodArray[t].nameProd << endl;
            cout << "Вартість продукції:" << prodArray[t].vartProd << " грн" << endl;
            cout << "У роздріб :" << prodArray[t].rozdrCina << " грн" << endl;
            cout << "У наявності:" << prodArray[t].nayavKilk << " шт" << endl;
            cout << "Залишилося :" << prodArray[t].kilkDniv << " днів" << endl;
        }
}
```

Нижче наведено завершений код програми ведення інвентарної відомості. Вам необхідно ввести цю програму в свій комп'ютер і досліджувати її ро-

боту. Внесіть деякі зміни і постежте, як вони відобразяться на її виконанні. Спробуйте також розширити програму, додавши функції пошуку в списку заданої продукції, видалення вже непотрібного запису або повного очищення інвентарної відомості.

**Код програми 10.1. Демонстрація механізму роботи програми ведення інвентарної відомості, у якій використовується масив структур**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

const int rozmir = 100;

struct invStruct {           // Попереднє оголошення типу структури
      char nameProd[40];     // Назва продукції
      double vartProd;       // Вартість продукції
      double rozdrCina;      // Роздрібна ціна
      int nayavKilk;         // Наявна кількість
      int kilkDniv;          // Кількість днів до поновлення запасів
} prodArray[rozmir];         // Визначення масиву структур

// Попереднє оголошення процедур і функцій
void Enter(), InitList(), Display();
void UpDate(), GetPut(int i);
int MenuSelect();

int main()                   // Початок основної програми
{
      char vybir;
      InitList();  // Ініціалізація елементів масиву структур.

      for( ; ; ) {
            // Отримання команди меню, вибраної користувачем.
            vybir = MenuSelect();
            switch(vybir) {
                  // Введення запису в інвентарну відомість.
                  case 'e': Enter();
                        break;
                  // Відображення на екрані записів інвентарної відомості.
                  case 'd': Display();
                        break;
                  // Модифікування полів наявного запису відомості.
                  case 'u': UpDate();
                        break;
                  case 'q': return 0;
            }
      }
} // Кінець основної програми

// Визначення процедур і функцій
```





```
void InitList()     // Ініціалізація елементів масиву структур.
{
      // Ім'я нульової довжини означає порожній запис.
      for(int t=0; t<rozmir; t++) *prodArray[t].nameProd = '\0';
}

int MenuSelect()          // Отримання команди меню, вибраної користувачем.
{
      char ch; cout << endl;
      do {
            cout << "(E)nter" << endl; // Ввести новий запис.
            cout << "(D)isplay" << endl; // Відобразити всю відомість.
            cout << "(U)pdate" << endl; // Змінити поля запису.
            cout << "(Q)uit\n" << endl; // Вийти з програми.
            cout << "Виберіть команду: "; cin >> ch;
      } while(!strchr("eduq", tolower(ch)));

      return tolower(ch);
}

void Enter()                        // Введення запису в інвентарну відомість.
{
      int i;
      // Знаходимо першу вільну структуру.
      for(i=0; i<rozmir; i++)
            if(!*prodArray[i].nameProd) break;

      // Якщо масив повний, то значення "i" буде дорівнювати rozmir.
      if(i==rozmir) {
            cout << "Перелік повний" << endl;
            return;
      }
      GetPut(i);          // Введення інформації.
}

void GetPut(int i)        // Введення інформації.
{
      cout << "Назва продукції          : "; cin >> prodArray[i].nameProd;
      cout << "Вартість продукції : "; cin >> prodArray[i].vartProd;
      cout << "Роздрібна ціна  : "; cin >> prodArray[i].rozdrCina;
      cout << "Наявна кількість : "; cin >> prodArray[i].nayavKilk;
      cout << "Кількість днів до поновлення запасів: ";
      cin >> prodArray[i].kilkDniv;
}

void UpDate()   // Модифікування полів наявного запису відомості.
{
      int i; char name[80];
      cout << "Введіть назву продукції: "; cin >> name;
```

```
        for(i=0; i<rozmir; i++)
                if(!strcmp(name, prodArray[i].nameProd)) break;

        if(i==rozmir) {
                cout << "Продукцію не знайдено" << endl;
                return;
        }
        cout << "Введіть нову назву продукції" << endl;
        GetPut(i);              // Введення інформації.
}


void Display() // Відображення на екрані записів інвентарної відомості.
{
        for(int t=0; t<rozmir; t++)
                if(*prodArray[t].nameProd) {
                        cout << "Назва продукції:" << prodArray[t].nameProd << endl;
                        cout << "Вартість продукції:" << prodArray[t].vartProd << " грн" << endl;
                        cout << "У роздріб :" << prodArray[t].rozdrCina << " грн" << endl;
                        cout << "У наявності:" << prodArray[t].nayavKilk << " шт" << endl;
                        cout << "Залишилося :" << prodArray[t].kilkDniv << " днів" << endl;
                }
}
```

### 10.1.5. Механізм присвоєння структур

Вміст однієї структури можна присвоїти інший, якщо обидві ці структури мають однаковий тип. Наприклад, наведений нижче код програми присвоює значення структурної змінної strVar1 змінній strVar2.

**Код програми 10.2. Демонстрація механізму присвоєння значень структурним змінним**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

struct demoStruct {          // Попереднє оголошення типу структури
        int a, b;
};

int main()
{
        demoStruct strVar1, strVar2;
        strVar1.a = strVar1.b = 10;
        strVar2.a = strVar2.b = 20;

        cout << "Структури до присвоєння" << endl;
        cout << "strVar1: " << strVar1.a << " " << strVar1.b << endl;
        cout << "strVar2: " << strVar2.a << " " << strVar2.b << endl << endl;
        strVar2 = strVar1;          // Присвоєння структур
```





```
        cout << "Структури після присвоєння" << endl;
        cout << "strVar1: " << strVar1.a << " " << strVar1.b << endl;
        cout << "strVar2: " << strVar2.a << " " << strVar2.b << endl;

        getch(); return 0;
}
```

Внаслідок виконання цієї програми на моніторі буде відображено такі результати:

```
Структури до присвоєння.
strVar1: 10 10
strVar2: 20 20

Структури після присвоєння.
strVar1: 10 10
strVar2: 10 10
```

У мові програмування С++ кожне нове оголошення структури визначає новий тип. Отже, навіть якщо дві структури фізично однакові, але мають різні імена типів, компілятор вважатиме їх різними і не дасть змоги присвоїти значення однієї з них інший. Розглянемо такий фрагмент коду програми. Він некоректний і тому не буде компільована:

```
struct demoStructA {
        int a, b;
};

struct demoStructB {
        int a, b;
};

int main()
{
        demoStructA strVar1;
        demoStructB strVar2;

        strVar2 = strVar1; // Помилка через невідповідності типів.
        . . . . .
}
```

Незважаючи на те, що структури demoStructA і demoStructB фізично однакові, з погляду компілятора вони є окремими типами.

***Нео! хідноапа, 'ятати!а****Одну структуру змінну можна присвоїти іншій тільки у тому випадку, якщо обидві вони мають однаковий тип.*

### 10.1.6. Передача структури функції як аргументу

При передачі структури функції як аргумент використовується механізм передачі параметрів за значенням. Це означає, що будь-які зміни, внесені у вміст структури в тілі функції, якій вона передана, не впливають на структуру, що використовується як аргумент цією функцією. Проте необхідно мати

на увазі, що передача великих структур вимагає значних витрат системних ресурсів. Як правило, чим більше даних передається функції, тим більше витрачається системних ресурсів.

Використовуючи структуру як параметр, необхідно також пам'ятати, що тип аргументу повинен відповідати типу параметра. Наприклад, у наведеному нижче коді програми спочатку оголошується структура demoStruct, а потім функція FunA() приймає параметр типу demoStruct.

**Код програми 10.3. Демонстрація механізму передачі структури функції як аргументу**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

struct demoStruct {          // Попереднє оголошення типу структури
      int a;
      char ch;
};

void FunA(demoStruct pm);    // Попереднє оголошення прототипу функції

int main()
{
      demoStruct arg;        // Визначення змінної arg типу demoStruct.

      arg.a = 1000; arg.ch = 'x';
      FunA(arg);             // Передача структури функції як аргументу
      getch(); return 0;
}

void FunA(demoStruct pm)     // Визначення функції
{
      cout << "Передана функції структура: "
           << "a= " << pm.a << "; ch= " << pm.ch << endl;
}
```

Внаслідок виконання цієї програми на моніторі буде відображено такий результат:

```
Передана функції структура: a= 1000; ch= x
```

У цьому коді програми як аргумент arg у функції **main**(), так і параметр pm у функції FunA() мають однаковий тип. Тому аргумент arg можна передати функції FunA(). Якби типи цих структур були різні, у процесі компілювання коду програми було б видано повідомлення про помилку.

### 10.1.7. Повернення функцією структури як значення

Для повернення функцією структури як значення використовується механізм повернення параметрів за значенням. Це означає, що будь-які зміни,





внесені у вміст структури, яка визначена в тілі функції, впливають на структуру, якій присвоюється значення, що повертається цією функцією. Проте необхідно мати на увазі, що повернення великих структур вимагає значних витрат системних ресурсів. Як правило, чим більше даних повертається функцією, тим більше витрачається системних ресурсів.

Використовуючи структуру як параметр, необхідно також пам'ятати, що тип структури повинен відповідати типу повернутого значення. Наприклад, у наведеному нижче коді програми спочатку оголошується структура demoStruct, а потім функція FunA() приймає параметр типу demoStruct.

**Код програми 10.4. Демонстрація механізму повернення функцією структури як значення**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

struct demoStruct {          // Попереднє оголошення типу структури
      int a;
      char ch;
};

void FunA(demoStruct pm, char zm);      // Попереднє оголошення прототипу функції
demoStruct FunB(demoStruct pm);         // Попереднє оголошення прототипу функції

int main()
{
      demoStruct arg1={1000,'x'}, arg2;     // Визначення структурних змінних

      FunA(arg1,'1');            // Передача структури функції як аргументу
      arg2 = FunB(arg1);         // Повернення функцією структури як значення
      FunA(arg2,'2');            // Передача структури функції як аргументу

      getch(); return 0;
}

void FunA(demoStruct pm, char zm)       // Визначення функції
{
      cout << "Передана функції структура arg" << zm
           << ": a= " << pm.a << "; ch= " << pm.ch << endl;
}

demoStruct FunB(demoStruct pm)          // Визначення функції
{
      demoStruct pn;

      pn.a = 3*pm.a; pn.ch = 'y';
      return pn;
}
```

Внаслідок виконання цієї програми на моніторі буде відображено такий результат:

```
Передана функції структура arg1: a= 1000; ch= x
Передана функції структура arg2: a= 3000; ch= y
```

У цьому коді програми як аргументи arg1 і arg2 у функції **main**(), так і параметр pm у функції FunA() мають однаковий тип. Аналогічно у функції FunB() визначена структурна змінна pn має цей самий тип. Тому аргументи arg1 і arg2 можна передати функції FunA(), а значення структурної змінної pn можна повернути з функції FunB() у функції **main**(). Якби типи цих структур були різні, у процесі компілювання коду програми було б видано повідомлення про помилку. Окрім цього, у функції FunA() як аргумент передається проста змінна типу **char** для ідентифікування назви структурної змінної.

## 10.2. Механізм використання покажчиків на структури і оператора "стрілка"

У мові програмування C++ покажчики на структури можна використовувати так само, як і покажчики на змінні будь-якого іншого типу. Проте використання покажчиків на структури має ряд особливостей, які необхідно враховувати.

### 10.2.1. Механізм використання покажчиків на структури

Покажчик на структуру оголошується так само, як покажчик на будь-яку іншу змінну, тобто за допомогою символу "*", поставленого перед іменем структурної змінної. Наприклад, використовуючи визначену вище структуру invStruct, можна записати таку настанову, яка оголошує змінну invPointer покажчиком на дані типу invStruct:

```
invStruct *invPointer;
```

Щоб знайти адресу структурної змінної, необхідно перед її іменем розмістити оператор "&". Наприклад, припустимо, що за допомогою наведеного нижче коду програми ми оголошуємо структуру, визначаємо структурну змінну і покажчик на структуру оголошеного нами типу:

```
struct balStruct {                  // Попереднє оголошення типу структури
      float balance;
      char name[80];
} person;          // Визначення структурної змінної

balStruct *p;      // Визначаємо покажчик на структуру.
```

Тоді у процесі виконання настанови

```
p = &person;
```

у покажчик p буде поміщено адресу структурної змінної person.

До членів структури можна отримати доступ за допомогою покажчика на цю структуру. Але у цьому випадку використовується не оператор "крапка", а оператор "->". Наприклад, у процесі виконання такої настанови ми отримуємо доступ до поля balance через покажчик p:

```
p->balance
```

Оператор "->" називається оператором "стрілка". Він утворюється з використанням знаків "мінус" і "більше".

---

*Оператор "стрілка" (->) дає змогу отримати доступ до членів структури за допомогою покажчика.*

---

Покажчик на структуру можна використовувати як параметр функції. Важливо пам'ятати про такий спосіб передачі параметрів, оскільки він працює набагато швидше, ніж у випадку, коли функції "власною персоною" передається об'ємна структура[1].

---

***Нео! хідноапа, 'ятати!а****Щоб отримати доступ до членів структури, необхідно використовувати оператор "крапка". Щоб отримати доступ до членів структури за допомогою покажчика, потрібно використовувати оператор "стрілка".*

---

### 10.2.2. Приклад використання покажчиків на структури

Використання покажчиків на структури можна розглянути на прикладі функцій часу і дати, які часто використовується у мові C++, призначена для зчитування значення поточного системного часу і дати. Для цього у програму необхідно включити заголовок <**ctime**>, який підтримує два типи дати, що вимагаються згаданими функціями. Один з цих типів, **time_t** призначений для представлення системного часу і дати у вигляді довгого цілочисельного значення, яке використовується як *календарний час*. Другий тип є структурою tm, яка містить окремі елементи дати і часу. Таке представлення часу називають *по-елементним*. Структура tm має такий формат:

```
struct tm {        // Попереднє оголошення типу структури часу і дати
      int tm_sec;     /* секунди, 0-61 */
      int tm_min;     /* хвилини, 0-59 */
      int tm_hour;    /* годинник, 0-23 */
      int tm_mday;    /* день місяця, 1-31 */
      int tm_mon;     /* місяць, починаючи з січня, 0-11 */
      int tm_year;    /* рік після 1900 */
      int tm_wday;    /* день, починаючи з неділі, 0-6 */
      int tm_yday;    /* день, починаючи з 1-го січня, 0-365 */
      int tm_isdst    /* індикатор літнього часу */
};
```

Значення індикатора tm_isdst є позитивним, якщо діє режим літнього часу (Daylight Saving Time), дорівнює нулю, якщо не діє, і є негативним, якщо інформація про це недоступна.

Основним засобом визначення часу і дати у мові C++ є функція **time**(), яка має такий прототип:

[1] Передача покажчика завжди відбувається швидше, ніж передача самої структури.

**time_t time(time_t *curtime);**

Функція **time**() повертає поточний календарний час системи. Якщо в системі відлік часу не здійснюється, повертається значення -1. Функцію **time**() можна викликати або з нульовим покажчиком, або з покажчиком на змінну *curtime* типу **time_t**. У останньому випадку цій змінній буде присвоєне значення поточного календарного часу.

Щоб перетворити календарний час в по-елементний, необхідно використати функцію **localtime**(), яка має такий прототип:

**struct tm *localtime(const time_t *curtime);**

Функція **localtime**() повертає покажчик на по-елементну форму параметра *curtime*, представленого у вигляді структури tm. Значення *curtime* вказує на локальний час. Його зазвичай отримують за допомогою функції **time**().

Структура, що використовується функцією **localtime**() для зберігання часу в по-елементній формі, розміщується в пам'яті статично і перезаписується під час кожного виклику цієї функції. Якщо потрібно зберегти вміст цієї структури, потрібно скопіювати його в яку-небудь іншу область пам'яті.

Наведений нижче код програми демонструє використання функцій **time**() і **localtime**() для відображення на екрані поточного системного часу.

**Код програми 10.5. Демонстрація механізму використання функцій time() і localtime() для відображення поточного системного часу**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
#include <ctime>             // Використання системного часу і дати
using namespace std;         // Використання стандартного простору імен

int main()
{
      struct tm *ptr;
      time_t lt;

      lt = time('\0');
      ptr = localtime(&lt);

      cout << ptr->tm_hour << ":" << ptr->tm_min;
      cout << ":" << ptr->tm_sec << endl;

      getch(); return 0;
}
```

Ось один з можливих результатів виконання цієї програми:

```
20:32:44
```

Незважаючи на те, що у наведеному вище коді програми може використовуватися по-елементна форма представлення часу і дати, найпростіше згенерувати рядок часу і дати за допомогою функції **asctime**(), прототип якої має такий вигляд:

**char *asctime(const struct tm *ptr);**





Функція **asctime**() повертає покажчик на рядок, який містить результат перетворення інформації, яка зберігається в ptr структурі, що адресується покажчиком, і має такий формат:

*день місяць число час:хвилини:секунди рік\n\0*

Покажчик на структуру, що передається функції **asctime**(), часто отримують за допомогою функції **localtime**().

Область пам'яті, яка використовується функцією **asctime**() для зберігання відформатованого рядка результату, є символьним масивом (що статично виділяється в пам'яті), який перезаписується під час кожного виклику цієї функції. Якщо потрібно зберегти вміст цього рядка, то необхідно скопіювати його в яку-небудь іншу область пам'яті.

У наведеному нижче коді програми продемонстровано механізм використання функції **asctime**() для відображення системного часу і дати.

**Код програми 10.6. Демонстрація механізму використання функції asctime() для відображення системного часу і дати**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
#include <ctime>             // Використання системного часу і дати
using namespace std;         // Використання стандартного простору імен

int main()
{
      struct tm *ptr;
      time_t lt = time('\0');

      ptr = localtime(&lt);
      cout << asctime(ptr);

      getch(); return 0;
}
```

Ось один з можливих результатів виконання цієї програми.

```
Sun Sep 28 17:20:35 2010
```

У мові програмування С++ передбачені й інші функції дати і часу, з якими можна познайомитися, звернувшись до документації, що додається до Вашого компілятора.

## 10.3. Посилання на структури

Для доступу до структури можна використовувати посилання. Посилання на структуру часто використовують як параметр функції або значення, що повертається функцією. Під час отримання доступу до членів структури за допомогою посилання використовують оператор "крапка"[1].

[1] Оператор "стрілка" зарезервований для доступу до членів структури за допомогою покажчика.

### 10.3.1. Механізм використання структур при передачі функції параметрів за посиланням

У наведеному нижче коді програми показано, як можна використовувати структуру при передачі функції параметрів за посиланням.

**Код програми 10.7. Демонстрація механізму використання посилання на структуру**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

struct myStruct {            // Попереднє оголошення типу структури
      int a, b;
};

// Попереднє оголошення прототипу функції, яка отримує і повертає посилання на структуру
myStruct &Fun2(myStruct &var);

int main()
{
      myStruct strX, strY;
      strX.a = 10; strX.b = 20;

      cout << "Початкові значення полів : strX.a= "
           << strX.a << "; strX.b= " << strX.b << endl;

      strY = Fun2(strX);

      cout << "Модифіковані значення полів: strX.a= "
           << strX.a << "; strX.b= " << strX.b << endl;
      cout << "Модифіковані значення полів: strY.a= "
           << strY.a << "; strY.b= " << strY.b << endl;

      getch(); return 0;
}

// Визначення Функції, яка отримує і повертає посилання на структуру
myStruct &Fun2(myStruct &var)
{
      var.a = var.a * var.a;
      var.b = var.b / var.b;
      return var;
}
```

Ось результати виконання цієї програми.

```
Початкові значення полів: strX.a= 10; strX.b= 20
Модифіковані значення полів: strX.a= 100; strX.b= 1
Модифіковані значення полів: strY.a= 100; strY.b= 1
```





Зважаючи на істотні витрати системних ресурсів, що використовуються на передачу структури функції (або при поверненні її функцією) багато С++-програмістів для виконання таких завдань використовують посилання на структури.

### 10.3.2. Механізм використання як членів структур масивів і структур

Кожен член структури може мати будь-який допустимий тип даних, у тому числі і такі складені типи, як масиви й інші структури. Оскільки ця тема пов'язана з певними труднощами для програмістів, то зупинимося на ній детальніше.

Масив, який використовується як член структури, обробляється цілком звичайним способом. Розглянемо таку структуру:

```
struct demoStruct {      // Попереднє оголошення типу структури
      int Array[10][10]; // Цілочисельний масив розміром 10x10.
      float b;
} var;                   // Визначення структурної змінної
```

Щоб присвоїти певне значення елементу масиву Array з "координатами" 3×7 в структурі var типу demoStruct, необхідно записати таку настанову:

```
var.Array[3][7] = 2.37;
```

Як показано у цьому прикладі, якщо масив є членом структури, то для доступу до елементів цього масиву індексується ім'я масиву, а не ім'я структури.

Якщо певна структура є членом іншої структури, то вона називається *вкладеною структурою*. У наведеному нижче прикладі структура addrStruct вкладена у структуру empStruct:

```
struct addrStruct {              // Попереднє оголошення адреси структури
      char name[40];   // Прізвище службовця
      char street[40]; // Вулиця
      char city[40];   // Місто
      char zip[10];    // Поштовий індекс
};

struct empStruct {               // Попереднє оголошення реквізитів структури
      addrStruct address;        // Адреса службовця
      float wage;                // Оклад службовця
} worker;
```

Тут структура empStruct має два члени. Першим членом є структура типу addrStruct, яка міститиме адресу службовця. Другим членом є змінна wage, яка зберігає його оклад. У процесі виконання наведеної нижче настанови полю zip структури address, яка є членом структури worker, буде присвоєно поштовий індекс 76285:

```
worker.address.zip = 76285;
```

Як бачимо, члени структур вказуються зліва направо, – від крайньої зовнішньої до найдальшої внутрішньої.

Структура також може містити як свій член покажчик на цю ж структуру. Тобто для структури цілком допустимо містити член, який є покажчиком на неї саму. Наприклад, в такій структурі змінна strP оголошується як покажчик на структуру типу myStruct, тобто на оголошувану тут структуру:

```
struct myStruct {                // Попереднє оголошення типу структури
    int a;
    char str[80];
    myStruct *strP;              // Покажчик на структуру типу myStruct
};
```

Структури, що містять покажчики на самих себе, часто використовують при створенні таких структур даних, як зв'язні списки. У міру вивчення мови C++ Вам обов'язково потраплять програмні коди, у яких застосовуються подібні речі.

### 10.3.3. Порівняння C- і C++-структур

C++-структури – нащадки C-структур. Отже, будь-яка C-структура також є дійсною C++-структурою. Між ними, проте, існують важливі відмінності, а саме:

- ***по-перше***, як буде показано в наступному розділі, C++-структури мають деякі унікальні атрибути, які дають змогу їм підтримувати об'єктно-орієнтоване програмування;
- ***по-друге***, у мові C структура не оголошує насправді новий тип даних. Цим може "похвалитися" тільки C++-структура.

Як було сказано вище, оголошуючи структуру у мові програмування C++, оголошуємо новий тип, який називається за іменем цієї структури. Цей новий тип можна використовувати для визначення змінних, визначення значень, що повертаються функціями, і т.ін. Проте у мові C ім'я структури називається її *тегом* (або *дескриптором*). *А* тег сам по собі не є іменем типу.

Щоб зрозуміти цю відмінність, розглянемо такий фрагмент C-коду:

```
struct cStruct {
    int a, b;
};
// Попереднє оголошення змінної cStruct
struct cStruct strVar;
```

Зверніть увагу на те, що наведене вище оголошення структури точно таке ж саме, як у мові C++. Тепер уважно розглянемо визначення структурної змінної strVar. Воно також починається з ключового слова **struct**. У мові C після оголошення структури для повного задавання типу даних все одно потрібно використовувати ключове слово **struct** спільно з тегом цієї структури (у цьому випадку з ідентифікатором cStruct).

Якщо Вам доведеться перетворювати старі C-програми у код програми мовою C++, не варто турбуватися про відмінності між C- і C++-структурами, оскільки мова C++, як і раніше, приймає C-орієнтовані оголошення. Наприклад, попередній фрагмент C-коду програми коректно скомпілюється як частина будь-якої C++-програми. З погляду компілятора мови C++ у визначенні змінної strVar зайвим використано тільки ключове слово **struct**, без якого мова C++ може обійтися.





## 10.4. Поняття про бітові поля структур

На відміну від багатьох інших комп'ютерних мов, у мові C++ передбачено вбудований спосіб доступу до конкретного розряду байта. Побітовий доступ можливий шляхом використання бітових полів. Бітові поля можуть виявитися корисними в різних ситуаціях. Наведемо всього три приклади

- *по-перше*, якщо ми маємо справу з обмеженим об'ємом пам'яті, можна зберігати декілька булевих (логічних) значень в одному байті;
- *по-друге*, деякі інтерфейси пристроїв передають інформацію, закодовану саме в бітах;
- *по-третє*, існують підпрограми кодування, яким потрібен доступ до окремих бітів у рамках байта.

Реалізація всіх цих функцій можлива за допомогою порозрядних операторів, як це було показано в попередньому розділі, але бітове поле може зробити нашу програму прозорішою і читабельною, а також підвищити її переносність.

***Бітове поле*** *– біт-орієнтований член структури.*

Метод, який використовується у мові C++ для доступу до бітів, базується на застосуванні структур. *Бітове поле* – спеціальний тип члена структури, який визначає свій розмір у бітах. Загальний формат визначення бітових полів є таким:

```
struct ім'я_типу_структури {
    тип ім'я_1: довжина;
    тип ім'я_2: довжина;
    . . . . . .
    тип ім'я_n: довжина;
};
```

У цьому записі елемент *тип* означає тип бітового поля, а елемент *довжина* – кількість бітів у цьому полі. Бітове поле повинно бути оголошено як значення цілого типу або перерахунку. Бітові поля завдовжки 1 біт оголошуються як значення типу без знаку (**unsigned**), оскільки єдиний біт не може мати знакового розряду.

Бітові поля зазвичай використовують для аналізу вхідних даних, які приймаються від пристроїв, що входять до складу устаткування системи. Наприклад, порт станів послідовного адаптера зв'язку може повертати байт стану, організований так:

| Біт | Значення у встановленому стані |
|---|---|
| 0 | зміна в лінії установки в початковий стан |
| 1 | зміна в лінії готовності даних |
| 2 | виявлений задній фронт |

| | |
|---|---|
| 3 | зміна в лінії прийому даних |
| 4 | встановлення в початковий стан |
| 5 | дані готові |
| 6 | телефонний сигнал виклику |
| 7 | сигнал прийнято |

Для представлення інформації, яка міститься в байті станів, можна використовувати такі бітові поля.

```
struct status_type {
    unsigned delta_cts: 1;   // Зміна в лінії установки в початковий стан
    unsigned delta_dsr: 1;   // Зміна в лінії готовності даних
    unsigned tr_edge: 1;     // Виявлений задній фронт
    unsigned delta_rec: 1;   // Зміна в лінії прийому даних
    unsigned cts:     1;     // Встановлення в початковий стан
    unsigned dsr:     1;     // Дані готові
    unsigned ring:    1;     // Телефонний сигнал виклику
    unsigned rec_line: 1;    // Сигнал прийнято
} status;
```

Щоб визначити, коли можна відправити або отримати дані, використовується такий програмний код:

```
status = get_port_status();
if(status.cts) cout << "Встановлення в початковий стан";
if(status.dsr) cout << "Дані готові";
```

Щоб присвоїти бітовому полю значення, достатньо використовувати таку ж форму, яка зазвичай застосовується для елемента структури будь-якого іншого типу. Наприклад, настанова

```
status.ring = 0;
```

очищає бітове поле ring. Як видно з цих прикладів, доступ до кожного бітового поля можна отримати за допомогою оператора "крапка". Але, якщо загальний доступ до структури здійснюється через покажчик, необхідно використовувати оператор "->".

Потрібно мати на увазі, що зовсім необов'язково присвоювати ім'я кожному бітовому полю. Це дає змогу звертатися тільки до потрібних бітів, "обходячи" інші. Наприклад, якщо нас цікавлять тільки біти cts і dsr, то ми могли б оголосити структуру status_type так:

```
struct status_type {
    unsigned: 4;
    unsigned cts: 1;
    unsigned dsr: 1;
} status;
```

Необхідно звернути тут увагу на те, що біти після останнього іменованого dsr немає потреби взагалі згадувати.

У структурі можна змішувати "звичайні" члени з бітовими полями. Наприклад,

```
struct empStruct {
    struct addrStruct address;
```





```
    float pay;
    unsigned lay_off: 1;      // працює чи ні
    unsigned hourly: 1;       // погодинна оплата або оклад
    unsigned deductions: 3; // утримання податку
};
```

Ця структура визначає запис по кожному службовцю, у якій використовується тільки один байт для зберігання трьох елементів інформації: статус службовця, характер оплати його праці (погодинна оплата або фіксований оклад) і податкова ставка. Без використання бітових полів для зберігання цієї інформації довелося б зайняти три байти.

Використання бітових полів має певні обмеження. Програміст не може отримати адресу бітового поля або посилання на нього. Бітові поля не можна зберігати в масивах. Їх не можна оголошувати статичними. При переході від одного комп'ютера до іншого неможливо знати впевнено порядок проходження бітових полів: справа наліво або зліва направо. Це означає, що будь-яка програма, у якій використовуються бітові поля, може страждати певною залежністю від марки комп'ютера. Можливі й інші обмеження, пов'язані з особливостями реалізації компілятора мови С++. Тому, перш ніж використовувати бітові поля структур, доцільно насамперед з'ясувати ці питання у відповідній документації до Вашого компілятора.

У наступному підрозділі представлено код програми (прог. 10.8), у якому для відображення символьних ASCII-кодів у двійковій системі числення використовуються бітові поля.

## 10.5. Механізм використання об'єднань

Об'єднання (англ. union) – тип даних користувача, який дуже схожий на структуру, проте тут всі дані займають одну і ту ж область пам'яті. Тому розмір об'єднання дорівнює розміру його найбільшого члена. У будь-який момент часу об'єднання зберігає значення тільки одного з членів. Тобто на якомусь етапі Вам потрібно один тип даних, на іншому – інший. Тому, загалом, об'єднання економить пам'ять комп'ютера від непотрібних на даному етапі змінних.

Оскільки об'єднання зберігає і використовує завжди одне поле їх множини на вибір, то виникає питання про виділення пам'яті під це поле. Тут принцип зрозумілий – вибирається найбільший з типів даних. У мові програмування С++ до об'єднання звертаються так само, як і до структури: через символ "->" при використанні покажчика, або "." при використанні звичайної змінної.

### 10.5.1. Оголошення об'єднання

Об'єднання складається з декількох змінних, які розділяють між собою одну і ту саму область пам'яті. Це означає, що об'єднання забезпечує можливість інтерпретації однієї і тієї ж самої конфігурації бітів двома (або більше)

різними способами. Оголошення об'єднання, як неважко переконатися на наведеному нижче прикладі, є подібним до оголошення структури:

```
union myUnion {                 // Попереднє оголошення типу об'єднання
    short int c;
    char ch;
};
```

У наведеному прикладі оголошується об'єднання, у якому значення типу **short int** і значення типу **char** розділяють одну і ту саму область пам'яті. Необхідно відразу ж з'ясувати один момент: неможливо зробити так, щоб це об'єднання зберігало і цілочисельне значення, і символ одночасно, оскільки змінні c та ch накладаються (у пам'яті) одне на друге. Але програма у будь-який момент може обробляти інформацію, що міститься у цьому об'єднанні, як цілочисельне значення або як символ. Отже, об'єднання забезпечує два (або більше) способи представлення однієї і тієї ж самої множини даних. Як видно з цього прикладу, об'єднання оголошується за допомогою ключового слова **union**.

Як і під час оголошення структур, так і під час оголошення об'єднання не визначається жодна змінна. Змінну можна визначити, розмістивши її ім'я в кінці оголошення або скориставшись окремою настановою визначення. Щоб визначити змінну об'єднання іменем uVar типу myUnion, достатньо записати:

```
myUnion uVar;
```

У змінній об'єднання uVar як змінна c типу **short int**, так і символьна змінна ch розділяють одну і ту саму область пам'яті[1]. Як змінні c та ch розділяють одну область пам'яті, показано на рис. 10.2:

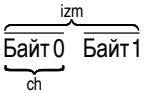



**Рис. 10.2. Змінні c та ch разом використовують об'єднання uVar**

Під час оголошення об'єднання компілятор автоматично виділяє область пам'яті, достатню для зберігання в об'єднанні змінних найбільшого за об'ємом типу.

Щоб отримати доступ до елемента об'єднання, використовують такий самий синтаксис, який застосовується і для структур: оператори "крапка" і "стрілка". При безпосередньому зверненні до об'єднання (або за допомогою посилання) використовують оператор "крапка". Якщо ж доступ до змінної об'єднання здійснюється через покажчик, використовують оператор "стрілка". Наприклад, щоб присвоїти букву "A" елемента ch об'єднання uVar, достатньо використати таку настанову:

```
uVar.ch = "A";
```

У наведеному нижче прикладі функції Fun1() передається посилання на об'єднання uVar. У тілі цієї функції за допомогою покажчика змінної c присвоюється значення 10:

[1] Безумовно, змінна i займає два байти, а символьна змінна ch використовує тільки один.





```
    Fun1(&uVar);    // Передаємо функції Fun1() покажчик
        // на об'єднання uVar.
        //...
}

void Fun1(myUnion *un)
{
    un->c =10; // Присвоюємо число 10 члену об'єднання uVar через покажчик.
}
```

Оскільки об'єднання дають змогу складеній програмі інтерпретувати одні і ті ж самі дані по-різному, то вони часто використовують у випадках, коли потрібне незвичайне перетворення типів. Наприклад, наведений нижче код програми використовує об'єднання для переставляння двох байтів, які становлять коротке цілочисельне значення. Тут для відображення вмісту цілочисельних змінних використовується функція DispBit(), розроблена в розд. 9[1].

**Код програми 10.8. Демонстрація механізму використання об'єднання для переставляння двох байтів, які становлять коротке цілочисельне значення**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

void DispBit(unsigned u);
union swapBytes {            // Попереднє оголошення типу об'єднання
    short int num;
    char ch[2];
};

int main()
{
    swapBytes bitS;
    char tmp;
    bitS.num = 15; // Двійковий код: 0000 0000 0000 1111

    cout << "Початкові байти:";
    DispBit(bitS.ch[1]); cout << " ";
    DispBit(bitS.ch[0]); cout << endl << endl;

    // Обмін байтів.
    tmp = bitS.ch[0]; bitS.ch[0] = bitS.ch[1]; bitS.ch[1] = tmp;

    cout << "Байти після переставляння:";
    DispBit(bitS.ch[1]); cout << " ";
    DispBit(bitS.ch[0]); cout << endl << endl;
    getch(); return 0;
}
```

[1] Ця програма написана в припущенні, що короткі цілочисельні значення мають довжину два байти.

```
// Відображення бітів, з яких складається байт.
void DispBit(unsigned u)
{
        register int t;
        for(t=128; t>0; t=t/2)
                if(u & t) cout << "1 ";
                else cout << "0 ";
}
```

У процесі виконання програма відображає на екрані такі результати:

```
Початкові байти          : 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 1 1
Байти після переставляння : 0 0 0 0 1 1 1 1 0 0 0 0 0 0 0 0
```

У цьому коді програми цілочисельній змінній bitS.num присвоюється число 15. Перестановка двох байтів, що становлять це значення, здійснюється шляхом обміну двох символів, які утворюють масив ch. Як наслідок, старший і молодший байти цілочисельної змінної num міняються місцями. Ця операція можлива тільки тому, що як змінна num, так і масив ch розділяють одну і ту саму область пам'яті.

У наведеному нижче коді програми продемонстровано ще один приклад використання об'єднання. Тут об'єднання пов'язуються з бітовими полями, що використовуються для відображення в двійковій системі числення ASCII-коду програми, які генерується при натисненні будь-якої клавіші. Ця програма також демонструє альтернативний спосіб відображення окремих бітів, що становлять байт. Об'єднання дає змогу присвоїти значення натиснутої клавіші символьної змінної, а бітові поля використовуються для відображення окремих бітів.

**Код програми 10.9. Демонстрація механізму відображення ASCII-коду програми символів у двійковій системі числення**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

// Бітові поля, які будуть розшифровані.
struct byteStruct {          // Попереднє оголошення типу структури
        unsigned a : 1;
        unsigned b : 1;
        unsigned c : 1;
        unsigned d : 1;
        unsigned e : 1;
        unsigned f : 1;
        unsigned g : 1;
        unsigned h : 1;
};
union bitsUnion {            // Попереднє оголошення типу об'єднання
        char ch;
        struct byteStruct bit;
} ascii;
```





```
void DispBits(bitsUnion b);

int main()
{
        do {
                cin >> ascii.ch; cout << ": ";
                DispBits(ascii);
        } while(ascii.ch !='q'); // Вихід при введенні букви 'q'.
        getch(); return 0;
}

// Відображення конфігурації бітів для кожного символу.
void DispBits(bitsUnion bitB)
{
        if(bitB.bit.h) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.g) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.f) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.e) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.d) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.c) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.b) cout << "1 ";
        else cout << "0 ";
        if(bitB.bit.a) cout << "1 ";
        else cout << "0 ";
        cout << endl;
}
```

Ось як буде виглядати один з можливих варіантів виконання цієї програми.

```
a: 0 1 1 0 0 0 0 1
b: 0 1 1 0 0 0 1 0
c: 0 1 1 0 0 0 1 1
d: 0 1 1 0 0 1 0 0
e: 0 1 1 0 0 1 0 1
f: 0 1 1 0 0 1 1 0
g: 0 1 1 0 0 1 1 1
h: 0 1 1 0 1 0 0 0
i: 0 1 1 0 1 0 0 1
j: 0 1 1 0 1 0 1 0
k: 0 1 1 0 1 0 1 1
l: 0 1 1 0 1 1 0 0
m: 0 1 1 0 1 1 0 1
n: 0 1 1 0 1 1 1 0
o: 0 1 1 0 1 1 1 1
p: 0 1 1 1 0 0 0 0
q: 0 1 1 1 0 0 0 1
```

***Важливо!*** *Оскільки об'єднання припускає, що декілька змінних розділяють одну і ту саму область пам'яті, то цей засіб дає змогу програмісту зберігати інформацію, яка (залежно від ситуації) може містити різні типи даних, і отримувати доступ до цієї інформації. По суті, об'єднання забезпечують низькорівневу підтримку принципів поліморфізму. Іншими словами, об'єднання забезпечує єдиний інтерфейс для декількох різних типів даних, втілюючи таким чином концепцію "один інтерфейс − багато методів" у своїй найпростішій формі.*

### 10.5.2. Поняття про анонімні об'єднання

У мові програмування C++ передбачено спеціальний тип об'єднання, який називається *анонімним*. Анонімне об'єднання не має назви типу, і тому об'єкт такого об'єднання визначити неможливо. Але анонімне об'єднання повідомляє компіляторові про те, що його члени розділяють одну і ту саму область пам'яті. При цьому звернення до самих змінних об'єднання відбувається безпосередньо, без використання оператора "крапка".

Розглянемо такий приклад.

**Код програми 10.10. Демонстрація механізму використання анонімного об'єднання**

```
#include <vcl>
#include <iostream>          // Потокове введення-виведення
#include <conio>             // Консольний режим роботи
using namespace std;         // Використання стандартного простору імен

int main()
{
    union {    // Попереднє оголошення анонімного об'єднання.
        short int count;
        char ch[2];
    };
    // Ось як відбувається безпосереднє звернення до членів анонімного об'єднання.
    ch[0] = 'x'; ch[1] = 'y';
    cout << "Об'єднання як символи: " << ch[0] << ch[1] << endl;
    cout << "Об'єднання як цілі значення: " << count << endl;

    getch(); return 0;
}
```

Ця програма відображає наступний результат.

```
Об'єднання у вигляді символів: xy
Об'єднання у вигляді цілого значення: 31096
```

Число 31096 отримане внаслідок занесення символів x і y в молодший і старший байти змінної *count* відповідно. Як бачите, до обох змінних, що входять до складу об'єднання, як *count*, так і ch, можна отримати доступ так само, як до звичайних змінних, а не як до складових об'єднання. Незважаючи на те, що вони оголошені як частина анонімного об'єднання, їх імена знаходяться на тому ж самому рівні області видимості, що і інші локальні змінні, які оголошено на рівні об'єднання. Таким чином, член анонімного об'єднання не може мати імені, що збігається з іменем будь-якої іншої змінної, оголошеної в тій самій області видимості.





Анонімне об'єднання є засобом, за допомогою якого програміст може повідомити компілятор про свій намір, щоб дві (або більше) змінні розділяли одну і ту саму область пам'яті. За винятком цього моменту, члени анонімного об'єднання поводяться подібно до будь-яких інших змінних.

### 10.5.3. Механізм використання оператора sizeof для гарантії переносності коду програми

Як було показано вище, структури та об'єднання створюють об'єкти різних розмірів, які залежать від розмірів і кількості їх членів. Окрім цього, розміри таких вбудованих типів як **int** можуть змінюватися при переході від одного комп'ютера до іншого. Іноді компілятор заповнює структуру або об'єднання так, щоб вирівняти їх по границі парного слова або абзацу[1]. Тому, якщо у програмі потрібно визначити розмір (у байтах) структури або об'єднання, використовують оператор **sizeof**. Не намагайтеся вручну виконувати додавання окремих членів. Через заповнення або інші апаратно-залежні чинники розмір структури або об'єднання може виявитися більшим від суми розмірів окремих їх членів.

***Вартоа' нати!*** *Об'єднання завжди займатиме область пам'яті, достатню для зберігання його найбільшого члена.*

Розглянемо такий короткий приклад:

```
union xUnion {  // Попереднє оголошення типу об'єднання
    char ch;
    int c;
    double f;
} uVar;             // Визначення змінної об'єднання
```

Тут у процесі виконання оператора **sizeof** для обчислення розміру об'єднання uVar отримаємо результат 8 (за умови, що **double**-значення займає 8 байтів). У процесі виконання програми немає значення, що реально зберігатиметься в змінній uVar; тут важливим є розмір найбільшої змінної, що входить до складу об'єднання, оскільки об'єднання повинно мати розмір найбільшого його елемента.

[1] Абзац містить 16 байтів.

# Додаток А. ОСОБЛИВОСТІ ЗАСТОСУВАННЯ С-СИСТЕМИ ВВЕДЕННЯ-ВИВЕДЕННЯ ДАНИХ

Цей додаток містить короткий опис С-системи введення-виведення даних. Незважаючи на те, що Ви не передбачаєте використовувати С++-систему введення-виведення даних, проте є ряд причин, згідно з якими Вам все-таки необхідно розуміти основи орієнтованої С-системи введення-виведення. По-перше, якщо Вам доведеться працювати з С-кодом (особливо, якщо виникне потреба його перекладу у С++-код), то Вам потрібно знати, як працює С-система введення-виведення. По-друге, часто в одній і тій самій програмі використовуються як С-, так і С++-операції введення-виведення даних. Це особливо характерний для дуже великих програм, окремі частини яких писалися різними програмістами протягом достатньо тривалого періоду часу. По-третє, велика кількість тих, що існують С-програм продовжують знаходитися в експлуатації і потребують підтримки. Нарешті, багато книг і періодичних видань містять програми, написаних мовою програмування С. Щоб розуміти ці С-програми, необхідно розуміти основи функціонування С-системи введення-виведення.

***Нео! хідноапа, 'ятати!*** *Для С++-програм необхідно використовувати об'єктно-орієнтовану С++-систему введення-виведення.*

У цьому додатку описані найбільш використовувані С-орієнтовані функції введення-виведення. Проте стандартна С-бібліотека містить таку величезну кількість функцій введення-виведення даних, які ми не в стані розглянути їх тут у повному об'ємі. Якщо ж Вам доведеться серйозно зануритися у С-програмування, то рекомендуємо звернутися до довідкової літератури.

Система введення-виведення мови С вимагає містити у програми заголовний файл **stdio.h** (йому відповідає заголовок <**cstdio**>, що відповідає новому стилю). Кожна С-програма повинна використовувати заголовний файл **stdio.h**, оскільки мова С не підтримує С++-стиль внесення заголовків. С++-програма може виконувати дії з використанням будь-якого з цих двох варіантів. Заголовок <**cstdio**> поміщає свій вміст у простір імен **std**, а заголовний файл **stdio.h** – у глобальний простір імен, що відповідає С-орієнтації. У цьому додатку як приклади наведені С-програми, тому вони використовують С-стиль внесення заголовного файлу **stdio.h** і не вимагають встановлення простору імен.

***Нео! хідноапа, 'ятати!*** *Як наголошувалося в розд. 1, стандарт мови С був оновлений у 1999 р. і отримав назву стандарту С99. У той час у С-систему введення-виведення було внесено декілька удосконалень. Але оскільки мова С++ опирається на стандарт С89, то він не підтримує засобів, які були додані у стандарт С99. Понад це, на момент написання цього навчального посібника жоден з широко доступних компіляторів С++ не підтримував стандарт С99. Та і жодна з широко поширюваних програм не використовувала засоби стандарту С99. Тому тут не описуються засоби, внесені у С-систему введення-виведення стандартом С99. Якщо ж Вас цікавить мова С, в т.ч. повний опис його системи введення-виведення і засобів, доданих стандартом С99, рекомендуємо звернутися до книги Герберта Шілдта "Полный справочник по С++".*





## А.1. Механізм використання потоків у С-системі введення-виведення даних

Подібно С++-системі введення-виведення, С-орієнтована система введення-виведення базується на понятті потоку. На початку роботи програми автоматично відкриваються три наперед певні текстові потоки: **stdin**, **stdout** і **stderr**. Їх називають стандартними *потоками введення даних* (*вхідний потік*), *виведення даних* (*вихідний потік*) *і помилок* відповідно. (Деякі компілятори відкривають також і інші потоки, які залежать від конкретної реалізації системи.) Ці потоки є С-версії потоків з **in**, **cout** і **cerr** відповідно. За замовчуванням вони пов'язані з відповідним системним пристроєм:

| Потік | Пристрій |
|---|---|
| **stdin** | клавіатура |
| **stdout** | екран |
| **stderr** | екран |

Кожен потік, що пов'язаний з файлом, має структуру керування файлом типу **FILE**. Ця структура визначена у заголовному файлі **stdio.h**. Ви не повинні модифікувати вміст цього блоку керування файлом.

***Нео! хідноапа, 'ятати!*** *Більшість операційних систем, в т.ч. Windows, дають змогу перенаправляти потоки введення-виведення, тому функції зчитування і запису даних можна перенаправляти на інші пристрої. Ніколи не намагайтеся безпосередньо відкривати або закривати ці потоки.*

## А.2. Функції консольного введення-виведення даних

Двома найбільш популярними С-функціями введення-виведення даних є **printf**() і **scanf**(). Функція **printf**() записує дані у стандартний пристрій виведення (консоль), а функція **scanf**(), її доповнення, зчитує дані з клавіатури. Оскільки мова С не підтримує перевизначення операторів і не використовує операторів "<<" і ">>" як оператори введення-виведення, то для консольного введення-виведення використовуються саме функції **printf**() і **scanf**(). Обидві вони можуть обробляти дані будь-яких вбудованих типів, в т.ч. символи, рядки і числа. Але оскільки ці функції не є об'єктно-орієнтованими, їх не можна використовувати безпосередньо для введення-виведення об'єктів класів, що створюються програмістом.

### А.2.1. Механізм використання функції printf()

Функція **printf**() має такий прототип:

**int printf(const char** **fmt_string, ...)*;

Перший аргумент fmt_string, визначає спосіб відображення всіх подальших аргументів. Цей аргумент часто називають *рядком форматування*. Вона складається з елементів двох типів: тексту і специфікаторів формату. До елементів першого типу належать символи (текст), які виводяться на екран. Елементи другого типа (специфікатори формату) містять команди форматування, які визначають спосіб відображення аргументів. Команда форматування починається з символу відсотка, за яким знаходиться код формату. Специфікатори формату перераховані в табл. А.1. Кількість аргументів повинна точно збігатися з кількістю команд форматування, причому збіг обов'язковий і у порядку їх проходження. Наприклад, під час виклику наведеної нижче функції **printf**()

**printf**("Привіт %c %d %s", 'c', 10, "всім!");

на екрані буде відображений "Привіт з 10 всім!".

Функція **printf**() повертає кількість реально виведених символів. Негативне значення повернення свідчить про помилку.

**Табл. А.1. Специфікатори формату функція printf()**

| Код | Формат |
|---|---|
| %c | Символ |
| %d | Десяткове ціле із знаком |
| %i | Десяткове ціле із знаком |
| %e | Експоненціальне представлення (рядкова буква e) |
| %E | Експоненціальне представлення (прописна буква E) |
| %f | Значення з плинною крапкою |
| %g | Використовує коротший з двох форматів:<br>%e або %f (якщо %e, використовує рядкову букву e) |
| %G | Використовує коротший з двох форматів:<br>%E АБО %F (якщо %e використовує прописну букву e) |
| %o | Вісімкове ціле без знаку |
| %s | Рядок символів |
| %u | Десяткове ціле без знаку |
| %x | Шістнадцяткове ціле без знаку (рядкові букви) |
| %x | Шістнадцяткове ціле без знаку (прописні букви) |
| %p | Покажчик |
| %n | Відповідний аргумент повинен бути покажчиком на ціле. Даний специфікатор зберігає у цьому цілому число символів, виведених у вихідний потік до поточного моменту (до виявлення специфікатора %n) |
| %% | Виводить символ % |

Команди формату можуть мати модифікатори, які задають ширину поля, точність (кількість десяткових розрядів) і ознака вирівнювання по лівому краю. Ціле значення, розташоване між знаком % і командою форматування, здійснює роль *специфікатора мінімальної ширини поля*. Наявність цього специфікатора приведе до того, що результат буде заповнений пропусками або





нулями, щоб гарантований забезпечити для значення, що виводиться, задану мінімальну довжину. Якщо значення (рядок або число), що виводиться, більше за цей мінімум, його буде виведено повністю, незважаючи на перевищення мінімуму. За замовчуванням як заповнювач використовується пропуск. Для заповнення нулями потрібно помістити 0 перед специфікатором ширини поля. Наприклад, рядок форматування %05d доповнить число, що виводиться, нулями (їх буде менше п'яти), щоб загальна довжина була дорівнювала п'яти символам.

Точне значення *модифікатора точності* залежить від коду програми формату, до якого він застосовується. Щоб додати модифікатор точності, поставте за специфікатором ширини поля десяткову крапку, а після неї значення специфікації точності. Для форматів a, A, e, E, f і F модифікатор точності визначає число десяткових знаків, що виводяться. Наприклад, рядок форматування %10.4f забезпечить виведення числа, ширина якого складе не менше десяти символів, з чотирма десятковими знаками. Стосовно цілих або рядкам, число, наступне за крапкою, задає максимальну довжину поля. Наприклад, рядок форматування %5.7s відобразить рядок завдовжки не менше п'яти, але не більше семи символів. Якщо рядок, що виводиться, виявиться довшим за максимальну довжину поля, кінцеві символи будуть відсічені.

За замовчуванням усі значення, що виводяться, вирівнюються *по правому краю:* якщо ширина поля більше від значення, що виводиться, його буде вирівняно по правому краю поля. Щоб встановити вирівнювання по лівому краю, поставте знак "мінус" відразу після знаку %. Наприклад, рядок форматування %-10.2f забезпечить вирівнювання дійсного числа (з двома десятковими знаками у 10-символьному полі) по лівому краю. Розглянемо програму, у якій продемонстровано механізм використання специфікаторів ширини поля і вирівнювання по лівому краю.

```
#include <stdio.h>              // Для введення/виведення

int main()
{
        printf("|%11.6 f|\n", 123.23);
        printf("|%-11.6 f|\n", 123.23);
        printf("|%11.6 s|\n", "Привіт усім");
        printf("|%-11.6 s|\n", "Привіт усім");
        getch(); return 0;
}
```

Внаслідок виконання ця програма відображає на екрані такі результати:

```
123.230000 123.230000
Привіт Привіт
```

Існують два модифікатори команд форматування, які дають змогу функції **printf**() відображати короткі (**short**) і довгі (**long**) цілі. Ці модифікатори можуть застосовуватися до специфікаторів типу d, i, про, i, x і x. Модифікатор 1 повідомляє функцію **printf**() про довгий формат значення. Наприклад, рядок %ld означає, що повинно бути виведено довге ціле. Модифікатор h вказує на

Додаток короткого формату. Отже, рядок %hu означає, що цілочисельне значення, що виводиться, має тип **short unsigned**.

Щоб позначити, що відповідний аргумент вказує на довге ціле, до специфікатора п можна застосувати модифікатор 1. Для вказівки на коротку цілу застосуєте до специфікатора п модифікатор h.

Якщо Ви використовуєте сучасний компілятор, який підтримує додані у 1995 р. засоби роботи з символами широкого формату (двобайтовими символами), то можете задіювати модифікатор 1 стосовно специфікатора з, щоб повідомити про використання двобайтових символів. Окрім цього, модифікатор 1 можна використовувати з командою формату s для виведення рядка двобайтових символів.

Модифікатор 1 можна також поставити перед командами форматування дійсних чисел e, E, f, F, g і G. У цьому випадку він повідомить про виведення значення типу **long double**.

### А.2.2. Механізм використання функції scanf()

Функція **scanf**()– С-функція загального призначення введення даних з консольного пристрою. Вона може зчитувати дані всіх вбудованих типів і автоматично перетворить числа у відповідний внутрішній формат. Її поведінка багато в чому назад поведінці функції **printf**(). Загальний формат функції **scanf**() такий:

**int scanf**(**const char** **fmt_string, ...*);

Керівний рядок задається параметром *fmt_string*, складається з символів трьох категорій:

- специфікаторів форматування;
- "пропускних" символів (пропуски, символи табуляції та порожнього рядка);
- символів, відмінних від "пропускних".

Функція **scanf**() повертає кількість введених полів, а під час виникнення помилки значення **EOF** (воно визначене у заголовку **stdio.h**).

Специфікатори формату ним передує знак відсотка (%) повідомляють, якого типу дане буде лічене наступним. Наприклад, специфікатор %s прочитає рядок, а %d ціле значення. Ці коди наведено в табл. А.2.

**Табл. А.2. Специфікатори формату функції scanf()**

| Код | Значення |
|---|---|
| %c | Зчитує один символ |
| %d | Зчитує десяткове ціле |
| %i | Зчитує ціле в будь-якому форматі (десяткове, вісімкове, шістнадцяткове) |
| %e | Зчитує дійсне число |
| %f | Зчитує дійсне число |
| %F | Аналогічно коду програми %f (тільки C99) |
| %g | Зчитує дійсне число |
| %o | Зчитує вісімкове число |
| %s | Зчитує рядок |
| %x | Зчитує шістнадцяткове число |





| | |
|---|---|
| %p | Зчитує покажчик |
| %n | Приймає ціле значення, дорівнює кількості символів, зчитаних дотепер |
| %u | Зчитує десяткове ціле без знаку |
| %[ ] | Проглядає набір символів |
| %% | Зчитує знак відсотка |

Пропускні символи в рядку форматування примушують функцію **scanf**() пропустити один або декілька пропускних символів у вхідному потоці. Під пропускним символом маємо на увазі пропуск, символ табуляції або символ нового рядка. По суті, один пропускний символ у керівному рядку примусить функцію **scanf**() зчитувати, але не зберігати будь-яку кількість (нехай навіть нульове) пропускних символів до першого не пропускного.

"Непропускний" символ у рядку форматування примушує функцію **scanf**() прочитати і відкинути відповідний символ. Наприклад, під час використання рядка форматування %d, %d функція **scanf**() спочатку прочитає ціле значення, потім прочитає і відкине кому і нарешті прочитає ще одне ціле. Якщо цей символ не виявиться, то робота функції **scanf**() буде завершена.

Всі змінні, що використовуються для прийому значень за допомогою функції **scanf**(), повинні передаватися за допомогою їх адрес. Це означає, що всі аргументи мають бути покажчиками на змінні. (Мова С не підтримує посилання або посилальні параметри.) Передача покажчиків дає змогу функції **scanf**() змінювати значення будь-якого аргументу. Наприклад, якщо потрібно рахувати цілочисельне значення у змінну *count*, використовується наступний виклик функції:

**scanf**().**scanf**("%d" &*count*);

Рядки звичайно зчитуються у символьні масиви, а ім'я масиву (без індексу) є адресою першого елемента у цьому масиві. Тому, щоб рахувати рядок у символьний масив address, використовують такий програмний код:

```
char address[80];
scanf("%s", address);
```

У цьому випадку параметр address вже є покажчиком, і тому йому не потрібно передувати оператором "&".

Елементи вхідного потоку, що зчитуються функцією **scanf**(), мають бути розділені пропусками, символами табуляції або нового рядка. Такі символи, як кома, крапка з комою і тому подібне, не розпізнаються як роздільники. Це означає, що настанова

**scanf**("%d%d" &r, &c);

прийме значення, введені як 10 20, але навідріз відмовиться від "блюда", поданого у вигляді 10,20.

Подібно до **printf**(), у функції **scanf**() специфікатори формату за порядком зіставляються із змінними, перерахованими у списку аргументів.

Символ "*", що знаходиться після знаку "%" і перед кодом формату, прочитає дані заданого типу, але заборонить їх присвоєння змінній. Отже, настанова

**scanf**("%d%*c%d" &x, &y);

під час введення даних у вигляді 10/20 помістить значення 10 у змінну x, відкине знак ділення і привласнить значення 2 0 змінною у.

Команди форматування можуть містити модифікатор максимальної довжини поля. Він є ціле число, що розташовується між знаком "%" і кодом формату, яке обмежує кількість символів, що зчитуються для будь-якого поля. Наприклад, якщо виникає бажання прочитати у змінну str не більше 20 символів, використаємо таку настанову:

```
scanf("%20 s", str);
```

Якщо вхідний потік містить більше 20 символів, то під час подальшого виконання операції введення зчитування почнеться з того місця, у якому "зупинився" попередній виклик функції **scanf**(). Наприклад, якщо (під час використання цього прикладу) вводиться такий рядок символів:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

то у змінну str будуть прийняті тільки перші 20 символів (до букви 'T'), оскільки команда форматування тут містить модифікатор максимальної довжини поля. Це означає, що решта символів, "UVWXYZ", не буде використана взагалі. У разі іншого виклику функції

```
scanf() scanf("%s", str);
```

символи "UVWXYZ" помістилися б у змінній str. Внаслідок виявлення "пропускного" символу введення даних для поля може завершитися до досягнення максимальної довжини поля. У цьому випадку функція **scanf**() переходить до зчитування наступного поля.

Незважаючи на те, що пропуски, символи табуляції та символ нового рядка використовуються як роздільники полів, під час зчитування одиночного символу вони читаються подібно до будь-якого іншому символу. Наприклад, якщо вхідний потік складається з символів x y, то настанова

```
scanf("%c%c%c" &a, &b, &c);
```

помістить символ x у змінну a, пропуск у змінну b і символ в у змінну c.

Функцію **scanf**() можна також використовувати як *набір сканованих символів* (*scansei*). У цьому випадку визначається набір символів, які можуть бути зчитані функцією **scanf**() і присвоєні відповідному масиву символів. Функція **scanf**() продовжує зчитувати символи і поміщати юс у відповідний символьний масив доти, доки не трапиться символ, відсутній у заданому наборі. Після цього вона переходить до наступного поля (якщо таке є).

Для визначення такого набору необхідно помістити символи, які підлягають скануванню, у квадратні дужки. Відкриваюча квадратна дужка повинна знаходитися відразу за знаком відсотка. Наприклад, наступний набір сканованих символів вказує на те, що необхідно зчитати тільки символи x, y і z:

```
%[XYZ]
```

Відповідна набору змінна повинна бути покажчиком на масив символів. При поверненні з функції **scanf**() цей масив міститиме рядок, що має завершальний нуль-символ, який складається з зчитаних символів. Наприклад, наведений нижче код програми використовує набір сканованих символів для зчитування цифр у масив s1. Якщо буде введений символ, відмінний від цифри, масив s1 завершиться нульовим символом, а решта символів зчитуватиметься в масив s2 доти, доки не буде введений наступний "пропускний" символ.





**Код програми А.1. Демонстрація реалізація простого прикладу використання набору сканованих символів**

```
#include <stdio.h>              // Для введення/виведення

int main()
{
    char s1[80], s2[80];
    printf("Введіть числа, а потім декілька букв:\n");
    scanf("%[0123456789]%s", s1, s2);
    printf("%s %s", s1, s2);
    getch(); return 0;
}
```

Багато компіляторів дають змогу за допомогою дефіса задати в наборі сканованих символів діапазон. Наприклад, у процесі виконання функція **scanf**() прийматиме символи від A до z:

```
%[A-z]
```

До того ж, в наборі сканованих символів можна задати навіть декілька діапазонів. Наприклад, ця програма зчитує спочатку цифри, а потім букви.

**Код програми А.2. Демонстрація механізму використання в наборі сканованих символів декількох діапазонів**

```
#include <stdio.h>              // Для введення/виведення

int main()
{
    char s1[80], s2[80];
    printf("Введіть числа, а потім декілька букв:\n");
    scanf("%[0-9]%[a-zA-z]", s1, s2);
    printf("%s %s", s1, s2);
    getch(); return 0;
}
```

Якщо перший символ у наборі сканованих символів є знаком вставлення (Л), то отримуємо зворотний ефект: дані, що вводяться, зчитуватимуться до першого символу із заданого набору символів, тобто знак вставлення примушує функцію **scanf**() приймати будь-які символи, які *не визначені* в наборі. У наступній модифікації попередньої програми знак вставлення використовують для заборони зчитування символів, тип яких вказано у наборі сканованих символів:

**Код програми А.3. Демонстрація механізму використання набору сканованих символів для заборони зчитування вказаних у ньому символів**

```
#include <stdio.h>              // Для введення/виведення

int main()
{
```

```
    char s1[80], s2[80];
    printf("Введіть не цифри, а потім не букви:\n");
    scanf("%[^0-9]%[^a-zA-z]", s1, s2);
    printf("%s %s", s1, s2);
    getch(); return 0;
}
```

Важливо пам'ятати, що набір сканованих символів розрізняє прописні та рядкові букви. Отже, якщо виникає бажання сканувати як прописні, так і рядкові букви, задайте їх окремо.

Деякі специфікатори формату можуть використовувати такі модифікатори, які точно указують тип змінної, що приймає дані. Щоб прочитати довге ціле, поставте перед специфікатором формату модифікатор 1, а щоб прочитати коротке ціле модифікатор h. Ці модифікатори можна використовувати з кодами формату d, i, o, u, x і n.

За замовчуванням специфікатори f, e і g примушують функцію **scanf**() присвоювати дані змінним типу **float**. Якщо поставити перед одним з цих специфікаторів формату модифікатор 1, функція **scanf**() привласнить прочитане дане змінній типу **double**. Використання ж модифікатора L означає, що змінна, що приймає значення, має тип **long**, **double**.

Модифікатор 1 можна застосувати і до специфікатора c, щоб позначити покажчик на двобайтовий символ з типом даних whcar_t (якщо Ваш компілятор відповідає стандарту C++). Модифікатор 1 можна також використовувати з кодом формату s, щоб позначити покажчик на рядок двобайтових символів. Окрім цього, модифікатор 1 можна використовувати для модифікації набору сканованих двобайтових символів.

## А.3. С-система оброблення файлів

Незважаючи на те, що файлова система в С відрізняється від тієї, що використовується у мові програмування C++, між ними є багато загального. С-система оброблення файлів складається з декількох взаємопов'язаних функцій. Найбільш популярні з них перераховані в табл. А.3.

**Табл. А.3. С-функції оброблення файлів**

| Функція | Призначення |
|---|---|
| **fopen**() | Відкриває потік |
| **fclose**() | Закриває потік |
| **fputc**() | Записує символ у потік |
| **fgetc**() | Зчитує символ з потоку |
| **fwrite**() | Записує блок даних у потік |
| **fread**() | Зчитує блок даних з потоку |
| **fseek**() | Встановлює індикатор позиції файлу на заданий байт у потоці |
| **fprintf**() | Робить для потоку те, що функція **printf**() робить для консолі |
| **fscanf**() | Робить для потоку те, що функція **scanf**() робить для консолі () |
| **feof**() | Повертає значення **true**, якщо досягнуто кінець файлу |
| **ferror**() | Повертає значення **true**, якщо виникла помилка |
| **rewind**() | Встановлює індикатор позиції файлу у початок файлу |
| **remove**() | Видаляє файл |





Загальний потік, який "цементує" С-систему введення-виведення, є *файловий покажчик* (file pointer). Файловий покажчик це покажчик на інформацію про файл, яка містить його ім'я, статус і поточну позицію. По суті, файловий покажчик ідентифікує конкретний дисковий файл і використовується потоком, щоб повідомити всім С-функціям введення-виведення, де вони повинні виконувати операції. Файловий покажчик це змінна-покажчик типу **FILE**, який визначено у заголовку **stdio.h**.

### А.3.1. Механізм використання функції fopen()

Функція **fopen**() здійснює три завдання.

- відкриває потік;
- пов'язує файл з потоком;
- повертає покажчик типу **FILE** на цей потік.

Найчастіше під файлом маємо на увазі дисковий файл. Функція **fopen**() має такий прототип:

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

У цьому записі параметр *filename* вказує на ім'я файлу, що відкривається, а параметр *mode*- на рядок, що містить потрібний статус (режим) відкриття файлу. Можливі значення параметра *mode* показані у наведеній табл. А.4. Параметр *filename* повинен представляти рядок символів, які становлять ім'я файлу, яке допустимо у даній операційній системі. Цей рядок може містити специфікацію шляху, якщо діюче середовище підтримує таку можливість.

**Табл. А.4. Допустимі значення параметра anode**

| mode | Призначення |
|---|---|
| "r" | Відкриває текстовий файл для зчитування |
| "w" | Створює текстовий файл для запису |
| "a" | Відкриває текстовий файл для запису у кінець файлу |
| "rb" | Відкриває двійковий файл для зчитування |
| "wb" | Створює двійковий файл для запису |
| "ab" | Відкриває двійковий файл для запису у кінець файлу |
| "r+" | Відкриває текстовий файл для зчитування і запису |
| "w+" | Створює текстовий файл для зчитування і запису |
| " a+ " | Відкриває текстовий файл для зчитування і запису у кінець файлу |
| "r+b" | Відкриває двійковий файл для зчитування і запису |
| "w+b" | Створює двійковий файл для зчитування і запису |
| "a + b" | Відкриває двійковий файл для зчитування і запису у кінець файлу |

Якщо функція **fopen**() успішно відкрила заданий файл, вона повертає покажчик **FILE**. Цей покажчик ідентифікує файл і використовується більшістю інших файлових системних функцій. Він не повинен піддаватися модифікації кодом програми. Якщо файл не вдається відкрити, повертається нульовий покажчик.

Як видно з табл. А.4, файл можна відкривати або в текстовому, або у двійковому режимі. Під час відкриття в текстовому режимі виконуються перетворення деяких послідовностей символів. Наприклад, символи нового ря-

дка перетворяться у послідовності символів "повернення каретки" / "переведення рядка". У двійковому режимі подібні перетворення не виконуються.

Якщо виникає потреба відкрити файл test для запису, використаємо таку настанову:

```
fp = fopen("test", "w");
```

У цьому записі змінна *fp* має тип **FILE** *. Проте часто для відкриття файлу використовують такий програмний код:

```
if((fp = fopen("test", "w"))==NULL) {
        printf("Не вдається відкрити файл.");
        exit(1);
}
```

При такому методі виявляється будь-яка помилка, пов'язана з відкриттям файлу (наприклад, під час використання захищеного від запису або заповненого диска), і тільки після цього можна робити спробу запису у заданий файл. **NULL** це макроім'я, що є визначеним у заголовку **stdio.h**.

Якщо Ви використовуєте функцію **fopen**(), щоб відкрити файл виключно для виконання операцій виведення (записи), будь-який вже наявний файл із заданим іменем буде стертий, і замість нього буде створено новий. Якщо файл з таким іменем не існує, він буде створено. Якщо виникає потреба додавати дані у кінець файлу, використовується режим "a". Якщо опиниться, що вказано файл не існує, він буде створено. Щоб відкрити файл для виконання операцій зчитування, необхідна наявність цього файлу. Інакше функція поверне значення помилки. Нарешті, якщо файл відкривається для виконання операцій зчитування-запису, то у разі його існування він не буде видалений; але його немає, він буде створено.

### А.3.2. Механізм використання функції fputc()

Функція **fputc**() використовують для виведення символів у потік, заздалегідь відкрито для запису за допомогою функції **fopen**(). Її прототип має такий вигляд:

```
int fputc(int ch, FILE *fp);
```

У цьому записі параметр *fp* файловий покажчик, що повертається функцією **fopen**(), а параметр ch символ, що виводиться. Файловий покажчик повідомляє функції **fputc**(), в який дисковий файл необхідно записати символ. Незважаючи на те, що параметр *ch* має тип **int**, *у ньому* використовується тільки молодший байт.

При успішному виконанні операції виведення функція **fputc**() повертає записаний у файл символ, інакше значення **EOF**.

### А.3.3. Механізм використання функції fgetc()

Функція **fgetc**() використовують для зчитування символів з потоку, відкритого в режимі зчитування за допомогою функції **fopen()**. Її прототип має такий вигляд:





```
int fgetc(FILE *fp);
```

У цьому записі параметр *fp* файловий покажчик, що повертається функцією **fopen**(). Незважаючи на те, що функція **fgetc**() повертає значення типу **int**, його старший байт дорівнює нулю. Під час виникнення помилки або досягненні кінця файлу функція **fgetc**() повертає значення **EOF**. Отже, для того, щоби рахувати весь вміст текстового файлу (до самого кінця), можна використати такий програмний код:

```
ch = fgetc(fp);
while(ch != EOF) {
        ch = fgetc(fp);
}
```

### А.3.4. Механізм використання функції feof()

Файлова система у мові С може також обробляти двійкові дані. Якщо файл відкрито у режимі введення двійкових даних, то не виключено, що може бути зчитано ціле число, дорівнює значенню **EOF**. У цьому випадку під час використання такого коду програми перевірки досягнення кінця файлу, як ch != **EOF**, буде створена ситуація, еквівалентна отриманню сигналу про досягнення кінця файлу, хоча насправді фізичний кінець файлу може бути ще не досягнутий. Щоб вирішити цю проблему, у мові С передбачена функція **feof**(), яка використовують для визначення факту досягнення кінця файлу під час зчитування двійкових даних. Її прототип має такий вигляд:

```
int feof(FILE *fp);
```

У цьому записі параметр *fp* ідентифікує файл. Функція **feof**() повертає ненульове значення, якщо кінець файлу був-таки досягнутий; інакше нуль. Таким чином, у процесі виконання такої настанови буде зчитаний весь вміст двійкового файлу:

```
while(!feof(fp)) ch = fgetc(fp);
```

Безумовно, цей метод застосовний і до текстових файлів.

### А.3.5. Механізм використання функції fclose()

Функція **fclose**() закриває потік, який був відкрито у результаті звернення до функції **fopen**(). Вона записує у файл усі дані, що ще залишилися у дисковому буфері, і закриває файл на рівні операційної системи. Під час виклику функції **fclose**() звільняється блок керування файлом, що пов'язаний з потоком, що робить його доступним для повторного використання. Ймовірно, Вам відомо про існування обмеження операційної системи на кількість файлів, які можна тримати відкритими у будь-який момент часу, тому, перш ніж відкривати наступний файл, рекомендується закрити всі файли, вже непотрібні для роботи.

Функція **fclose**() має такий прототип:

```
int fclose(FILE *fp);
```

У цьому записі параметр *fp* файловий покажчик, що повертається функцією **fopen**(). При успішному виконанні функція **fclose**() повертає нуль; інакше повертається значення **EOF**. Спроба закрити вже закритий файл розцінюється як помилка. Під час видалення носія даних до закриття файлу згенерує помилка, як і у разі недоліку вільного простору на диску.

### А.3.6. Механізм використання функцій fopen(), fgetc(), fputc() і fclose()

Функції **fopen**(), **fgetc**(), **fputc**() і **fclose**() становлять мінімальний набір операцій з файлами. Їх використання продемонстровано у наведеному нижче коді програми, яка здійснює копіювання файлу. Зверніть увагу на те, що файли відкриваються у двійковому режимі і що для перевірки досягнення кінця файлу використовується функція **feof**().

**Код програми А.4. Демонстрація механізму копіювання вмісту одного файлу в іншій**

```
#include <stdio.h>              // Для введення/виведення

int main(int argc, char *argv[])
{
    FILE *in, *out;
    char ch;

    if(argc != 3) {
        printf("Ви забули ввести ім'я файлу" << endl;);
        return 1;
    }

    if((in=fopen(argv[1], "rb")) == NULL) {
        printf("Не вдається відкрити початковий файл" << endl;);
        return 1;
    }

    if((out=fopen(argv[2], "wb")) == NULL) {
        printf("Не вдається відкрити файл-приймач" << endl;);
        return 1;
    }

    // Код копіювання вмісту файлу
    while(!feof(in)) {
        ch = fgetc(in);
        if(!feof(in)) fputc(ch, out);
    }

    fclose(in);
    fclose(out);

    getch(); return 0;
}
```





### А.3.7. Механізм використання функції ferror() і rewind()

Функція **ferror**() використовують для визначення факту виникнення помилки у процесі виконання операції з файлом. Її прототип має такий вигляд:

```
int ferror(FILE *fp);
```

У цьому записі параметр *fp* дійсний файловий покажчик. Функція **ferror**() повертає значення **true**, якщо у процесі виконання останньої файлової операції відбулася помилка; інакше значення **false**. Оскільки виникнення помилки можливе у процесі виконання будь-якої операції з файлом, функцію **ferror**() необхідно викликати відразу після кожної функції оброблення файлів; інакше інформацію про помилку можна просто втратити.

Функція **rewind**() переміщає індикатор позиції файлу у початок файлу, що задається як аргумент. Її прототип має такий вигляд:

```
void rewind(FILE *fp);
```

У цьому записі параметр *fp* дійсний файловий покажчик.

### А.3.8. Механізм використання функції fread() і fwrite()

У файловій системі мови С передбачено дві функції, **fread**() і **fwrite**(), які дають змогу зчитувати і записувати блоки даних. Ці функції подібні С++-функціям **read**() і **write**(). Їх прототипи мають такий вигляд:

```
size_t fread(void *buffer, size_t num_bytes, size_t count, FILE *fp);

size_t fwrite(const void *buffer, size_t num_bytes size_t count, FILE *fp);
```

Під час виклику функції **fread**() параметр *buffer* є покажчик на область пам'яті, яка призначена для прийому даних, які зчитуються з файлу. Функція зчитує *count* об'єктів завдовжки *num_bytes* з потоку, яка адресується файловим покажчиком *fp*. Функція **fread**() повертає кількість зчитаних об'єктів, яка може опинитися менше заданого значення *count*, якщо у процесі виконання цієї операції виникла помилка або був досягнуто кінець файлу.

Під час виклику функції **fwrite**() параметр *buffer* є покажчик на інформацію, яка підлягає запису у файл. Ця функція записує *count* об'єктів завдовжки *num_bytes* у потік, яка адресується файловим покажчиком *fp*. Функція **fwrite**() повертає кількість записаних об'єктів, яка буде дорівнює значенню *count*, якщо у процесі виконання цієї операції не було помилки.

Якщо під час виклику функцій **fread**() і **fwrite**() файл був відкрито для виконання двійкової операції, то вони можуть зчитувати або записувати дані будь-якого типу. Наприклад, наведений нижче код програми записує у дисковий файл значення типу **float**.

**Код програми А.5. Демонстрація механізму запису у дисковий файл значення з плинною крапкою**

```
#include <stdio.h>              // Для введення/виведення

int main()
{
```

```
        FILE *fp;
        float f = 12.23F;

        if((fp=fopen("test","wb"))==NULL) {
                printf("Не вдається відкрити файл" << endl;);
                return 1;
        }

        fwrite(&f, sizeof(float), 1, fp);

        fclose(fp);
        getch(); return 0;
}
```

Як показано у цьому коді програми, значення буфера може виконувати (і при тому достатньо часто) одна змінна.

За допомогою функцій **fread**() і **fwrite**() часто здійснюється зчитування і запис вмісту масивів або структур. Наприклад, наведений нижче код програми, використовуючи одну тільки функцію **fwrite**(), записує вміст масиву значень з плинною крапкою balance у файл з іменем balance. Потім за допомогою однієї тільки функції **fread**() програма зчитує елементи цього масиву і відображає їх на екрані.

**Код програми А.6. Демонстрація механізму використання функції fwrite() для запису вміст масиву значень з плинною крапкою у файл**

```
#include <stdio.h>              // Для введення/виведення

int main()
{
        register int i;
        FILE *fp;
        float balance[100];

        // Відкриваємо файл для запису.
        if((fp=fopen("balance","w"))==NULL) {
                printf("Не вдається відкрити файл" << endl;);
                return 1;
        }

        for(int i=0; i<100; i++) balance[i] = (float) i;

        // Одним махом" зберігаємо весь масив balance.
        fwrite(balance, sizeof balance, 1, fp);
        fclose(fp);

        // Онулюємо масив.
        for(int i=0; i<100; i++) balance[i] = 0.0;

        // Відкриваємо файл для зчитування.
        if((fp=fopen("balance","r"))==NULL) {
                printf("Не вдається відкрити файл" << endl;);
```





```
                return 1;
        }
        // Одним "махом" зчитуємо весь масив balance.
        fread(balance, sizeof balance, 1, fp);

        // Відображаємо вміст масиву.
        for(int i=0; i<100; i++) printf("%f ", balance[i]);

        fclose(fp);

        getch(); return 0;
}
```

Використовувати функції **fread**() і **fwrite**() для зчитування і запису блоків даних ефективніше, ніж багато разів викликати функції **fgetc**() і **fputc**().

## А.4. Виконання операцій введення-виведення даних з довільним доступом до файлів

### А.4.1. Механізм використання функції fseek()

С-система введення-виведення дає змогу виконувати операції зчитування і запису даних з довільним доступом. Для цього слугує функція **fseek**(), яка встановлює потрібним чином індикатор позиції файлу. Її прототип такий.

**int fseek**(**FILE** *fp, **long** *numbytes*, **int** *origin*);

У цьому записі параметр *fp* файловий покажчик, що повертається функцією **fopen**(), параметр *numbytes* кількість байтів щодо початкового положення, які задаються параметром *origin*. Параметр *origin* може приймати одне наступних макроімен (визначених у заголовку **stdio.h**).

| Ім'я | Призначення |
|---|---|
| **SEEK_SET** | Пошук з початку файлу |
| **SEEK_CUR** | Пошук з поточної позиції |
| **SEEK_END** | Пошук з кінця файлу |

Отже, щоб перемістити індикатор позиції у файлі на *numbytes* байтів відносно його початку, як параметр *origin* необхідно використовувати значення **SEEK_SET**. Для переміщення щодо поточної позиції використовуйте значення **SEEK_CUR**, а для зсуву з кінця файлу значення **SEEK_END**.

Нульове значення результату функції свідчить про успішне виконання функції **fseek**(), а ненульове – про виникнення збою. Як правило, функцію **fseek**() не рекомендується використовувати для файлів, відкритих у текстовому режимі, оскільки перетворення символів може призвести до помилкових переміщень індикатора позиції у файлі. Тому краще використовувати цю функцію для файлів, відкритих у двійковому режимі. Якщо виникає потреба обчислення 234-й байт у файлі test, виконайте такий програмний код:

**Код програми А.7. Демонстрація механізму використання функції fseek()**

```
int Fn1()
{
        FILE *fp;
        if((fp=fopen("test", "rb")) == NULL) {
                printf("Не вдається відкрити файл" << endl;);
                exit(1);
        }

        fseek(fp, 234L SEEK_SET);

        return getc(fp); // Зчитування одного символу розташованого на 234-й позиції.
}
```

### А.4.2. Механізм використання функції fprintf() і fscanf()

Крім розглянутих вище основних функцій введення-виведення, С-система введення-виведення містить функції **fprintf**() і **fscanf**(). Поведінка цих функцій аналогічно поведінці функцій **printf**() і **scanf**(), за винятком того, що вони працюють з файлами. Саме тому ці функції звичайно використовуються у С-програмах. Прототипи функцій **fprintf**() і **fscanf**() виглядають так:

```
int fprintf(FILE *fp, const char *fmt_string, ...);
int fscanf(FILE *fp, const char *fmt_string, ...);
```

У цьому записі параметр *fp* файловий покажчик, що повертається функцією **fopen**(). Функції **fprintf**() і **fscanf**() працюють подібно до функцій **printf**() і **scanf**() відповідно, за винятком того, що їх дія спрямована у файл, визначено параметром *fp*.

### А.4.3. Механізм видалення файлів

Функція **remove**() видаляє заданий файл. Її прототип виглядає так.

```
int remove(const char * filename);
```

Вона повертає нуль при успішному видаленні файлу і ненульове значення – в іншому випадку.





# ЛІТЕРАТУРА


**1. Александреску А.** Современное проектирование на C++ / А. Александреску. – Сер.: C++ In-Depth. 3. – М. : Изд. дом "Вильямс", 2002. – 336 с.

**2. Аммераалъ Л.** STL для программистов на C++ / Л. Аммераалъ. – М. : Изд-во ДМК, 1999. – 240 с.

**3. Архангельский А.Я.** Программирование в C++ Builder 6 / А.Я. Архангельский. – М. : Изд-во "Бином", 2004. – 1152 с.

**4. Бронштейн И.Н.** Справочник по математике для инженеров и учащихся втузов / И.Н. Бронштейн, К.А. Семендяев. – 13-е изд., испр. – М. : Изд-во "Наука", Гл. ред. физ.-мат. лит., 1986. – 544 с.

**5. Буч Г.** Объектно-ориентированный анализ и проектирование с примерами на C++ / Г. Буч. – М. : Изд-во "Бином", 1998. – 560 с.

**6. Влиссидес Дж.** Применение шаблонов проектирования. Дополнительные штрихи / Дж. Влиссидес. – М. : Изд. дом "Вильямс", 2003. – 144 с.

**7. Гамма Э.** Приемы объектно-ориентированного программирования. Патерны проектирования / Э. Гамма, Р. Хелм, Р. Джонсон, Дж. Влиссидес. – СПб. : Изд-во "Питер", 2004. – 366 с.

**8. Глинський Я.М.** C++ і C++ Builder / Я.М. Глинський, В.Є. Анохін, В.А. Ряжська. – Львів : Деол, СПД Глинський, 2003. – 192 с.

**9. Кениг Э.** Эффективное программирование на C++. Серия C++ In-Depth. tType. 2 / Э. Кениг, Б. Му. – М. : Изд. дом "Вильямс", 2002. – 384 с.

**10. Ласло М.** Вычислительная геометрия и компьютерная графика на C++ / М. Ласло. – М. : Изд-во "Бином", 1997. – 304 с.

**11. Лафоре, Роберт.** Объектно-ориентированное программирование в C++. Классика Computer Science / Роберт Лафоре. – 4-е изд. : пер. с англ. – СПб. : Изд-во "Питер", 2005. – 924 с.

**12. Либерти Д.** Освой самостоятельно C++ за 21 день / Д. Либерти. – М. : Изд. дом "Вильямс", 2000. – 816 с.

**13. Липпман С.** Язык программирования C++. Вводный курс / С. Липпман, Ж. Лажойе. – 3-е изд. : пер. с англ. – СПб.-М. : Изд-во "Невский диалект – ДМК Пресс", 2004. – 1104 с.

**14. Павловская Т.А.** Программирование на языке высокого уровня : учебник / Т.А. Павловская. – СПб. : Изд-во "Питер", 2005. – 461 с.

**15. Павловская Т.А.** C++. Объектно-ориентированное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 265 с.

**16. Павловская Т.** C/C++. Программирование на языке высокого уровня / Т.А. Павловская. – СПб. : Изд-во "Питер", 2001. – 460 с.

**17. Павловская Т.** C/C++. Структурное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 240 с.

**18. Павловская Т.А.** С++. Объектно-ориентированное программирование : практикум / Т.А. Павловская, Ю.А. Щупак. – СПб. : Изд-во "Питер", 2005. – 265 с.

**19. Прата, Стивен.** Язык программирования С++. Лекции и упражнения : учебник : пер. с англ. / Стивен Прата. – СПб. : ООО "ДиаСофтБП", 2005. – 1104 с.

**20. Саттер Г.** Решение сложных задач на С++. – Сер.: С++ In-Depth / Г. Саттер. – Т. 4. – М. : Изд. дом "Вильямс", 2002. – 400 с.

**21. Синтес, Антони.** Освой самостоятельно объектно-ориентированное программирование за 21 день : пер. с англ. / Антони Синтес. – М. : Изд. дом "Вильямс", 2002. – 672 с.

**22. Справочник** по элементарной математике: геометрия, тригонометрия, векторная алгебра / под ред. члена-корр. АН УССР П.Ф. Фильчакова. – К. : Вид-во "Наук. думка", 1966. – 444 с.

**23. Страуструп Б.** Язык программирования С++ / Б. Страуструп. – СПб. : Изд-во "Бином", 1999. – 991 с.

**24. Халперн, Пабло.** Стандартная библиотека С++ на примерах : пер. с англ. / Пабло Халперн. – М. : Изд. дом "Вильямс", 2001. – 336 с.

**25. Шаллоуэй А.** Шаблоны проектирования. Новый подход к объектно-ориентированному анализу и проектированию / А. Шаллоуэй, Д. Тротт. – М. : Изд. дом "Вильямс", 2002. – 288 с.

**26. Шилдт, Герберт.** Искусство программирования на С++ : пер. с англ. / Герберт Шилдт. – СПб. : Изд-во БХВ-Петербург, 2005. – 496 с.

**27. Шилдт, Герберт.** Полный справочник по С++ / Герберт Шилдт. – 4-е изд. : пер. с англ. – М. : Изд. дом "Вильямс", 2010. – 800 с.

**28. Шилдт, Герберт.** С++: Базовый курс / Герберт Шилдт. – 3-е изд. : пер. с англ. – М. : Изд. дом "Вильямс", 2005. – 624 с.

**29. Шилдт, Герберт.** Самоучитель С++ / Герберт Шилдт. – 3-е изд. : пер. с англ. – СПб. : Изд-во БХВ-Петербург, 2005. – 688 с.

**30. Штерн В.** Основы С++. Методы программной инженерии / В. Штерн. – М. : Изд-во "Лори", 2003. – 860 с.

**31. Элджер Д.** С++ библиотека программиста / Д. Элджер. – СПб. : Изд-во "Питер", 2000. – 320 с.





## ДЛЯ НОТАТОК

Навчальне видання

**ГРИЦЮК Юрій Іванович,**
**РАК Тарас Євгенович**

# ПРОГРАМУВАННЯ МОВОЮ С++

*Навчальний посібник*

Літературний редактор *В.В. Дудок*

Редактор *Г.М. Падик*
Технічний редактор *О.В. Хлевной*
Авторські комп'ютерний набір та верстка

Підписано до друку 25.05.2011. Формат 60×84/16.
Папір офсетний. Гарнітура *Times*. Друк на різографі.
Ум. др. арк. 16,97. Ум. фарбо-відб. 17,21.
Наклад 350 прим. Зам. № *28/2011*

Видавництво ЛДУ БЖД, Україна, 79007, м. Львів, вул. Клепарівська, 35
Тел./факс: (032) 233-14-77; E-mail: mail@ubgd.lviv.ua; Wed-адреса: http://www.ubgd.lviv.ua
Свідоцтво про внесення суб'єкта видавничої справи до державного реєстру видавців, виготовників і розповсюджувачів видавничої продукції, серія ДК, № 368 від 20.03.2001 р.

**Грицюк Ю.І., Рак Т.Є.**

Г 85 Програмування мовою С++ : навчальний посібник. – Львів : Вид-во Львівського ДУ БЖД, 2011. – 292 с. – Статистика: іл. 10, табл. 18, бібліогр. 31.

**ISBN 978-966-3466-85-9**

У навчальному посібнику розглянуто різні аспекти програмування мовою С++. У початкових розділах ґрунтовно описано синтаксис та семантику основних стандартних конструктивних компонентів мови: лексем, виразів, операторів, функцій. Значну увагу приділено різновидам типів даних, зокрема опрацюванню масивів, символьних рядків, структур тощо. Далі у посібнику даються особливості застосування покажчиків і С++-функцій, механізм використання засобів програмування для розширення їх можливостей, наведено механізм реалізації С++-специфікаторів і спеціальних операторів, даються основні поняття про структури і об'єднання даних.

Викладений матеріал базується на стандарті ANSI/ISO мови програмування С++, а також зазначено нововведення, які затверджені в стандарті ISO/IEC 14882:2003. Наведено важливу для практичного використання та програмування інформацію про додаткові можливості компілятора, середовища та бібліотек Borland C/C++.

Видання призначено для курсантів і студентів, які вивчають програмування в рамках різних навчальних дисциплін, а також для всіх, хто бажає самостійно опанувати технологію програмування мовою С++.