Лекція 1
Вступ

План

1. Програми, дані, моделі, мови
2. Алгоритми та їхні властивості
3. Мови програмування
4. Види діяльності зі створення програми
5. Кроки роботи з програмою

1. Програми, дані, моделі, мови

Сучасна людина використовує комп'ютер для розв'язання різноманітних задач – від виконання простих арифметичних дій до моделювання атмосферних явищ і керування польотами в космос. Насправді все, що "вміє" комп'ютер – це *виконувати програми*. Щоб комп'ютер розв'язав потрібну задачу, необхідно спочатку створити відповідну програму, а потім запустити її на виконання.

Програма (*program*) – це опис тих дій, які має виконати комп'ютер, щоб розв'язати деяку задачу. **Програмування** – це діяльність, яка полягає у створенні програм. Мета програмування – забезпечити, щоб замість людини ту чи іншу роботу виконував комп'ютер.

Усі дії комп'ютера пов'язані з **обробкою даних** – числових, символьних, текстових тощо. Згідно з В. М. Глушковим, **дані** (*data*) зображують (позначають) деякий **зміст**, або **інформацію**. Поняття змісту та інформації не мають чіткого означення або тлумачення. Будемо розуміти їх як знання, відомості про щось.

Отже, зображення об'єктів реального світу за допомогою даних є основою будь-якої взаємодії, у тому числі й комп'ютера із зовнішнім світом.

Комп'ютер має розв'язувати задачі для людини. У цих задачах фігурують різноманітні об'єкти реального світу – картинки, фізичні явища, особи, технологічні процеси тощо. Щоб запрограмувати обробку даних, пов'язаних із цими об'єктами, треба спочатку *зобразити об'єкти у вигляді даних*.

Зображення об'єкта, явища або процесу за допомогою даних про нього є його *моделлю*. Узагалі, **модель** – це спрощене зображення об'єкта або явища, що відображає його необхідні суттєві властивості. В обробці даних модель є сукупністю властивостей і співвідношень між ними, що відображають суттєві риси досліджуваного об'єкта, явища або процесу. Модель зазвичай містить опис не лише даних, а також їх обробки, яка відтворює реальні процеси, пов'язані з об'єктом.

Існує безліч різновидів моделей. Один і той самий об'єкт реального світу може мати кілька різних моделей, що виражають властивості, потрібні з різних поглядів на нього.

Власне дані – це деякі позначення, тобто записи, що позначають властивості об'єкта. Однак для правильного розуміння запису (відновлення позначеного ним змісту) необхідно знати, що саме позначають його окремі елементи.

Система позначень деякого змісту називається **мовою**. Мова включає елементарні позначення, правила утворення складніших позначень із простіших і правила, за якими зміст і позначення відповідають одне одному. Правила

утворення позначень визначають **синтаксис** мови, а правила, що задають зміст позначень, – **семантику**.

Для спілкування й мислення людина використовує *природні мови* (українську, російську, англійську тощо). Проте в науці й техніці природні мови неефективні або недостатні, тому розроблено також спеціальні *штучні мови*. Їх застосовують насамперед для обміну інформацією між користувачем і/або прикладними процесами. Одним із класів штучних мов є *мови програмування*, призначені для запису програм.

2. Алгоритми та їхні властивості

Програму, призначену для виконання комп'ютером, можна розглядати як різновид алгоритму, що є загальнішим поняттям.

Алгоритм – це опис послідовності дій, які треба виконати, щоб розв'язати деяку задачу. Позначення дій у алгоритмі називаються **командами** або **інструкціями** (*statement*).

Зазвичай у алгоритмі вказано деякі **вхідні**, **результатні** (**вихідні**) та **проміжні дані**, що не є ні вхідними, ні вихідними.

Послідовність дій, що виконується за алгоритмом, називається **процесом**. Алгоритм зазвичай визначає не один, а деяку множину процесів. Алгоритми мають кілька загальних властивостей: зрозумілість, результативність, однозначність, дискретність, масовість і виконуваність. Розглянемо їх.

Зрозумілість. Для виконання алгоритму завжди потрібен *виконавець*. Це може бути людина або деяка технічна система, зокрема комп'ютер. Наприклад, виконувати арифметичні дії, розв'язуючи квадратне рівняння (і не тільки), може людина. Однак вона може перекласти цю роботу на комп'ютер, якщо створить відповідну програму та примусить комп'ютер її виконати.

Так само збирати прилади може спеціальна автоматична лінія, якщо виконує відповідну програму.

Зрозумілість алгоритму полягає в тому, що виконавець може правильно зрозуміти й виконати команди, записані в алгоритмі. Команди завжди записуються за допомогою певної системи позначень, тобто мови. Отже, виконавець повинен розуміти мову запису алгоритму.

Результативність. Виконання будь-якого алгоритму має приносити його виконавцю або іншій особі відчутні результати. Наприклад, "корені рівняння визначено", "прилад зібрано" тощо.

Однозначність. В алгоритмі не допускаються команди, зміст яких можна сприйняти неоднозначно. Наприклад, якби в алгоритмі розв'язання квадратного рівняння була команда "обчислити x або написати, що коренів немає", то виконавець не знав би, що саме йому робити. Окрім того, після виконання кожної команди виконавець повинен точно знати, що робити далі.

Дискретність. Дискретність алгоритму полягає в тому, що він задає послідовність дій, чітко відокремлених одна від одної. Отже, дії, задані командою, мають починатися лише після закінчення дій за попередньою командою. Окрім того, виконання кожної команди повинне займати обмежений проміжок часу.

Масовість. Конкретні об'єкти, до яких застосовуються дії під час виконання алгоритму, визначають конкретні задачі, що часто називаються **екземплярами**

задачі. Наприклад, конкретна трійка чисел 3, 10, 2 відповідає квадратному рівнянню, яке треба розв'язати. Масовість алгоритму полягає в тому, що він застосовний до різних наборів вхідних даних, тобто до різних екземплярів задач. Найчастіше алгоритм описує не один, а деяку *множину процесів*, які відбуваються при розв'язанні всіх можливих екземплярів задачі, хоча існують і алгоритми, що задають тільки один процес.

Виконуваність і скінченність. Алгоритм має бути таким, щоб на кожному екземплярі задачі його можна було *виконати до кінця* (й отримати результат). Кожен процес, заданий алгоритмом, має бути *скінченним* і тривати скінченний час. Окрім того, процес *не повинен обриватися* без отримання результату.

Комп'ютерна програма є послідовністю команд, основний зміст яких – *обробка даних*. Центральний процесор зчитує дані й команди програми з оперативної пам'яті та виконує їх. Команди задають зчитування даних із пам'яті, створення нових даних і запис їх у пам'ять. Є також команди, за якими дані надходять до зовнішніх пристроїв або зчитуються з них.

3. Мови програмування

Комп’ютери розуміють тільки дуже обмежений набір інструкцій і щоб змусити їх щось зробити, потрібно чітко сформулювати завдання, використовуючи ці ж інструкції. **Програма** (також «**додаток**», «**програмне забезпечення**», «**софт**») — це набір інструкцій, які вказують комп’ютеру, що йому потрібно зробити. Фізична частина комп’ютера, яка виконує ці інструкції, називається «**залізом**» або **апаратною частиною** (наприклад: процесор, материнська плата, оперативна пам’ять, тощо).

Машинна мова

Процесор комп’ютера не здатен розуміти мови програмування (такі як C++, Java, Python і т.д) напряму. Дуже обмежений набір інструкцій, які розуміє процесор, називається **машинним кодом** (або ще «**машинною мовою**»). Варто відзначити наступні дві речі:

По-перше, кожна команда (інструкція) складається тільки з двох цифр: 0 або 1. Ці числа називаються **бітами** (скорочено від англ. «**bi**nary digi**t**») або **двійковим кодом**.

Наприклад, нижче представлена команда машинної мови архітектури x86:

```
10110000 01100001
```

По-друге, кожен набір бітів трансформується процесором в інструкції для виконання певного завдання (наприклад, порівняти два числа між собою або перемістити вказане число в певну комірку пам’яті). Різні типи процесорів зазвичай мають різні набори інструкцій, тому інструкції, які працюватимуть на процесорах Intel цілком ймовірно, що не працюватимуть на процесорах Xenon (які використовуються в ігрових консолях Xbox). Раніше, коли комп’ютери тільки починали масово поширюватися, програмісти писали більшість програм тільки на машинній мові, що було дуже незручно, важко і займало набагато більше часу, ніж зараз.

Мова Асемблера

Так як програмувати на машинній мові є специфічним задоволенням, то програмісти винайшли **Мову Асемблера**. У цій мові кожна команда ідентифікується коротким ім'ям (а не набором одиниць з нулями), і змінними можна управляти через їх імена. Таким чином, писати/читати код стало набагато легше. Проте процесор все рівно не здатен розуміти мову асемблера напряму. Цю мову також потрібно перекладати в машинний код. **Асемблер** — це транслятор (перекладач), який перекладає код, написаний на мові асемблера, в машинну мову.

Перевагою Асемблера є його продуктивність (а саме швидкість виконання) і він досі використовується, коли швидкість виконання має вирішальне значення. Причина даної переваги заключається в тому, що код, написаний на мові асемблера, адаптується до кожного конкретного процесора окремо. Програми, адаптовані під один процесор, не будуть працювати з іншим. Крім того, для того, щоб програмувати на Асемблері, потрібно також знати дуже багато не дуже читабельних інструкцій для виконання навіть простих завдань.

Наприклад, ось команда, яка записана вище, але уже на мові асемблера:

```
mov al, 061h
```

Високорівнені мови програмування

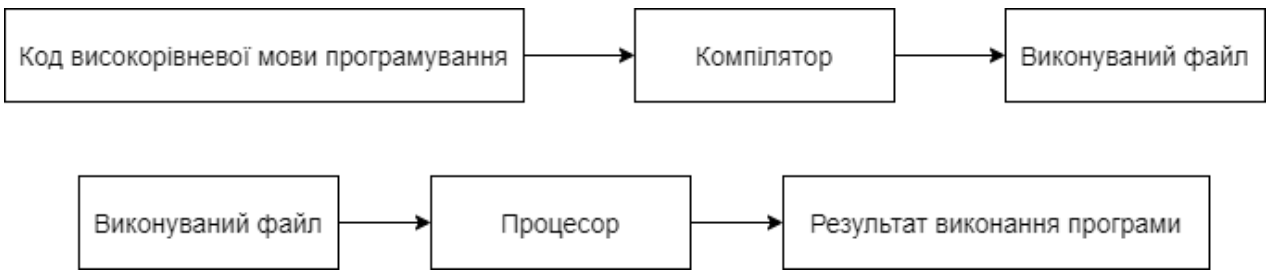
Для вирішення проблем читабельності коду і надмірної складності були розроблені високорівневі мови програмування. C, C++, Pascal, Java, JavaScript і Perl відносяться до високорівневих мов програмування, однією з основних властивостей яких є портативність під різні архітектури процесорів. Програми, написані на мовах високого рівня, також повинні бути перекладені в машинну мову перед їх виконанням процесором. Є два варіанта перекладу в машинну мову:

компіляція, яка виконується компілятором;

інтерпретація, яка виконується інтерпретатором.

Компілятор — це програма, яка зчитує код і створює автономну (здатну працювати незалежно від іншого апаратного або програмного забезпечення) виконувану програму, яку процесор здатен зрозуміти. При запуску програми весь її код повністю компілюється, а потім створюється виконуваний файл і вже при повторному запуску цієї програми компіляція (вже другий раз) не виконується.

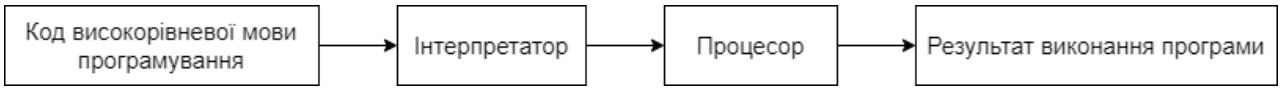
Процес компіляції можна зобразити наступним чином:



Інтерпретатор — це програма, яка виконує код напряму, без його попередньої компіляції в виконуваний файл. Інтерпретатори більш гнучкі, але

менш ефективні, так як процес інтерпретації виконується повторно при кожному новому запуску програми.

Процес інтерпретації можна зобразити наступним чином:



Будь-яка мова програмування може або компілюватися, або інтерпретуватися. Однак, такі мови, як C, C++ і Pascal — компілюються, в той час як “скриптові” мови програмування, такі, як Perl і JavaScript — інтерпретуються. Деякі мови програмування (наприклад, Java) можуть як компілюватися, так і інтерпретуватися.

Переваги високорівневих мов програмування

Перевага №1: Легше писати/читати код. Ось та ж команда, що вище, але вже на мові C++:

```
a = 97;
```

Перевага №2: Потрібно менше інструкцій для виконання завдань. В C++ ви можете зробити щось на зразок наступного в одному рядку:

```
a = b * 2 + 5;
```

В мові асемблера вам довелося б використати 5 або 6 інструкцій.

Перевага №3: Вам не потрібно турбуватися про такі деталі, як завантаження змінних в регістри процесора. Компілятор або інтерпретатор бере це на себе.

Перевага №4: Високорівневі мови програмування є портативними, тобто підходять під різні архітектури (але є один нюанс):

Нюанс полягає в тому, що більшість платформ, такі як Microsoft Windows, мають свої власні специфічні функції, за допомогою яких писати код набагато легше. Але ціна цьому — портативність, так як функції, специфічні для однієї платформи, цілком ймовірно, що не працюватимуть на іншій платформі.

4. Види діяльності зі створення програми

Процес створення програми в найзагальніших рисах вимагає кількох видів

діяльності:

- аналіз задачі й уточнення її постановки;
- проектування програми;
- розробка програми (кодування);
- перевірка програми (тестування);
- передача замовнику (упровадження).

Аналіз задачі й уточнення її постановки. Спочатку замовник формулює задачу, яку йому необхідно розв'язати. Задачу зазвичай сформульовано недостатньо точно, навіть може бути, що замовник чітко не уявляє, що саме йому насправді потрібно. Саме тому робота над програмою починається з аналізу задачі й уточнення її постановки. Для цього потрібно заглибитися в предметну область замовника й у діалозі з ним уточнити деякі питання. Зазвичай за результатами аналізу задачі будується спрощена модель предметної області, у термінах якої уточнюється задача. Необхідно також з'ясувати вхідні дані майбутньої програми й результат її роботи над ними. Якщо розв'язання поставленої задачі можливе не за всіх вхідних даних, то слід *визначити поведінку програми на некоректних вхідних даних.*

Приклад. Розглянемо задачу: написати програму ділення двох чисел. Вхідними даними є пара чисел: перше - ділене, друге - дільник. Проте дані з дільником 0 є некоректними. Отже, уточнення постановки полягає в тому, що для коректних вхідних даних програма має виводити частку від ділення першого числа на друге, а для некоректних - повідомлення про неможливість ділення.

В умовах навчального процесу замовником виступає викладач. Проте умова задачі все рівно не обов'язково формулюється цілком точно й однозначно (див. попередній приклад).

Якщо не провести аналіз задачі й на його підставі не уточнити її постановку, то можна розв'язати не ту задачу. Аналіз задачі дозволяє визначити, якими саме засобами (математичними та програмними) розв'язувати задачу, а уточнена постановка - що саме й у яких ситуаціях має робити програма.

Проектування програми. Між написанням твору та програми є певна аналогія. Писати твір починають із плану, який далі розкривають у творі. Так само з програмою: спочатку формулюють її план у вигляді проекту, а потім втілюють цей план у життя - пишуть код (текст) програми.

Під час проектування з'ясовують структуру програми, її складові частини та взаємодію між ними. Тут поступово *уточнюють дії* з розв'язання задачі *та їх опис.* З'ясовують і *уточнюють дані,* потрібні для розв'язання задачі. Дуже часто в задачі можна виділити кілька *підзадач* і описати їх розв'язання окремо. Тоді й алгоритм складається зі зв'язаних і узгоджених між собою частин *(допоміжних алгоритмів),* що описують розв'язання підзадач.

Одночасно з проектуванням зазвичай відбувається подальше уточнення постановки задачі й моделі предметної області. На практиці замовник може вирішити дещо змінити умову задачі (причому будь-коли!), і тоді доводиться знов уточнювати постановку та аналізувати задачу.

Результатом проектування є модель (проект) програми, що далі поступово перетворюється на текст програми. Ця модель може бути записана, наприклад, у вигляді алгоритму з достатньо абстрактними кроками.

У багатьох програм є чотири основні частини: отримати вхідні дані,

обробити некоректні вхідні дані, обробити коректні вхідні дані, вивести результати обробки. Для складніших задач під час проектування потрібно визначати не лише загальний алгоритм роботи програми, але й необхідні структури даних і, можливо, програмні засоби для створення програми та її окремих частин.

Розробка програми (кодування). Коли дії та дані уточнено до вигляду, в якому їх можна виразити мовою програмування, починають розробку програми. Найчастіше програму записують *мовою високого рівня* (інколи окремі її частини - різними мовами).

Розробляючи програму, можна припуститися помилок, що виявляються під час трансляції або виконання програми. Помилки необхідно виявляти й виправляти, тобто налагоджувати програму. **Налагодження програми** полягає в тому, що її багаторазово запускають зі *спеціально підібраними* вхідними даними, які допомагають виявити й відшукати помилки, а потім виправити їх.

Проект програми може залежати від можливостей мови програмування та обладнання. У довготривалих проектах трапляється, що під час розробки змінюються програмні й апаратні засоби, замовник уточнює задачу відповідно до нових можливостей свого обладнання, тому всі процеси починаються наново.

Перевірка програми (тестування). У реальних виробничих процесах програму розробляють програмісти, а перевіряють інші спеціалісти – **тестувальники**. Тестування починається, коли програмісти впевнені, що програма (або деяка її частина) є правильною. Задачею тестування є лише встановлення факту відсутності або наявності помилок. Зазвичай спочатку тестувальники виявляють помилки, і цикл "розробка – тестування" повторюється. В умовах навчання ситуація аналогічна, тільки в ролі програміста виступає студент, а тестувальника – спочатку студент, а потім викладач.

Передача замовнику (упровадження). У найпростішому випадку робота програміста над програмою завершується передачею програми й супроводжувальної документації з її використання замовникові. Для серйозніших програм може знадобитися не просто передати код замовнику, але й установити програму в замовника, зокрема у випадках, коли програма потребує спеціального налаштування параметрів. Інколи потрібно навчити персонал замовника працювати з програмою. Усі ці дії (передавання, установлення, навчання), що дозволяють замовнику користуватися програмою у своїх виробничих процесах, є частиною впровадження програми.

Серйозні програми потребують **супроводження**. Розробник виправляє помилки, виявлені під час експлуатації програми, модернізує її, передає оновлені варіанти користувачам тощо.

У реальних великих проектах одночасно можуть виконуватися кілька видів робіт. Як тільки постановку задачі більш-менш зрозуміло, можна проектувати програму, обирати програмні засоби для реалізації, писати частини коду й тестувати їх, демонструвати готові частини замовнику, отримувати від нього уточнення й навчати його користуватися програмою. Зазвичай програма та її можливості нарощуються поступово, шляхом багаторазових повернень до аналізу задачі та інших видів роботи.

5. Кроки роботи з програмою

Типова послідовність роботи з програмою включає такі кроки: набирання

тексту, компіляція, компонування, завантаження й виконання або інтерпретація.

Набирання тексту. Текст програми мовою високого рівня (**вхідний текст**) найчастіше набирають за допомогою спеціальної програми (текстового редактора) і зазвичай записують на диск у вигляді вхідного файлу (рис. 1.2). Програма може складатися з кількох файлів – у великих програмах їх можуть бути десятки й сотні.

Компіляція. Компілятор – це програма, під час виконання якої читається вхідний текст і створюється його машинний еквівалент – **об'єктний код** (рис.1.2).

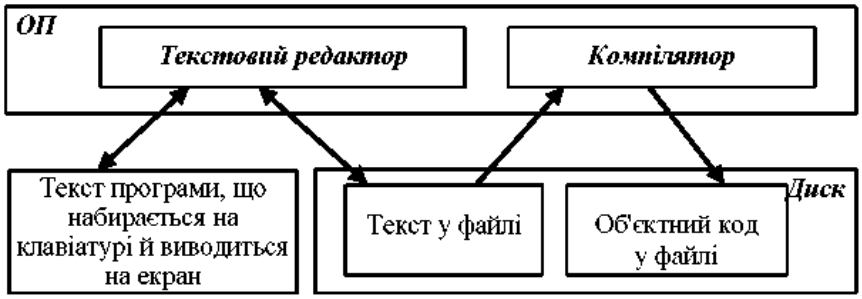


Рис. 1.2. Створення й компіляція тексту

Зазвичай об'єктний код програми містить далеко не всі необхідні команди– програма може складатися з частин; деякі з них є стандартними.

Компонування. Об'єктний код обробляє ще одна програма – **компонувальник**. Ця програма "збирає з частин" (компонує) виконуваний код, тобто машинну програму, і записує його або в оперативну пам'ять (**завантажує**), або на диск у вигляді файлу, *готового до виконання* (рис. 1.3). Такий файл можна завантажити пізніше.

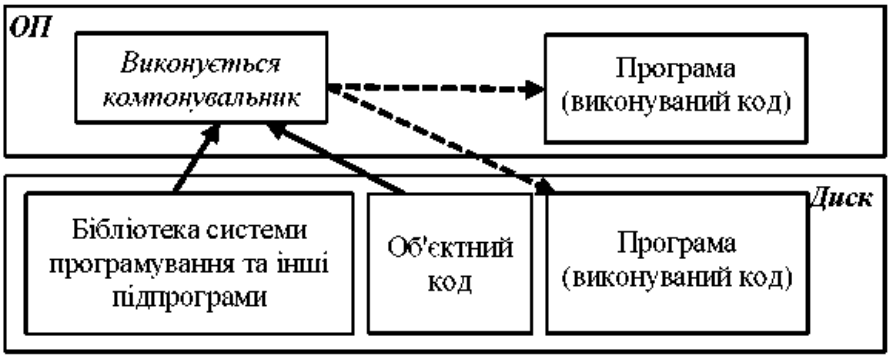


Рис. 1.3. Компонування й завантаження програми

Завантаження й виконання. Запис машинної програми в оперативну пам'ять називається **завантаженням** (рис. 1.4). Його здійснює спеціальна програма – **завантажувач**, який може входити до складу компонувальника. Якщо завантаження здійснене успішно, то починається процес виконання завантаженої програми.

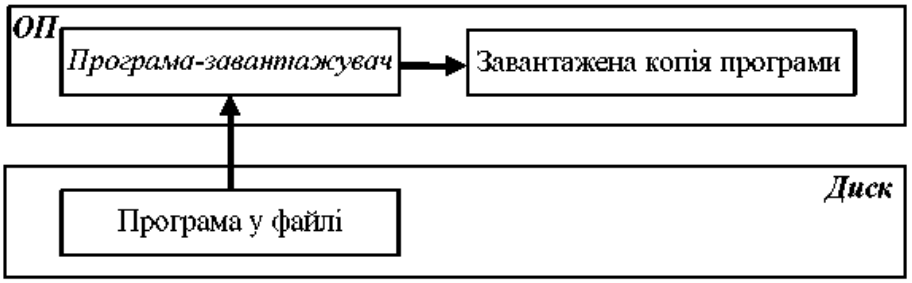


Рис. 1.4. Завантаження програми

Інтерпретація. Це такий спосіб обробки високорівневої програми, за яким машинна програма не створюється (рис. 1.5). Вхідна високорівнева програма обробляється спеціальною програмою – **інтерпретатором**. При цьому дії вхідної програми, які вона задає, відразу виконуються. Зазвичай інтерпретація вхідної програми відбувається повільніше, ніж виконання відповідної машинної програми.

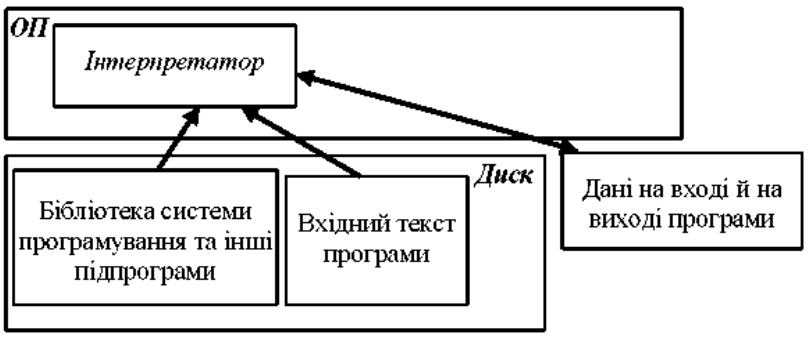


Рис. 1.5. Інтерпретація високорівневої програми

Інтерпретація програми використовується в такому інструменті, як **налагоджувач**. Він забезпечує інтерпретацію вхідної програми невеликими порціями (кроками) і дає можливість побачити результати виконання після кожного кроку. Це полегшує виявлення помилок у вхідній програмі.

Інколи компіляцію та інтерпретацію узагальнюють словом **трансляція**, називаючи трансляторами всі види програм перетворення вхідних текстів до машинного чи проміжного вигляду.

Описані засоби (текстовий редактор, компілятор, інтерпретатор, компонувальник, завантажувач і налагоджувач) зазвичай утворюють **систему програмування**, або **інтегроване середовище**. Крім них, до складу цієї системи входить **бібліотека стандартних підпрограм**, які можна використовувати під час створення програми.

Що краще розробник програми володіє технологіями, інструментом і бібліотеками, то його робота ефективніша.

Лекція 1
Вступ. Основні поняття

План

1. Преамбула
2. Алфавіт і лексика мови C++, ідентифікатори.
3. П’ять основних типів дних
4. Змінні-1
    4.1. Ініціалізація змінних
5. Операції в C++
6. Структура програми
    6.1. Стейтменти
    6.2. Вирази
    6.3. Функції
    6.4. Бібліотеки
    6.5. Приклад простої програми
7. Функції
    7.1. Значення, що повертаються
    7.2. Тип повернення void
    7.3. Повернення значень функцією main()
    7.4. Детальніше про значення, що повертаються
    7.5. Ікладені функції
    7.6. Локальні змінні
8. Фази компіляції
9. Замість висновків

1. Преамбула.

Мова програмування C++ (вимовляється як *«Сі плюс плюс»*) була розроблена Б'ярном Страуструпом в *Bell Telephone Laboratories* в 1979 році в якості доповнення до мови С. Вона додала безліч нових функціональних можливостей в мову С, однією з яких було об'єктно-орієнтованість даної мови. Щодо об'єктно-орієнтованого програмування (ООП) і його відмінностей від традиційних методів програмування ми поговоримо на відповідних заняттях.

Мова С++ була схвалена комітетом ISO в 1998 році і потім знову в 2003 році (під назвою **C++03**). Потім були наступні оновлення версій (раз в 3 роки), котрі додали ще більше функціональних можливостей:

- **C++11** в 2011 році;
- **C++14** в 2014 році;
- **C++17** в 2017 році;
- **C++20** в 2020 році.

Філософія мови програмування C++

Філософію мови програмування С++ можна визначити виразом "довіряти програмісту". Наприклад, компілятор не заважатиме вам зробити щось нове (що має сенс), але також не заважатиме вам зробити щось таке, що може призвести до збою. Це одна з головних причин, чому так важливо знати те, що можете робити в С/С++, так і те, що ви не повинні робити.

2. Алфавіт і лексика мови C++, ідентифікатори.

Алфавіт C++ складається з латинських великих і маленьких літер, арабських чисел та спеціальних символів:

+ - * / < > == | & ! \ ` ' @ # $ % ^ ? _ : ; , . ( ) [ ] { } " .

Лексика C++:

- ідентифікаторі відрізняються по першим 32 символам, причому великі і маленькі літери відрізняються, тобто name і Name різні змінні
- зарезервовані та ключові слова мови, подані в таблиці

Таблиця 1 Ключові ідентифікатори C++

asm	const	dynamic_cast	goto
auto	const_cast	else	if
break	continue	enum	inline
case	default	extern	int
catch	delete	float	long
char	do	for	new
class	double	friend	operator

private	struct	virtual	
protected	switch	void	
public	template	while	
register	this		
return	throw		
short	try		
signed	typedef		
sizeof	typeid		
static	union		
static_cast	unsigned		

Ідентифікатори

У мові C/C++ імена змінних, функцій, міток і інших об'єктів, створених користувачем, називаються *ідентифікаторами* (indentifiers). Ідентифікатори можуть складатися з одного або декількох символів. Перший символ ідентифікатора повинен бути буквою або символом підкреслення, а наступні символи повинні бути буквами, цифрами або символами підкреслення. Нижче наведені приклади правильних і неправильних ідентифікаторів.

Правильно	***Неправильно***
Count	lcount
test23	hi!there
high_balance	high...balance

Однак не всі символи, що утворять ідентифікатор, уважаються значущими. Якщо ідентифікатор використовується в процесі редагування зовнішніх зв'язків, то значущими вважаються, принаймні, шість його перших символів. Ці ідентифікатори називаються *зовнішніми іменами* (external names). До них ставляться імена функцій з глобальними змінними, використовуваними декількома файлами. У мові C++ немає обмежень на довжину ідентифікаторів і значущими вважаються, принаймні, 1024 перших символів.

Символи, набрані у верхньому й нижньому регістрі, різняться. Отже, count, Count і COUNT — це різні ідентифікатори.

Ключові слова не можна використовувати як ідентифікатори. Крім того, ідентифікатори не повинні збігатися з іменами функцій зі стандартних бібліотек.

3. П'ять основних типів даних

У C++ існують п'ять елементарних типів даних: символ, ціле число, число із плаваючою крапкою, число із плаваючою крапкою подвоєної точності й змінна,

що не має значень. Їм відповідають наступні ключові слова: **char, int, float, double** і **void.** Всі інші типи даних у мові С++ створюються на основі елементарних типів, зазначених вище. Розмір змінних і діапазон їхніх значень залежить від типу процесора й компілятора. Однак у всіх випадках розмір символу дорівнює 1 байт. Розмір цілочисельної змінної звичайно дорівнює довжині машинного слова, прийнятої в конкретній операційній системі. Однак, прагнучи до машино незалежності програм, варто уникати конкретних припущень про розмір цілочисельних змінних. Важливо чітко розуміти, що в мові С++ акцентується лише на *мінімальному діапазоні* у якому змінюються значення змінних кожного типу, а не їхній розмір у байтах.

Точне подання чисел із плаваючою крапкою залежить від їхньої конкретної реалізації. Розмір цілого числа звичайно дорівнює довжині машинного слова, прийнятої в операційній системі. Значення змінних типу **char,** як правило, використовуються для подання символів, передбачених у системі кодування ASCII. Значення, що виходять за межі припустимого діапазону, на різних комп'ютерах обробляються по-різному.

Діапазон зміни змінних типу **float** і **double** залежить від способу подання чисел із плаваючою крапкою. У кожному разі, цей діапазон досить широкий. Мінімальна кількість цифр, що визначають точність чисел із плаваючою крапкою, зазначене в табл. 2.

Таблиця 2. Характеристики основних типів даних С++

Ім’я типа	Байти	Діапазон значень
int	4	від -2 147 483 648 до 2 147 483 647
bool	1	false або true
char	1	від -128 до 127
double	8	1,7E +/- 308 (15 знаків)
float	4	3.4E+/- 38

Тип **Void** використовується для визначення функції, що не повертає ніяких значень, або для створення узагальненого покажчика (generic pointer).

4. Змінні-1

Змінна (variable) являє собою ім'я комірки пам'яті, яку можна використовувати для зберігання значення, що модифікується. Всі змінні повинні бути оголошені до свого використання. Нижче наведений загальний вид оголошення змінної

тип список_змінних;

Тут слово *тип* означає один із припустимих типів даних, включаючи модифікатори, а *список_змінних* може складатися з одного або декількох ідентифікаторів, розділених комами. Розглянемо кілька прикладів оголошень.

```
int  i, j, L;
 short   int   si;
 unsigned int ui;
 double balance, profit, loss;
```

Де оголошуються змінні

Як правило, змінні повідомляють у трьох місцях: усередині функцій, у визначенні параметрів функції й за межами всіх функцій. Відповідно такі змінні називаються локальними, формальними параметрами й глобальними.

4.1 Ініціалізація змінних

```
int a = 9;     int a=4*6+b;
```

Копіююча ініціалізація (або ще **“ініціалізація копіюванням”**) з допомогою знаку рівності (=).

```
int b(9);
```

Пряма ініціалізація з допомогою круглих дужок.

```
int c{ 6 };
```

uniform-ініціалізація

Пряма ініціалізація може працювати краще з деякими типами даних, копіююча ініціалізація — з іншими типами даних.

Пряма або копіююча ініціалізації працюють не з усіма типами даних (наприклад, ви не зможете використовувати ці форми для ініціалізації списку значень).

У спробі забезпечити єдиний механізм ініціалізації, який буде працювати з усіма типами даних, в C++11 додали нову форму ініціалізації, яка називається **uniform-ініціалізацією**:

Ініціалізація змінної з порожніми дужками вказує на ініціалізацію за замовчуванням (змінній присвоюється 0):

```
int  c{};  // ініціалізація змінної за замовчуванням - значенням 0
```

У uniform-ініціалізації є ще одна додаткова перевага: ви не можете присвоювати змінній значення, яке не підтримує її тип даних — компілятор видасть попередження або повідомлення про помилку, наприклад:

```
int value{ 4.5 }; // помилка: цілочисельна змінна не може містити не цілочисельні значення
```

```
int a;
cin >> a;
```

Введення значення з клавіатури

```
int g = rand();
```

За допомогою функції генерування випадкових чисел

5. Операції в С++

Арифметичні операції С++ наводяться нижче в таблиці. За винятком операції обчислення остачі від ділення, яка має сенс лише для символьного і цілого типу, арифметичні операції застосовуються до всіх арифметичних типів, в тому числі булевих.

Арифметичні операції

Символ операції	Назва	Спосіб використання
*	Множення	op1 * op2
/	Ділення	op1 / op2
%	Остача	op1 % op2
+	Додавання	op1 + op2
-	Віднімання	op1 - op2
,	Кома	op1, op2

Операція “кома” використовується тоді, коли на місці, синтаксично придатному для розміщення одного виразу, необхідно розмістити декілька виразів. Значенням виразу, побудованого за допомогою цієї операції, вважається значення її другого операнда. Перший операнд виявляється як би зайвим, на перший погляд його обчислення стає безрезультатним. Але це не так, оскільки в С++ існують операції, здатні змінювати значення змінних, зокрема, операцією є саме присвоєння, а також операції інкременту і декременту.

Операції порівняння

Символ операції	Назва	Спосіб використання
<	Менше	op1 < op2
<=	Менше або рівне	op1 <= op2
>	Більше	op1 > op2
>=	Більше або рівне	op1 >= op2
= =	Рівне	op1 == op2
!=	Не рівне	op1 != op2

Особливу увагу варто звернути на специфічне позначення рівності. Вживання знаку присвоєння ”=” замість знаку рівності ”= =” — одна з самих типових помилок в С/С++-програмах.

С++ успадкував від С два типи логічних операцій. Крім звичайних заперечення, кон'юнкції і диз'юнкції, є ще побітові логічні операції, які застосовуються до будь-яких інтегральних типів.

Складне присвоєння

До операторів складного присвоювання відносяться +=, -=, *=, /=.

Оператор x+=p; призначений для збільшення x на величину p. Оператор x-=p; призначений для зменшення x на величину p. Оператор x*=p; призначений для множення x на p. Оператор x/=p; призначений для розподілу x на p.

Операції цілочисельної арифметики

До операцій цілочисельної арифметики відносяться:
цілочисельне ділення /;
остача від ділення %.

При цілочисельному діленні операція / повертає цілу частина частки (дробова частина відкидається), а операція % - остача від ділення. Нижче наведені приклади цих операцій

11 % 4 = 3
11/4 = 2
7 % 3 = 1
7/3 = 2
26/5 = 5
26 % 5 = 1

Операції збільшення (інкримент) і зменшення (декремент)

У мові С++ є операції збільшення (++) і зменшення (--) на одиницю.
Оператор
p=p+1;
можна записати в префіксній формі
++p;
так і в постфіксній
p++;

Ці форми відрізняються при використанні їх у виразах. Якщо знак декремента (інкримента) передує операнду, то спочатку виконується збільшення (зменшення) значення операнда, а потім операнд бере участь у виразі.
Наприклад,
x=12;
y=++x;

У результаті в у буде зберігатися число 13.

Якщо знак декремента (інкремента) треба після операнда, то спочатку операнд бере участь у виразі, а потім виконується збільшення (зменшення) значення операнда.
Наприклад,
x=12;
y=x++;

У результаті в у буде зберігатися число 12.

Логічні операції

Символ операції	Назва	Спосіб використання
!	Заперечення	! op
&&	Кон'юнкція	op1 && op2
||	Диз'юнкція	op1 || op2
? :	Імплікація	(op1 ? op2: op3)

Аргументи і результати логічних операцій, крім імплікації, булеві. Імплікація служить для створення умовних виразів. Її перший операнд буллевий, а два інші довільні арифметичні, результат арифметичний. Визначають

імплікацію тотожності
(true ? x: y) == x
(false ? x: y) == y

Крім названих вище логічних операцій, в C++ є також побітові логічні операції. Вони застосовуються до арифметичних типів і самі дають результат арифметичного типу. Найпростіше ці операції вживати до типу unsigned int або unsigned long , що відповідатиме роботі з машинними словами. У випадку коротших типів або при наявності знаку результат може залежати від типу компілятора. Це значить, що одна і та ж програма даватиме після підготовки до виконання різними компіляторами даватиме різні результати. Але сучасні технології програмування віддають перевагу програмам, не залежним від компілятора і навіть платформи. Такі програми називають переносимими (portable).

Логічні побітові операції

Символ операції	Назва	Спосіб використання
~	Заперечення	~ op
<<	Зсування вліво	op1 << op2
>>	Зсування вправо	op1 >> op2
&	Кон'юнкція	op1 & op2
^	Виключна диз'юнкція	op1 ^ op2
|	Диз'юнкція	op1 | op2

Побітові логічні операції — це програмування на низькому, близькому до машинного, рівні. Програмування рівнем вище використовує стандартний клас bitset , визначений у стандартній бібліотеці.

В C++ існують прототипи стандартних математичних функцій. Перелік цих функцій наведено в табл.

Таблиця стандартних математичних функцій

abs (x)	Повертає абсолютне значення x
acos (x)	Повертає арккосинус x
asin (x)	Повертає арксинус x
atan (x)	Повертає арктангенс x
cbrt (x)	Повертає корінь кубічний з x
cos (x)	Повертає косинус x
cosh (x)	Повертає гіперболічний косинус x
exp (x)	Повертає значення E^x
expm1 (x)	Повертає e^x -1
fabs (x)	Повертає абсолютне значення плаваючого x
fdim (x, y)	Повертає додатну різницю між x та y
hypot (x, y)	Повертає sqrt ($x^2 + y^2$) без проміжного переповнення або недоливу
fma (x, y, z)	Повертає x * y + z без втрати точності

fmax (x, y)	Повертає найвище значення плаваючих x та y
fmin (x, y)	Повертає найнижче значення плаваючих x та y
fmod (x, y)	Повертає залишок з плаваючою комою від x / y
log(x)	натуральний логарифм від x
log10(x)	десятковий логарифм числа x
pow (x, y)	Повертає значення x у ступінь y
sin (x)	Повертає синус x (x в радіанах)
sinh (x)	Повертає гіперболічний синус подвійного значення
tan (x)	Повертає тангенс кута
tanh (x)	Повертає гіперболічний тангенс подвійного значення

Операції округлення:

round(), - математичне округлення
ceil(), - округлення до більшого
floor() - округлення в сторону меншого
trunc() – відкидання дробової частини

6. Структура програми

6.1 Стейтменти

Стейтмент (англ. ***"statement"***) — це найбільш поширений тип інструкцій у програмах. Це і є та сама найменша незалежна одиниця в **мові програмування C++**. Стейтмент в програмуванні — це те ж саме, що і "речення" в українській мові. Ми використовуємо речення для того, щоб виразити якусь думку/ідею. У C++ ми пишемо стейтменти, щоб виконати якесь завдання. **Всі стейтменти в C++ закінчуються крапкою з комою**.

Є дуже багато різних видів стейтментів в C++. Переглянемо найбільш поширені з них:

```
1  int x;
2  x = 5;
3  std::cout << x;
```

int x — це **стейтмент оголошення** (англ. *"statement declaration"*). Він повідомляє компілятору, що x є змінною.
x = 5 — це **стейтмент присвоювання** (англ. *"assignment statement"*). Тут ми присвоюємо значення 5 змінній x.
std::cout << x; — це **стейтмент виводу** (англ. *"output statement"*). Ми виводимо значення змінної x на екран.

6.2 Вирази

Компілятор також здатний опрацьовувати вирази. **Вираз** (англ. *"expression"*) — це математичний об'єкт, який генерує певне значення. Наприклад, в математиці вираз 2 + 3 генерує значення 5.

Виразами можуть бути:

- значення (наприклад, 2, 4);
- змінні (наприклад, x, y);
- оператори (наприклад, +, -);
- функції.

Вони можуть бути як одним значенням (наприклад, 2 чи x), так і комбінацією значень (наприклад, 2 + 3, 2 + x, x + y чи (2 + x) * (y - 3)). Наприклад, x = 2 + 3; — це коректний стейтмент присвоювання. Вираз 2 + 3 генерує результат: значення 5, яке потім присвоюється змінній x.

6.3 Функції

В C++ стейтменти об'єднуються в блоки — функції. **Функція** — це послідовність стейтментів. Кожна програма в C++ повинна містити **головну функцію main()**. Саме з першого стейтмента в main() і починається виконання програми. Функції, як правило, виконують конкретне завдання. Наприклад, функція max() може містити стейтменти, які визначають максимальне число з двох переданих їй, а функція calculateGrade() може обчислювати оцінку студента.

6.4 Бібліотеки

Бібліотека — це набір скомпільованого коду (наприклад, функцій), який був "упакований" для повторного використання в інших програмах. За допомогою бібліотек можна розширити функціонал програм. Наприклад, якщо ви пишете гру, то вам доведеться підключити бібліотеку звуку або графіки (якщо ви самостійно не хочете їх писати з нуля).

Мова C++ не така вже й велика, як ви могли б подумати. Проте, вона поставляється в комплекті зі **Стандартною бібліотекою C++**, яка надає додатковий функціонал. Однією з найбільш часто використовуваних частин Стандартної бібліотеки C++ є **бібліотека iostream**, яка дозволяє виводити інформацію на екран і опрацьовувати дані, які вводить користувач.

6.5 Приклад простої програми в C++

Тепер, коли у вас є загальне уявлення про те, що таке стейтменти, функції та бібліотеки, давайте розглянемо програму "Hello, world!":

```
1 #include <iostream>
2
3 int main()
4 {
5     std::cout << "Hello, world!";
6     return 0;
7 }
```

Рядок №1: Спеціальний тип інструкції, який називається **директивою препроцесора**. Директиви препроцесора повідомляють компілятору, що йому потрібно виконати певне завдання. В цьому випадку ми повідомляємо компілятору, що хотіли б підключити вміст заголовкового файлу iostream до нашої програми. Заголовковий файл iostream дозволяє нам отримати доступ

до функціоналу бібліотеки iostream для можливості виведення інформації на екран.
Рядок №2: Порожній простір, який ігнорується компілятором.
Рядок №3: Оголошення головної функції main().
Рядок №4 і №7: Вказуємо компілятору область функції main(). Все, що знаходиться між відкриваючою фігурною дужкою в рядку №4 і закриваючою фігурною дужкою в рядку №7, — вважається частиною функції main().
Рядок №5: Наш перший стейтмент (який закінчується крапкою з комою) — стейтмент виводу. std::cout — це спеціальний об'єкт за допомогою якого ми можемо виводити дані на екран. << — це оператор виводу. Все, що ми відправляємо в std::cout, — виводиться на екран. Тут ми виводимо текст "Hello, world!".
Рядок №6: **Оператор повернення return**. Коли програма завершує своє виконання, функція main() передає значення, яка вказує на результат виконання програми (успішно чи ні), назад в операційну систему. Якщо оператор return повертає значення 0, то це означає, що все добре! Будь-які ненульові числа використовуються для того, щоб вказати, що щось пішло не так.

Більш функціональніший шаблон функції **main()** описується наступним кодом:

```
1   #include <iostream>
2   #include "windows.h"
3   using namespace std;
4
5   int main()
6   {
7           SetConsoleCP(1251);
8           SetConsoleOutputCP(1251);
9   // все що потрібно тут!!!
10          // і тут також
11          cout << endl;
12          system("pause");
13          return 0;
14  }
```

З нового тут появилось:

1. **Кирилиця в консолі**

```
#include <Windows.h>
…
SetConsoleCP(1251);
SetConsoleOutputCP(1251);
```

2. **Затримка консолі**

```
system("pause");
```

також можна використовувати:
cin.clear();
cin.ignore(32767, '\n');
cin.get();
system("pause"); - можуть працювати не на всіх типах операційних систем

3. підключення простору імен std **рядок 3** – дане підключення дозволяє не писати постійно std::cout
4. **в рядку 11** ми переводимо курсор виведення на рядок нище команда **endl**, тим самим створюємо порожній рядок в консолі виведення, що дозволяє візуально відділити різну інформацію при виведенні на екра.

7. Функції

Функція — це послідовність стейтментів для виконання певного завдання. Часто програми перериваються виконанням одних функцій заради виконання інших. Ви це постійно робите в реальному житті, наприклад, ви читаєте книгу і згадали, що повинні були зробити телефонний дзвінок. Ви залишаєте *закладку* в своїй книзі, берете телефон і набираєте номер. Після того, як ви вже поговорили, ви повертаєтеся до тієї сторінки в книзі, на якій ви зупинилися.

Програми в C++ працюють схожим чином. Іноді, коли програма виконує код, вона може зіткнутися з викликом функції. **Виклик функції** — це **вираз**, який вказує процесору перервати виконання поточної функції і приступити до виконання іншої функції. Процесор "залишає закладку" в поточній точці виконання, а потім виконує функцію, що викликається. Коли виконання функції, що викликається, — завершено, то процесор повертається до "закладки" і відновлює виконання перерваної функції.

Функція, в якій знаходиться виклик, називається **викликаючою функцією** (англ. ***"caller"***). Наприклад:

```
1  #include <iostream> // для std::cout і std::endl
2
3  // Оголошення функції doPrint(), яку ми будемо викликати
4  void doPrint() {
5      std::cout << "In doPrint()" << std::endl;
6  }
7
8  // Оголошення функції main()
9  int main()
10 {
11     std::cout << "Starting main()" << std::endl;
12     doPrint(); // перериваємо виконання main() викликом
   функції doPrint(). Функція main() в даному випадку є
   викликаючою функцією
13     std::cout << "Ending main()" << std::endl;
14     return 0;
15 }
```

Результат виконання програми:

```
Starting main()
In doPrint()
Ending main()
```

Ця програма починає своє виконання з першого рядка функції main(), в якому виводиться на екран Starting main(). Другий рядок функції main() викликає функцію doPrint(). На цьому етапі виконання стейтментів у функції main() призупиняється і процесор переходить до виконання стейтментів всередині функції doPrint(). Перший (і єдиний) рядок в doPrint() виводить текст In doPrint(). Коли процесор завершує виконання doPrint(), він повертається назад в функцію main() до тієї точки, на якій зупинився. Отже, наступним стейтментом є вивід рядка Ending main() на екран.

Зверніть увагу, для виклику функції потрібно вказати її ім'я і список параметрів в круглих дужках (). У вищенаведеному прикладі параметри не використовуються, тому круглі дужки порожні.

Правило: Не забувайте вказувати круглі дужки () при виклику функцій.

7.1 Значення, що повертаються

Коли функція main() завершує своє виконання, вона повертає цілочисельне значення назад в операційну систему, використовуючи **оператор return**.

Функції, які ми пишемо, також можуть повертати значення. Для цього потрібно вказати **тип повернення значення**. Він вказується при оголошенні функції, перед її ім'ям. Зверніть увагу, що тип повернення не вказує, яке саме значення повертатиметься. Він вказує тільки тип цього значення.

Після цього всередині функції, що викликається, ми використовуємо оператор return, щоб вказати фактичне **значення, що повертається**.

Розглянемо просту функцію, яка повертає цілочисельне значення:

```
1  #include <iostream>
2
3  // int означає, що функція повертає цілочисельне значення у
   викликаючу функцію
4  int return7()
5  {
6        // Ця функція повертає цілочисельне значення, тому ми повинні
   використовувати оператор return
7        return 7; // повертаємо число 7 у викликаючу функцію
8  }
9
10 int main()
11 {
12       std::cout << return7() << std::endl; // на екран виведеться 7
13       std::cout << return7() + 3 << std::endl; // на екран виведеться
   10
14
15       return7(); // значення 7, що повертається, - ігнорується, тому
   що main() з ним нічого не робить
16
17       return 0;
18 }
```

Результат виконання програми:

```
7
10
```

Тепер давайте розберемося детально:

Перший виклик функції return7() **рядок 12** повертає 7 у викликаючу функцію, де воно потім передається в **std::cout** для виводу на екран.

Другий виклик функції return7() **рядок 13** знову повертає 7 у викликаючу функцію. Вираз 7 + 3 генерує результат 10, який виводиться на екран.

Третій виклик функції return7() **рядок 15** знову повертає 7 у викликаючу функцію. Проте main() нічого з ним не робить, тому нічого і не відбувається (значення, що повертається, просто ігнорується).

> **Примітка**: Значення, що повертаються, не виводяться на екран, якщо їх не передавати об'єкту std::cout. В останньому виклику функції return7() значення не надсилається в std::cout, тому нічого і не відбувається.

7.2 Тип повернення void

Функції можуть і не повертати значення. Щоб повідомити компілятору, що функція не повертає значення, потрібно використати **тип повернення void**. Погляньмо ще раз на функцію doPrint() з вищенаведеного прикладу:

```
1 void doPrint() // void - це тип повернення
2 {
3     std::cout << "In doPrint()" << std::endl;
4     // Ця функція не повертає значення, тому і оператор return
  тут не потрібен
5 }
```

Ця функція має тип повернення void, який означає, що функція не повертає значення. Оскільки значення не повертається, то і оператор return тут не потрібен.

Ось ще один приклад використання функції типу void:

```
1  #include <iostream>
2
3  // void означає, що функція не повертає значення
4  void returnNothing()
5  {
6      std::cout << "Hi!" << std::endl;
7      // Ця функція не повертає значення, тому і оператор return
   тут не потрібен
8  }
9
10 int main()
11 {
12     returnNothing(); // функція returnNothing() викликається,
   але в main() нічого не повертається
13
14     std::cout << returnNothing(); // помилка: цей рядок не
   скомпілюється (вам потрібно буде його закоментувати)
15     return 0;
16 }
```

У першому виклику функції returnNothing(), **рядок 12,** виводиться Hi!, але нічого не повертається назад у викликаючу функцію. Точка виконання повертається назад в main(), де програма продовжує своє виконання.

Другий виклик функції returnNothing(), **рядок 14.** навіть не скомпілюється. Функція returnNothing() має тип повернення void, який означає, що ця функція не повертає значення. Однак main() намагається відправити це значення (яке не повертається) в std::cout для виведення на екран. std::cout не може опрацювати цей випадок, оскільки не було надано значення для виводу. Отже, компілятор видасть помилку. Вам потрібно буде **закоментувати** цей рядок, щоб компіляція пройшла успішно.

7.3 Повернення значень функцією main()

Тепер у вас є розуміння того, як працює функція main(). Коли програма виконується, операційна система викликає функцію main() і починається її виконання. Стейтменти в main() виконуються послідовно. В кінці функція main() повертає цілочисельне значення (зазвичай 0) назад в операційну систему. Тому main() оголошується як int main().

Чому потрібно повертати значення назад в операційну систему? Справа в тому, що значення, що повертається функцією main(), є **кодом стану**, який повідомляє операційній системі про те, чи успішно було виконання програми. Зазвичай, значення 0 (нуль) означає що все пройшло успішно, тоді як будь-яке інше значення означає невдачу/помилку.

Зверніть увагу, за стандартами мови C++ функція main() повинна повертати цілочисельне значення. Однак, якщо ви не вкажете return в кінці функції main(), то компілятор поверне 0 автоматично, якщо ніяких помилок не буде. Проте все ж рекомендується вказувати оператор return в кінці функції main() і використовувати тип повернення int для функції main().

7.4 Детальніше про значення, що повертаються

По-перше, якщо тип повернення функції не є void, то вона повинна повертати значення зазначеного типу (використовуючи оператор return). Єдиний виняток — це функція main(), яка повертає 0, якщо не надано інше значення.

По-друге, коли процесор зустрічає в функції оператор return, він негайно виконує повернення значення назад у викликаючу функцію і точка виконання також переходить у викликаючу функцію. Будь-який код, який знаходиться за оператором return у функції — ігнорується.

Функція може повертати тільки одне значення через оператор return у викликаючу функцію. Це може бути або число (наприклад, 7), або значення змінної, або вираз (який генерує результат), або певне значення з набору можливих значень.

Є способи обійти правило повернення одного значення, але про це трішки пізніше.

Нарешті, автор функції вирішує, що означатиме значення, яке повертає функція. Деякі функції використовують повернені значення в якості кодів стану для надання інформації щодо результату виконання функції (успішне виконання чи ні). Інші функції повертають певне значення з набору можливих значень, ще інші функції взагалі нічого не повертають.

Повторне використання функцій

Одну і ту ж функцію можна викликати декілька разів, навіть в різних програмах, що дуже корисно:

```
1   #include <iostream>
2
3   // Функція getValueFromUser() отримує значення від користувача,
    а потім повертає його назад у викликаючу функцію
4   int getValueFromUser()
5   {
6       std::cout << "Enter an integer: ";
7       int x;
8       std::cin >> x;
9       return x;
10  }
11
12  int main()
13  {
14      int a = getValueFromUser(); // перший виклик функції
    getValueFromUser()
15      int b = getValueFromUser(); // другий виклик функції
    getValueFromUser()
16
17      std::cout << a << " + " << b << " = " << a + b <<
    std::endl;
18
19      return 0;
20  }
```

Результат виконання програми:

```
Enter an integer: 4
Enter an integer: 9
4 + 9 = 13
```

Тут виконання функції main() переривається 2 рази. Зверніть увагу, в обох випадках отримане значення від користувача зберігається в змінній x, а потім передається назад в функцію main() за допомогою оператора return, де присвоюється змінній a або b!

Також main() не є єдиною функцією, яка може викликати інші функції. Будь-яка функція може викликати будь-яку іншу функцію!

```
1   #include <iostream>
2
3   void printO()
4   {
5       std::cout << "O" << std::endl;
6   }
7
8   void printK()
9   {
10      std::cout << "K" << std::endl;
11  }
12
```

```
13  // Функція printOK() викликає як printO(), так і
14  printK()
    void printOK()
15  {
16      printO();
17      printK();
18  }
19
20  // Оголошення функції main()
21  int main()
22  {
23      std::cout << "Starting main()" << std::endl;
24      printOK();
25      std::cout << "Ending main()" << std::endl;
26      return 0;
27  }
```

Результат виконання програми:

```
Starting main()
O
K
Ending main()
```

7.5 Вкладені функції

В мові C++ одні функції можуть бути оголошені всередині інших функцій (тобто бути вкладеними). Наступний код викличе помилку компіляції:

```
1  #include <iostream>
2
3  int main()
4  {
       int boo() // ця функція знаходиться всередині
5  функції main(), що є заборонено
6      {
7          std::cout << "boo!";
8          return 0;
9      }
10
11     boo();
12     return 0;
13 }
```

Правильно ось так:

```
1  #include <iostream>
2
3  int boo() // тепер вже не в функції main()
4  {
5      std::cout << "boo!";
6      return 0;
```

```
7  }
8
9  int main()
10 {
11      boo();
12      return 0;
13 }
```

7.6 Локальні змінні

Змінні, оголошені усередині функції, називаються *локальними* (local variables). Локальні змінні можна використовувати тільки в операторах, розташованих усередині блоку, де вони оголошені. Інакше кажучи, локальні змінні невидимі зовні їхнього блоку. Нагадаємо, що блок обмежений відкриваючою й закриваючою фігурною дужками.

Найчастіше локальні змінні оголошуються усередині функцій. Розглянемо два приклади.

```
void  func1(void)
{
      int x;
      x = 10
}

void func2(void)
{
      int x;
      x = -199;
     }
```

Змінна x оголошена двічі: спочатку – у функції **find ()**, а потім — у функції **func2().** Змінна x з функції **fund** () не має ніякого відношення до змінного x, оголошеної усередині функції **func2 ()**. Кожна із цих змінних існує тільки усередині блоку, де вона була оголошена.

З міркувань зручності й за традицією більшість програмістів оголошують всі змінні, використовувані у функції, відразу після відкриваючої фігурної дужки й перед всіма іншими операторами. Однак варто мати на увазі, що локальні змінні можна повідомляти в будь-якому місці блоку. Розглянемо наступний приклад.

```
void f(void)
{
      int   t;
      cin >> t;
      if (t = = l) {
           char s[80];   /* Ця змінна створюється тільки при
                                          вході в даний блок
*/
           cout << "Уведіть ім'я: ";
           gets(s); /*  Якісь операції   ...   */
      }
     }
```

У цьому фрагменті локальна змінна **s** створюється при вході в блок if і руйнується відразу після виходу з нього. Крім того, змінна **s** є видимою тільки

усередині блоку **if**, і до неї неможливо звернутися ззовні, навіть із інших частин функції, що містить даний блок.

Оголошення змінних усередині блоків дозволяє уникнути побічних ефектів. Оскільки локальна змінна поза блоком не існує, її значення неможливо змінити ненавмисно.

Оскільки локальна змінна створюється при вході в блок, де вона оголошена, і руйнується при виході з нього, її значення губиться. Це особливо важливо враховувати при виклику функції. Локальні змінні функції створюються при її виклику й знищуються при поверненні керування в зухвалий модуль. Це значить, що між двома викликами функції значення її локальних змінних не зберігаються. (Змусити їх зберігати свої значення можна за допомогою модифікатора **static**.)

За замовчуванням локальні змінні зберігаються в стеці. Оскільки стік є динамічною структурою, і обсяг займаної їм пам'яті постійно змінюється, стає зрозумілим, чому локальні змінні в принципі не можуть зберігати свої значення між двома викликами функції, усередині якої вони оголошені.

Формальні параметри

Якщо функція має аргументи, варто оголосити змінні, які будуть приймати їхні значення. Ці змінні називаються *формальними параметрами* (formal parameters). Усередині функції вони нічим не відрізняються від інших локальних змінних. Як показано в наведеному нижче фрагменті програми, оголошення таких змінних повинне розміщатися відразу після ім'я функції й полягати в дужки.

Приклад:

```
/*  Функція  повертає 1, якщо символ з є частиною рядка s; у
протилежному випадку вона повертає 0 */
int  is_in(char* s, char c)
{
     while (*s)
          if (*s == c)  return 1;
          else  s++;
     return 0;
    }
```

Функція **is_in()** має два параметри: **s** і **c.** Вона повертає 1, якщо символ з є частиною рядка **s**, у противному випадку функція повертає 0.

Тип формальних параметрів задається при їхньому оголошенні. Після цього їх можна використовувати як звичайні локальні змінні. Урахуйте, що, як і локальні змінні, формальні параметри є динамічними й руйнуються при виході з функції.

Формальні параметри можна використовувати в будь-яких конструкціях і вираженнях. Незважаючи на те що свої значення вони одержують від аргументів, переданих ззовні функції, у всім іншому вони нічим не відрізняються від локальних змінних.

Глобальні змінні

На відміну від локальних, *глобальні змінні* (global variables) доступні з будь-якої частини програми й можуть бути використані де завгодно. Крім того, вони зберігають свої значення на всім протязі виконання програми. Оголошення глобальних змінних повинні розміщатися поза всіма функціями. Ці змінні можна використовувати в будь-якому вираженні, у якому би блоці воно не перебувало.

У наведеному нижче фрагменті програми змінна **count** оголошена поза всіма функціями. Незважаючи на те, що її оголошення розташоване до функції **main (),** цю змінну можна було б з таким же успіхом оголосити в іншому місці, але поза функціями й до її першого використання. І все-таки найкраще розміщати оголошення глобальних змінних на самому початку програми.

```
#include  <iostream>
int   count;   /*   Змінна   count   є   глобальною   */
void  func1(void);
void func2(void);

int main(void)
{
     count = 100;
     func1();

     return  0;
}

void  func1(void)
{
     int  temp;
     temp = count;
     func2();
     std::cout<<"count   = "<< count;   /*   Виведе   число   100.
*/
}

void func2(void)
{
     int   count;
     for (count = 1; count < 10; count++)
          putchar('.');
    }
```

Придивіться до цієї програми уважніше. Помітьте, що, хоча ні функція **main ()**, ні функція **fund ()** не містять оголошення змінної **count,** обидві ці функції успішно неї використовують. Однак усередині функції **func2()** оголошена локальна змінна **count.** Коли функція **func2 ()** посилається на змінну **count,** вона використовує лише локальну змінну, а не глобальну. Якщо глобальна і локальна змінні мають однакові імена, то всі посилання на ім'я змінної усередині блоку будуть ставитися до локальної змінної й не впливати на її глобальну тезку. Можливо, це зручно, однак про цю особливість легко забути, і тоді дії програми можуть стати непередбаченими.

Глобальні змінні зберігаються в спеціально відведеному для них місці. Вони виявляються досить корисними, якщо різні функції в програмі використовують ті самі дані. Однак, якщо в них немає особою необхідності, глобальних змінних варто уникати. Справа в тому, що вони займають пам'ять під час усього виконання програми, навіть коли вже не потрібні. Крім того, використання глобальних змінних там, де можна було б обійтися локальними, послабляє незалежність функцій, оскільки вони змушені залежати від сутності, певної в іншому місці. Отже, застосування великої кількості глобальних змінних підвищує ймовірність помилок внаслідок непередбачених побічних ефектів.

Більшість проблем, що виникають при розробці більших програм, є наслідком непередбаченої зміни значень змінних, які використовуються в будь-якому місці програми. Це може трапитися, якщо в програмі оголошено занадто багато глобальних змінних.

8. Фази компіляції

Після створення вихідний текст компілюється. Етап компіляції і редагування зв'язків розпадається на кілька фаз.

1. Вихідний файл кодується основним набором текстових символів, причому наприкінці кожного рядка програми розставляються символи переходу на новий рядок. Кожен символ, що не входить в основний набір текстових символів, замінюється відповідним універсальним символом.

2. З проміжного тексту видаляються символи переходу на новий рядок, і фізичні рядки вихідного коду стають логічними. Якщо в процесі трансформації тексту випадково виникнуть універсальні символи, поводження компілятора стандартом не регламентується. Це стосується і ситуації, коли непорожній текстовий файл не завершується символом переходу на новий рядок.

3. Вихідний файл розкладається на лексеми і послідовності роздільників, включаючи коментарі. При цьому текст програми не повинний завершуватися незакінченою лексемою чи незавершеним коментарем (наприклад, <include чи /* неповний коментар). Кожен коментар замінюється пробілом.

Символи переходу на новий рядок зберігаються. Обробка інших роздільників залежить від конкретного компілятора. Інтерпретація символів, що утворюють текст програми, залежить від контексту (наприклад, символ < у директиві #include <ім'я файлу> і символ < в умовному вираженні обробляються по-різному).

4. Виконуються директиви препроцесора і макровизначення. Якщо в процесі конкатенації лексем випадково утвориться універсальний символ, поводження компілятора не регламентується. У результаті виконання директиви #include у вихідний текст програми вставляється текст відповідного заголовного файлу, до якого рекурсивно застосовуються фази 1–4.

5. Кожен символ з основного набору текстових символів, escape-послідовність чи універсальний символ, що є частиною символьного чи рядкового літерала, конвертується у відповідний елемент із набору керуючих символів.

6. Суміжні рядкові і розширені рядкові літерали конкатенуються.

7. Роздільники між лексемами ігноруються. Кожна лексема препроцесора перетворюється в звичайну лексему. Отримані лексеми піддаються синтаксичному і семантичному аналізу. Поняття “вихідний файл” і “одиниці трансляції” є абстрактними — їм не обов'язково відповідають фізичні файли. Інакше кажучи, щоб скомпілювати програму, її не обов'язково зберігати у файлі. (Це зауваження стосується, скоріше, до інтегрованих систем програмування, що поєднують текстовий редактор, компілятор і редактор зв'язків єдиним інтерфейсом.)

8. Розпізнаються всі зв'язки програми з зовнішніми об'єктами і функціями. Для зв'язування функцій і об'єктів, не визначених у процесі трансляції,

підключаються бібліотечні компоненти. У результаті виникає образ програми, що містить всю інформацію, необхідну для виконання програми (exe-модуль).

9. Замість висновків

Середовища для роботи

Веб-компілятори підходять для написання простих та невеликих програм, ідеально підходять для навчальних цілей. Їх функціонал обмежений: ви не зможете створювати виконувані файли або ефективно проводити відлагодження програм, тому краще скачати повноцінну IDE. Веб-компілятори більше підійдуть для швидкого запуску невеликих програм.

Популярні веб-компілятори:

1. **C++ Shell** http://cpp.sh/
2. **OnlineGDB** https://www.onlinegdb.com/
3. **TutorialsPoint** https://www.tutorialspoint.com/
    a. https://wandbox.org/
4. **Repl.it** - є можливість працювати із файлами.

Для прикладу можна запустити наступну програму:

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

int main() {
 cout << "Hello KH!\n";

ofstream g;
g.open("my.txt");
g<<"Hello word";
g.close();
return 0;
}
```

Популярні середовища.

1. **Visual Studio 2019 Community edition** - https://visualstudio.microsoft.com/
2. **Code::Blocks** - https://www.codeblocks.org/
3. **Jet Brains** https://www.jetbrains.com/resharper-cpp/
4. **Qt creator** - https://www.qt.io/download

Лекція 3
Оператори умови

План

1. Оператори
2. Оператор if
3. Вкладені оператори if
4. Тернарна альтернатива
5. Оператор switch
6. Вкладені оператори switch

Оператор — це частина програми, яку можна виконати окремо. Іншими словами, оператор визначає якусь дію. Оператори мови С и С++ розділяються на наступні категорії.

- Умовні оператори
- Оператори циклу
- Оператори переходу
- Мітки
- Оператора-виразів
- Блоки

До *умовного* (conditional) ставляться оператори **if** й **switch.** (Умовні оператори іноді називають *операторами розгалуження* (selection statement).) *Оператори циклу* (iteration statements) позначаються ключовими словами while, **for** й **do while.** Група операторів переходу складається з операторів **break, continue, goto** й **return.** Мітками служать оператори **case, default** (вони розглядаються в розділі "Оператор switch") і властиво мітки, які описуються в розділі "Оператор goto". Оператори-вираження — це оператори, що складаються із припустимих виражень. Блок являє собою фрагмент тексту програми, ув'язнений у фігурні дужки. Іноді блоки називають *складеними операторами* (compound statements).

Оскільки в багатьох операторах результат обчислення залежить від істинності або хибності деяких перевірок, почнемо з понять "істина" й "неправда".

Істині та хибні значення в мовах С и С++

При виконанні багатьох операторів мови С/С++ обчислюються значення умовних виразів, які мають істині або хибні значення. У мові С++ істинним уважається будь-яке ненульове значення, у тому числі негативне. Хибне значення завжди дорівнює нулю. Таке подання істинних або хибних значень дозволяє створювати надзвичайно ефективні програми.

Поряд із цим у мові С++ використовується булевий тип даних з ім'ям **bool,** що передбачає тільки два значення: **true й false.** У мові С++ число 0 автоматично перетворює в значення **false,** а будь-яке ненульове значення — у значення **true.** Справедливо й зворотне твердження: значення **false** перетвориться в 0, а **true** — в **1. З** формальної точки зору умовне вираження, що входить в умовний оператор, має тип **bool.** Однак, оскільки будь-яке ненульове значення перетвориться в значення **true,** а число 0 — у значення **false,** між мовами С и С++ щодо цього немає ніякої різниці.

Оператор if

Оператор if має такий вигляд:

if (*вираз) оператор;* [else *оператор;*]

Тут *оператор* може складатися з одного або декількох операторів або бути відсутнім зовсім (порожній оператор). Розділ else є необов'язковим.

Якщо *вираз* істинний (тобто не дорівнює нулю), виконується оператор або блок, зазначений у розділі if, у противному випадку виконується оператор або блок, передбачений у розділі else. Оператори, зазначені в розділах if або else, є взаємовиключними.

У мові С результатом умовного виразу є *скаляр,* тобто ціле число, символ, покажчик або число із плаваючою крапкою. У мові С++ до цього набору типів додається тип bool. Число із плаваючою крапкою рідко застосовується як результат умовного вираження, що входить в умовний оператор, оскільки значно сповільнює виконання програми. (Це пояснюється тим, що операції над числами із плаваючою крапкою виконуються повільніше, ніж над цілими числами або символами.)

Вкладені оператори if

Вкладеним (nested) називається оператор **if,** що перебуває усередині іншого оператора **if** або **else.** Вкладені оператори **if** зустрічаються досить часто. У вкладеному умовному операторі розділ **else** завжди пов'язаний з найближчим оператором **if,** що перебуває з ним в одному блоці й не пов'язаним з іншим оператором **else.** Розглянемо приклад.

```
if (i)
{
  if(j)   оператор1;
  if(k)   оператор2;   /*  даний  if   */
  else    оператор3;   /*  пов'язаний з даним оператором else  */
}
else оператор 4;   /*  пов'язаний з оператором if(i)   */
```

Останній розділ **else** зв'язаний не з оператором **if** (j), що перебуває в іншому блоці, а з оператором **if(i).** Внутрішній розділ **else** пов'язаний з оператором **if** (к), тому що цей оператор **if** є найближчим.

Набагато важливіше, що мова С++ допускає до 256 рівнів вкладення. Однак на практиці глибоко вкладені умовні оператори використаються вкрай рідко, оскільки це значно ускладнює логіку програми.

Тернарна альтернатива

Замість операторів **if-else** можна використати тернарний оператор "?". Загальний вид оператора **if-else** виглядає в такий спосіб.

if *(умова) вираз;* **else** *вираз;*

Однак у цьому випадку з операторами **if** й **else** зв'язані окремі вирази, а не оператори.

Оператор "?" називається *тернарним,* оскільки має три операнда. Його загальний вид такий.

Вираз1 ? Вираз2: Вираз3 Зверніть увагу на використання й місце розташування двокрапки.

Оператор "?" виконується в такий спосіб. Спочатку обчислюється *Вираз 1*. Якщо він є істинним, обчислюється *Вираз 2,* і його значення стає значенням усього тернарного оператора. Якщо *Вираз 1* є хибним, обчислюється *Вираз 3*, і результатом виконання тернарного оператора вважається саме його значення. Розглянемо приклад.

```
x = 10;
y = x>9 ? 100 : 200;
```

У цьому випадку змінній ***y*** привласнюється значення 100. Якби змінна ***x*** була менше 9, то змінна в одержала б значення 200. Цей код можна переписати за допомогою операторів **if-else.**

```
x = 10;
if(x>9) y = 100;
else y = 200;
```

Тернарний оператор можна використати замість конструкції **if-else** не тільки для присвоєння значень. Як відомо, всі функції повертають яке-небудь значення (крім функцій, що повертають значення типу void). Отже, замість виразів в операторі "?" можна використати виклики функцій. Якщо в операторі "?" зустрічається ім'я функції, вона викликається, а її результат використається замість значення відповідного виразу. Це означає, що, використовуючи виклики функцій у якості операндів тернарного оператора, можна виконати одну або кілька функцій відразу. Розглянемо приклад.

```
#include <stdio.h>

int f1(int n);
int f 2(void);
int main(void)
{
  int t;
  cout<<"Уведіть число: "; cin>>t;
  /* Вивід відповідного повідомлення */
  t ? fl(t) + f2() : cout<<"Уведений нуль.\n";
  return 0;
}
int fl(int n)
{
cout<<n;
  return 0;
}
int f2(void)
{
cout<<”уведено " ;
  return 0;
}
```

Оператор switch

У мові C/C++ передбачений оператор різноманітного розгалуження **switch,** що послідовно порівнює значення виразів зі списком цілих чисел або символьних констант. Якщо виявляється збіг, виконується оператор, пов'язаний з відповідною константою. Оператор **switch** має такий вигляд.

switch *(вираз)*
{
  case *констант1:*
    послідовність операторів

```
        break;
    case константа2:
        послідовність операторів
        break;
case константа3:
        послідовність операторів
        break;
.
.
.
default:
    послідовність операторів
}
```

Значенням *виразу* повинне бути символ або ціле число. Наприклад, вираз, результатом яких є число із плаваючою крапкою, не допускаються. Значення виразу послідовно порівнюється з константами, зазначеними в операторах **case.** Якщо виявляється збіг, виконується послідовність операторів, пов'язаних з даним оператором **case,** поки не зустрінеться оператор **break** або не буде досягнутий кінець оператора **switch.** Якщо значення виразу не збігається з жодною з констант, виконується оператор **default.** Цей розділ оператора **switch** є необов'язковим. Якщо він не передбачений, під час відсутності збігів не буде виконаний жоден оператор.

Стандарт мови C++ передбачає до 16384 операторів **case!** На практиці кількість розділів **case** в операторі **switch** варто обмежувати, оскільки воно впливає на ефективність програми. Незважаючи на те що оператор **case** є міткою, він використається тільки усередині оператора **switch.**

Оператор **break** ставиться до групи операторів переходу. Його можна використати як в операторі **switch,** так й у циклах. Коли потік керування досягає оператора **break,** програма виконує перехід до оператора, що знаходиться за оператором **switch.**

Варто знати три важливих властивості оператора **switch.**

- Оператор **switch** відрізняється від оператора if тим, що значення його виразу рівняється винятково з константами, у той час як в операторі if можна виконувати які завгодно порівняння або обчислювати будь-які логічні вираження.
- Дві константи в різних розділах **case** не можуть мати однакових значень, за винятком, коли один оператор **switch** вкладений в іншій.
- Якщо в операторі **switch** використаються символьні константи, вони автоматично перетворюють у цілочисельні.

З формальної точки зору наявність оператора **break** усередині оператора **switch** не обов'язково. Ці оператори перериває виконання послідовності операторів, пов'язаних з відповідною константою. Якщо його пропустити, будуть виконані всі наступні оператори **case,** поки не зустрінеться наступний оператор **break,** або не буде досягнутий кінець оператора **switch.** Наприклад, наведена нижче функція використає цей ефект для обробки інформації, що надходить на вхід драйвера.

```
/*  Обробка значення  */
void inp_handler(int  i)
{
int flag;
```

```
flag = -1;

switch(i) {
   case 1: /* Ці оператори case мають загальну */
   case 2: /* послідовність операторів. */
   case 3:
        flag = 0;
         break;
   case 4:
           flag = 1;
   case 5:
           error(flag);
            break;
default:
  process(i);
}
 }
```

Цей приклад ілюструє дві властивості оператора **switch**. По-перше, оператор **case** може не мати пов'язаної з ним послідовності операторів. У цьому випадку потік керування просто переходить до наступного оператора **case**, як би "провалюючись" униз. У нашому прикладі три перших оператори **case** пов'язані з однієї й тією же послідовністю операторів, а саме:

```
flag = 0;
break;
```

По-друге, якщо оператор **break** відсутній, виконується послідовність операторів, зв'язана з наступним оператором **case**. Якщо значення і дорівнює 4, змінній **flag** привласнюється число 1, і, оскільки оператора **break** наприкінці даного розділу **case** відсутній, виконання оператора **switch** триває, і викликається функція **error (flag)**. Якщо значення і дорівнює 5, функція **error** буде викликана з параметром **flag**, рівним -1, а не 1.

Те, що під час відсутності оператора **break** оператори **case** виконуються один за іншим, дозволяє уникнути непотрібного дублювання операторів і підвищити ефективність програми.

Вкладені оператори switch

Оператори switch можуть бути вкладені. Навіть якщо константи розділів **case** зовнішнього й внутрішнього операторів **switch** збігаються, проблеми не виникають. Наприклад, наведений нижче фрагмент програми є цілком прийнятним.

```
switch(x) {
     case 1:
        switch(y) {
           case 0: cout<<"Ділення на нуль."<<endl;
                    break;
          case 1: process(x,y);
   }
   break;
case 2:
.
```

.

Оперетор умовного переходу if-else

```
#include <iostream.h>
void main()
{
float a,x,y;
cin>>x>>a;
if (x>2&&x<3) y=x*a;//якщо x>2 й x<3 те y=x*a
else if (x>=3){a=3;y=x+a;}//інакше, якщо x>=3 те a=3;y=x+a
else y=a;//інакше y=a
cout<<y;
}
```

Оператор switch

```
#include <iostream.h>
void main()
{
int x;
float y;
cin>>x;
    switch (x)
    {
    case 1:y=x;break;//якщо x=1 те y=x
    case 2:y=x*x;break;//якщо x=2 те y=x*x
    case 3:y=x*x*x;break;//якщо x=3 те y=x*x*x
default: y=0;//в інших випадках y=0
    }
cout<<y;
    }
```

Тернарний оператор ?:

```
#include <iostream.h>
void main()
{
float x,y;
cin>>x;
y=(x>2||x==0)?x*x:x*x+2;//якщо x>2 або x==0 те y=x*x, інакше y=x*x+2
cout<<y;
}
```

Використана література

1. Г. Шилдт, Полный справочник по C++, 4-е видання, в-во «Вильямс», 2006
2. Х.М.Дейтел, Как програмировать на C++, 4-е видання, в-во «Бином-Пресс» 2009.
3. Р. Лафоре, Объектно-ориентированное программирование в C++, в-во «Питер», 2004, с 924.

Лекція 4
Оператори циклу
План

1. Цикл for.
2. Варіанти циклу for.
3. Нескінченний цикл.
4. Порожній цикл for.
5. Цикл while.
6. Цикл do-while.
7. Оголошення змінних в умовних операторах і циклах
8. Оператори переходу:
   - return
   - goto
   - break
   - continue

У мові C/C++, як і у всіх інших сучасних мовах програмування, оператори циклу призначені для виконання повторюваних інструкцій, поки діє певна умова. Ця умова може бути як задана заздалегідь (у циклі **for),** так і мінятися під час виконання циклу (в операторах **while** й **do-while**).

Цикл for

У тому або іншому виді цикл **for** є у всіх процедурних мовах програмування. Однак у мові C/C++ він забезпечує особливо високу гнучкість і ефективність.

Загальний вид оператора **for** такий.

for *(ініціалізація; умова; збільшення)*

Цикл **for** має багато варіантів. Однак найбільш загальна форма цього оператора працює в такий спосіб. Спочатку виконується *ініціалізація* (initialization) — оператор присвоювання, що задає початкове значення лічильника циклу. Потім перевіряється *умова* (condition), що представляє собою умовний вираз. Цикл виконується доти, поки значення цього виразу залишається істинним. *Збільшення* (increment) змінює значення лічильника циклу при черговому його виконанні. Ці розділи оператора відокремлюються один від одного крапкою з комою. Як тільки умова циклу стане хибною, програма припинить його виконання й перейде до наступного оператора.

У наступному прикладі цикл **for** виводить на екран числа від 1 до 100.

```
#include<iostream>
#include"windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int  x;
	for (x = 1; x <= 100; x++)   cout << x;

	cout << endl;
	system("pause");
	return 0;
}
```

Спочатку змінній *x* присвоюється число 1, а потім вона порівнюється із числом 100. Оскільки її значення менше 100, виконується команда cout<<. Потім змінна x збільшується на одиницю, і умова циклу перевіряється знову. Як тільки її значення перевищить число 100, виконання циклу припиниться. У цьому випадку змінна *x* є лічильником циклу, що змінюється й перевіряється на кожній ітерації.

Розглянемо приклад циклу for, тіло якого складається з декількох операторів.

```
for (x = 100; x != 65; x -= 5)
      {
            z = x * x; cout << "Квадрат числа”<<x<<”дорівнює”<<z
      }
```

Піднесення числа *x* у квадрат і виклик команди cout<< виконуються доти, поки значення змінної *x* не стане рівним 65. Зверніть увагу на те, що в цьому циклі лічильник зменшується: спочатку йому привласнюється число 100, а потім на кожній ітерації з нього віднімається число 5.

У циклі **for** перевірка умови виконується перед кожною ітерацією. Іншими словами, якщо умова циклу із самого початку є помилковою, його тіло не буде виконано жодного разу. Розглянемо приклад.

```
int y;
      int x = 10; for (y = 10; y != x; ++y)  cout<<y;

      cout << y; /*   Це  єдиний стейтмент що виконується*/
```

Цей цикл ніколи не буде виконаний, оскільки значення змінних x та y при вході в цикл рівні. Отже, умова циклу є помилковим, і ні тіло циклу, ні збільшення лічильника виконуватися не будуть. Таким чином, значення змінної в залишиться рівним 10, і саме воно буде виведено на екран.

Варіанти циклу for

Найпоширенішим є варіант, у якому використається оператор послідовного виконання ("кома"), що дозволяє застосовувати кілька лічильників циклу одночасно. Наприклад, змінні x і y є лічильниками наведеного нижче циклу. Їхня ініціалізація виконується в тому самому розділі циклу.

```
int x, y;
      for (x = 0, y = 0; x + y < 10; ++x)
      {
      y = getchar(); y = y - '0';   /*  Відняти  зі  змінної в ASCII-код нуля   */
      }
```

Як бачимо, два оператори ініціалізації розділені комою. При кожній ітерації значення змінної x збільшується на одиницю, а змінна y вводиться із клавіатури. Незважаючи на це, змінна y повинна мати якесь початкове значення, інакше перед першою ітерацією циклу умова може виявитися помилковою.

Функція converge(), наведена нижче, демонструє одночасне застосування декількох лічильників циклу. Вона копіює один рядок в інший, переміщаючись від кінців до середини.

```
#include<iostream>
#include"windows.h"
using namespace std;
void converge(char* targ, const char *src);
int main()
{
      SetConsoleCP(1251);
```

```
	SetConsoleOutputCP(1251);


		char target[80] = "XXXXXXXXXXXXXXXXXXXXXXXXXXXX"; // "Перевірка функції
converge()."
		converge(target, "Перевірка функції converge().");
		cout << "Результат:" << target << endl;


	cout << endl;
	system("pause");
	return 0;
}

/* Ця функція копіює один рядок в інший, переміщаючись від кінців до середини. */
void converge(char* targ, const char *src)
{
	int i, j;
	cout << targ << endl;
	for (i = 0, j = strlen(src); i <= j; i++, j--) {
		targ[i] = src[i];
		targ[j] = src[j];
		cout << targ << endl;
	}
}
```

Програма виводить на екран наступні рядки.

```
D:\c++ projects\Project5\x64\Debug\Project5.exe
XXXXXXXXXXXXXXXXXXXXXXXXXXXX
ПXXXXXXXXXXXXXXXXXXXXXXXXXXX
ПеXXXXXXXXXXXXXXXXXXXXXXXXXX.
ПерXXXXXXXXXXXXXXXXXXXXXXXX).
ПереXXXXXXXXXXXXXXXXXXXXXX().
ПеревXXXXXXXXXXXXXXXXXXXe().
ПеревіXXXXXXXXXXXXXXXXXge().
ПеревірXXXXXXXXXXXXXXXrge().
ПеревіркXXXXXXXXXXXXXerge().
ПеревіркаXXXXXXXXXXXverge().
Перевірка XXXXXXXXXnverge().
Перевірка фXXXXXXXonverge().
Перевірка фуXXXXXXconverge().
Перевірка фунXXXX converge().
Перевірка функXXї converge().
Перевірка функції converge().
Результат:Перевірка функції converge().

Press any key to continue . . .
```

У функції **converge** () для індексації рядка з обох кінців використаються два лічильники циклу **for** — змінні і й j. При виконанні циклу лічильник і збільшується, а лічильник j зменшується. Виконання циклу припиняється, коли значення лічильника і стає більше, ніж значення лічильника j. У цей момент всі символи рядка вже скопійовані.

Умовне вираження не обов'язково пов'язане з перевіркою лічильника циклу. Як умова циклу може використатися будь-який припустимий оператор

порівняння або логічний оператор. Це дозволяє задавати кілька умов циклу одночасно.

Розглянемо функцію, що перевіряє пароль користувача. Для уведення пароля користувач може зробити три спроби. Виконання циклу припиняється, якщо користувач вичерпав всі спроби або ввів правильний пароль.

```
void sign_on()
{
	char str[20];
	int x;
	for (x = 0; x < 3 && strcmp(str, "пароль"); ++x) {
		cout << "Уведіть пароль: ";
		gets_s(str);
	}
	if (x == 3)  return;
	/*  Інакше користувач  одержує доступ до системи . . . */
    }
```

У цій функції використовується стандартна функція **strcmp()**, що порівнює два рядки й повертає нуль, якщо вони збігаються.

Нагадаємо, що кожний із трьох розділів циклу **for** може складатися з будь-яких припустимих виразів. Ці вирази можуть бути ніяк не пов'язані із призначенням розділів. З огляду на вищесказане, розглянемо наступний приклад.

```
#include<iostream>
#include"windows.h"
using namespace std;
int sqrnum(int num);
int readnum(void);
int prompt(void);

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int t;
	for (prompt(); t = readnum(); prompt())
		sqrnum(t);

	cout << endl;
	system("pause");
	return 0;
}

int prompt(void)
{
	cout<<"Уведіть число: ";
	return 0;
}
int readnum(void)
{
	int t;

	cin>>t;
	return t;
}
int sqrnum(int num)
{
	cout<<num * num<<endl;
	return num * num;
}
```

Зверніть увагу на цикл **for** у функції **main()**. Кожний з його розділів містить виклик функції, у якій користувачеві пропонується ввести із клавіатури якесь число. Якщо уведено число 0, виконання циклу припиняється, оскільки умовне

вираження стає помилковим. У протилежному випадку число зводиться у квадрат. Таким чином, даний цикл **for** використовує розділи ініціалізації й збільшення вкрай незвичайно, при цьому й із синтаксичної, і з семантичної точки зору цикл є абсолютно правильним.

Інша цікава особливість циклу **for** полягає в тім, що його розділи можна пропускати. Кожний з його розділів є необов'язковим. Наприклад, цикл, наведений нижче, виконується доти, поки користувач не введе число **123:**

```
for (x = 0; x != 123; )   cin>>x;
```

Зверніть увагу на те, що розділ збільшення лічильника в даному циклі **for** відсутній. Це значить, що при кожній ітерації значення змінної *x* рівняється із числом **123,** і ніякі дії з нею більше не виконуються. Однак, якщо користувач уведе із клавіатури число **123,** умова циклу стане помилковим, і програма припинить його виконання.

Лічильник можна ініціалізовувати за межами циклу **for.** Цим способом користуються, коли початкове значення лічильника є результатом складних обчислень, як у наступному прикладі.

```
gets(s);  /* Вважати рядок як змінну s */
if (*s) x = strlen(s); /* Обчислити довжину рядка */
else x = 10;

for (; x < 10; ) {
      cout<<x;
      ++x;
    }
```

Тут розділ ініціалізації залишений порожнім, а змінна *x* ініціалізується до входу в цикл.

Нескінченний цикл

Хоча в якості нескінченного можна використати будь-який цикл, традиційно для цієї мети застосовується оператор **for.** Оскільки всі розділи оператора **for** є необов'язковими, його легко зробити нескінченним, не задавши ніякого умовного виразу.

for(; ;) printf("Цей цикл виконується нескінченно.\n");

```
for (; ; )   cout<<"Цей  цикл  виконується  нескінченно."<<endl;
```

Якщо умовний вираз не зазначений, він вважається істинним. Зрозуміло, у цьому випадку можна як і раніше виконувати ініціалізацію й збільшення лічильника, однак програмісти мовою C++ як нескінченний цикл найчастіше використають конструкцію **for** (; ;).

Насправді конструкція **for(;;)** не гарантує нескінченне виконання циклу, оскільки його тіло може містити оператор **break,** що приводить до негайного виходу. (Ми детально вивчимо цей оператор небагато пізніше.) У цьому випадку програма передасть керування наступному операторові, що перебуває за межами тіла циклу **for,** як показано нижче.

```
ch = '\0';
for (; ; ) {
      ch = getchar(f);   /*  Ввести символ  */
      if (ch == 'A')   break;   /*  Вихід із циклу  */
}
```

```
cout<<"Ви  ввели  букву A";
```

Цей цикл виконується доти, поки користувач не введе із клавіатури букву А.

Порожній цикл for

Оператор може бути порожнім. Це значить, що тіло циклу **for** (як і будь-якого іншого циклу) може не містити жодного оператора. Цей факт можна використати для підвищення ефективності деяких алгоритмів і затримки виконання програми.

Видалення пробілів із вхідного потоку – одне з найпоширеніших завдань. Наприклад, система керування базою даних може допускати запит "показати всі рахунки, залишок на яких менше 400". База даних розпізнає кожне слово окремо, не з огляду на пробіли. Отже, вона розпізнає слово "показати", але не зрозуміє слова " показати". Таким чином, пробіли в рядку запиту необхідно ігнорувати. Це завдання вирішує цикл **for**, що пропускає всі пробіли, що коштують перед словами в рядку str.

```
for (; *str == '   '; str++)  ;
```

Як бачимо, цей цикл не має тіла – воно йому не потрібно.

Цикли часто використаються для затримки виконання програми. Нижче показано, як цього можна досягти, використовуючи оператор **for**.

```
for (t = 0; t < SOME_VALUE; t++);
```

Цикл while

Другий по значимості цикл у мові C/C++ - оператор **while**. Він має такий вигляд.

while *(умова) оператор;*

Тут *оператор* може бути порожнім, окремим оператором або блоком операторів. *Умова* може задаватися будь-яким виразом. Умова циклу вважається істиною, якщо значення цього виразу не дорівнює нулю. Як тільки умова циклу стає хибною, програма передає керування операторові, що знаходиться відразу після оператора while. Розглянемо приклад функції, що обробляє уведення із клавіатури, що терпляче чекає, поки користувач не введе букву А.

```
char wait_for_char()
{
      char ch;

      ch = ' \0';      /*   початкове  значення  змінної  ch  */
      while (ch != 'A')   ch = getchar(t);
      return  ch;
}
```

Спочатку змінній **ch** привласнюється нуль. Оскільки вона є локальною, її значення поза функцією **wait_for__char()** не визначено. Потім цикл **while** перевіряє, чи не дорівнює значення змінної **ch** символу А. Оскільки початкове значення змінної **ch** дорівнює нулю, умова циклу є щирим, і його виконання триває. Умова циклу перевіряється щораз, коли користувач уводить символ із клавіатури. Як тільки він уведе букву А, умова циклу стане помилковим, і програма припинить його виконання.

Як і цикл **for**, цикл **while** перевіряє умову перед початком виконання тіла. Отже, якщо умова циклу із самого початку є хибною, його тіло не буде виконано

жодного разу. Завдяки цій властивості немає необхідності окремо перевіряти умову циклу перед входом у нього. Цю особливість добре ілюструє функція **pad()**, що додає пробіли в кінець рядка, поки її довжина не стане рівною заданій. Якщо довжина рядка вже дорівнює необхідній величині, пробіли не додаються.

```
#include<iostream>
#include"windows.h"
using namespace std;
void pad(char* s, int length);

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	char str[80];
	strcpy(str, "Перевірка");
	pad(str, 40);
	cout<<strlen(str);


	cout << endl;
	system("pause");
	return 0;
}

/* Додає пробіли в кінець рядка. */
void pad(char* s, int length)
{
	int l;
	l = strlen(s); /* Обчислюємо довжину рядка */
	while (l < length) {
		s[l] = ' '; /* Вставляємо пробіл */
		l++;
	}
	s[l] = '\0';   /*   Рядок   повинна   завершуватися   нульовим   байтом   */
}
```

Функція **pad()** має два аргументи - покажчик **s** на довжину рядка і цілочисельну змінну **length**, що задає необхідну довжину рядка. Якщо вихідна довжина рядка дорівнює числу **length** або перевищує його, то тіло циклу **while** не виконується. Якщо ж довжина рядка, на яку посилається покажчик **s**, менше необхідної, функція **pad()** додає потрібну кількість пробілів. Довжина рядка обчислюється за допомогою стандартної функції **strlen()**.

Якщо вихід із циклу **while** залежить від декількох умов, звичайно як умова циклу використають окрему змінну, значення якого змінюється декількома операторами в різних місцях циклу. Розглянемо приклад.

```
void func1()
{
	int working;

	working = 1; /* тобто, істина */

	while (working) {
		working = processl();
		if (working)
			working = process2();
		if (working)
			working = process3();
	}
    }
```

Кожна функція, викликана в цьому фрагменті програми, може повернути помилкове значення й привести до виходу із циклу.

Тіло циклу **while** може бути порожнім. Наприклад, наведений нижче порожній цикл виконується доти, поки користувач не введе букву A.

```
while((ch=getchar())  !=  'A')  ;
```

Оператор присвоювання усередині умовного вираження циклу **while** виглядає незвично, але все стане на свої місця, якщо згадати, що його значенням є значення операнда, розташованого в правій частині.

Цикл do-while

На відміну від операторів **for** й **while**, які перевіряють умови на початку циклу, оператор **do-while** робить це наприкінці. Іншими словами, цикл **do-while** виконується принаймні один раз. Загальний вид оператора **do-while** такий.

```
do  {
оператор;
} while(умова);
```

Якщо тіло циклу складається лише з одного оператора, фігурні дужки не обов'язкові, хоча вони роблять цей цикл зрозуміліше (програмістові, але не компіляторові). Цикл **do-while** повторюється доти, поки *умова* не стане помилковим.

Розглянемо приклад, у якому цикл **do-while** зчитує числа із клавіатури, поки вони не перевищать 100.

```
do {
      cin>>num;
    } while (num > 100);
```

Найчастіше оператор **do-while** застосовується для вибору пунктів меню. Коли користувач вводить припустиме значення, воно повертається відповідною функцією. Неприпустимі значення ігноруються. Розглянемо поліпшену версію програми для перевірки правопису.

```
void menu(void)
{
       char ch;
       cout<<"1.   Перевірка правопису"<<endl;
       cout << "2.   Виправлення помилок" << endl;
       cout << "3.   Вивід помилок " << endl;
       cout << "      Виберіть пункт меню: ";

       do {
              ch = getchar(f); /* Зчитуємо символ із клавіатури */
              switch (ch) {
              case '1':
                     check_spelling();
                     break;
              case '2':
                     correct_errors();
                     break;
              case '3':
                     display_errors();
                     break;
```

```
        }
    } while (ch != 'l' && ch != '2' && ch != '3');
}
```

Тут оператор **do-while** дуже зручний, оскільки функція повинна вивести меню принаймні один раз. Якщо користувач увів припустиме значення, програма продовжить виконувати цикл.

Оголошення змінних в умовних операторах і циклах

У мові C++ (але не в стандарті C89) можна повідомляти змінні усередині умовних виражень, що входять в оператори **if** або **switch**, усередині умови циклу **while**, а також у розділі ініціалізації циклу **for**. Область видимості змінної, оголошеної в одному із цих трьох місць, обмежена блоком, що ставиться до даного оператора. Наприклад, змінна, оголошена усередині циклу **for**, є локальною стосовно нього.

Розглянемо приклад, у якому змінна оголошена в розділі ініціалізації циклу **for**.

/* Змінна i є локальною стосовно циклу for, а змінна j існує як усередині, так і поза циклом. */

```
int  j;
for (int i = 0; i < 10; i++)
    j = i * i;
/* i  =  10;  //  ***  Помилка *** -  змінна  i  тут  невидима! */
```

У цьому прикладі змінна і оголошена усередині циклу **for** і використається як лічильник. Поза циклом **for** ця змінна не існує!

Оскільки лічильник потрібний лише усередині циклу, його оголошення в розділі ініціалізації оператора **for** стало загальноприйнятою практикою.

Якщо ваш компілятор повністю підтримує стандарт мови C++, то змінні можна повідомляти не тільки усередині циклів, але й усередині звичайних умовних виражень. Розглянемо приклад.

```
if (int x = 20) {
    x = x - y;
    if (x > 10)   y = 0;
}
```

Тут оголошується змінна x, якій привласнюється число 20. Оскільки це значення вважається істинним, виконується блок, пов'язаний з оператором **if**. Область видимості змінної, оголошеної усередині умовного вираження, обмежується цим блоком. Таким чином, поза оператором **if** змінної *x* не існує. Чесно говорячи, не всі програмісти вважають прийнятним такий спосіб оголошення змінних, тому ми не будемо його застосовувати.

Оператори переходу

У мові C/C++ передбачені чотири оператори безумовного переходу: **return, goto, break, continue.** Оператори **return** й **goto** можна застосовувати в будь-якому місці програми, у той час як оператори **break** й **continue** пов'язані з

операторами циклів. Крім того, оператор **break** можна застосовувати усередині оператора **switch**.

Оператор return

Цей оператор використається для повернення керування з функції. Він ставиться до операторів безумовного переходу (jump operators), оскільки виконує повернення в точку виклику функції. З ним може бути зв'язане певне значення, хоча це й не обов'язково. Якщо оператор **return** пов'язаний з певним значенням, воно стає результатом функції. У стандарті С89 всі функції, що повертають значення, формально не зобов'язані були містити оператор **return.** Якщо програміст забував указати його, функція повертала якесь випадкове значення, так зване "сміття". Однак у мові С++ (і в стандарті С99) всі функції, що повертають значення, *зобов'язані* містити оператор **return.** Іншими словами, якщо функція в мові С++ повертає якесь значення, будь-який оператор **return,** що з'являється в її тілі, повинен бути пов'язаний з якимсь значенням.

Оператор **return** має такий вигляд.

return *вираз;*

Вираз вказується лише тоді, коли у відповідності зі своїм оголошенням функція повертає якесь значення. У цьому випадку результатом функції є значення даного *виразу*.

Усередині функції можна використати скільки завгодно операторів **return.** Однак функція припинить свої обчислення, як тільки досягне першого оператора **return.** Закриваюча фігурна дужка, що обмежує тіло функції, також приводить до припинення її виконання. Вона інтерпретується як оператор **return,** не зв'язаний ні з яким значенням. Якщо програміст не вкаже оператор **return** у функції, що повертає якесь значення, то її результат залишиться невизначеним.

Функція, визначена специфікатором **void,** може не містити жодного оператора **return,** пов'язаного з яким-небудь значенням. Оскільки такі функції по визначенню не мають значень, що повертають, безглуздо зв'язувати з ними оператор **return.**

Оператор goto

Оскільки в мові С/С++ існує багатий вибір керуючих структур на основі операторів **break** й **continue,** в операторі **goto** немає особливої необхідності. Вважається, що використання операторів **goto** знижує читабельність програм. Проте, незважаючи на те, що оператор **goto** протягом багатьох років піддається критиці, він як і раніше іноді застосовується. Важко уявити собі ситуацію, до якої не можна було б обійтися без оператора **goto.** І все-таки у вмілих руках і при обережному використанні оператор **goto** може принести користь, наприклад, при виході із глибоко вкладених циклів.

Для оператора **goto** необхідна *мітка* (label), що представляє собою ідентифікатор із двокрапкою. Мітка повинна перебувати в тій же функції, що й оператор goto, - перестрибувати з функції у функцію не можна. Оператор goto має такий вигляд.

goto *мітка:*

.

.

мітка;

Мітка може перебувати як до, так і після оператора goto. Наприклад, використовуючи оператор goto і мітку, можна організувати цикл від 1 до 100.

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

Оператор break

Оператор **break** застосовується у двох ситуаціях. По-перше, він використовується для припинення виконання **case** усередині оператора **switch**. По-друге, за допомогою оператора **break** можна негайно вийти із циклу незалежно від істинності або хибності його умови.

Якщо в циклі зустрічається оператор **break**, ітерації припиняються й виконання програми відновляється з оператора, що випливає за оператором циклу. Розглянемо приклад.

```
#include<iostream>
#include"windows.h"
using namespace std;
void pad(char* s, int length);

int main()
{
      SetConsoleCP(1251);
      SetConsoleOutputCP(1251);

      int t;
      for (t = 0; t < 100; t++) {
            cout<< t;
            if (t == 10) break;
      }



      cout << endl;
      system("pause");
      return 0;
   }
```

Ця програма виводить на екран числа від 0 до 10, а потім цикл припиняється, оскільки виконується оператор **break**. Умова t < 100 при цьому ігнорується.

Оператор **break** часто застосовується в циклах, виконання яких варто негайно припинити при настанні певної події. У наведеному нижче прикладі натискання клавіші припиняє виконання програми:

```
#include<iostream>

#include"windows.h"
using namespace std;


int main()
{
      SetConsoleCP(1251);
      SetConsoleOutputCP(1251);

      do {
            /* пошук імен ... */
            if (getchar()) break;
      } while (1);
      /* точка виходу */
```

```
    cout << endl;
    system("pause");
    return 0;
}
```

Якщо клавіша натиснута, функція `getchar()` повертає код клавіши, і в такому випадку умова буде істина.

Якщо цикли вкладені один у одного, оператор **break** виконує вихід із внутрішнього циклу в зовнішній. Наприклад, наведена нижче програма 100 разів виводить на екран числа від 1 до 10, причому щораз, коли лічильник досягає значення 10, оператор **break** передає керування зовнішньому циклу **for**.

```
for (t = 0; t < 100; ++t) {
            count = 1;
            for (;;) {
                  cout<<count;
                  count++;
                  if (count == 10) break;
            }
    }
```

Якщо оператор **break** утримується усередині оператора **switch**, що вкладений у якийсь цикл, то вихід буде здійснений тільки з оператора **switch**, а керування залишиться в зовнішньому циклі.

Функція exit

Хоча функція **exit()** не ставиться до керуючих операторів, прийшов час її вивчити. Виклик стандартної бібліотечної функції **exit()** приводить до припинення роботи програми й передачі керування операційній системі. Її ефект можна зрівняти з катапультуванням із програми.

Функція **exit()** виглядає в такий спосіб.

void exit(int *код_повернення);*

Значення змінної *код_повернення* передається процесу, у ролі якого частіше за все виступає операційна система. Нульове значення коду повернення відповідає нормальному завершенню роботи. Інші значення аргументу вказують на вид помилки. Як код повернення можна застосовувати макроси **EXIT_SUCCESS** й **EXIT_FAILURE**. Для виклику функції **exit()** необхідний заголовний файл **stdlib.h**. У програмах мовою C++ можна також використати заголовний файл **<cstdlib>**.

Функція **exit()** часто використовується, коли обов'язкова умова, що гарантує правильну роботу програми, не виконується. Розглянемо, наприклад, віртуальну комп'ютерну гру, для якої потрібно спеціальний графічний адаптер. Функція **main()** у цій програмі може виглядати в такий спосіб.

```
if (!virtual_graphics()) exit(1);
      play;
      /* ... */
```

Тут функція **virtual_graphics ()** визначається користувачем і повертає істинне значення, коли в комп'ютері є необхідний графічний адаптер. Якщо ж його нема, вона повертає помилкове значення, і функція **exit()** припиняє роботу програми.

У програмі для перевірки правопису функція **menu()** може використати функцію **exit()** для виходу із програми й повернення керування операційній системі.

```
char ch;
      cout << "1.   Перевірка правопису" << endl;
```

```
	cout << "2.   Виправлення помилок" << endl;
	cout << "3.   Вивід помилок " << endl;
	cout << "4.   Вихід" << endl;
	cout << "      Виберіть пункт меню: ";

	do {
		ch = getchar(); /* Введення символу із клавіатури */
		switch (ch) {
		case '1':
			check_spelling();
			break;
		case ' 2 ':
			correct_errors();
			break;
		case '3':
			display_errors();
			break;
		case '4':
			exit(0); /* Повернення в операційну систему */
		}
	} while (ch != 'l' && ch != '2' && ch!='3');
```

Оператор continue

Оператор **continue** нагадує оператор **break**. Вони розрізняються тим, що оператор **break** припиняє виконання всього циклу, а оператор **continue** - лише його поточної ітерації, викликаючи перехід до наступної ітерації й пропускаючи всі оператори, що залишилися, у тілі циклу. У циклі **for** оператор **continue** викликає перевірку умови й збільшення лічильника циклу. У циклах **while** й **do-while** оператор **continue** передає керування операторам, що входять в умову циклу. Наведена нижче програма підраховує кількість пробілів у рядку, уведеної користувачем.

```
#include<iostream>
#include"windows.h"
using namespace std;


int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	char s[80], * str;
	int space;

	cout<<"Введіть рядок: ";
	gets_s(s);
	str = s;

	for (space = 0; *str; str++) {
		if (*str != ' ') continue;
		space++;
	}
	cout<<"пробілів"<<space<<endl;

	cout << endl;
	system("pause");
	return 0;
    }
```

Перевіряється кожен символ рядка. Якщо він не є пробілом, оператор **continue** перериває поточну ітерацію циклу **for** і починає нову, у протилежному випадку значення лічильника **space** збільшується на 1.

У наступному прикладі оператор **continue** виконує вихід із циклу **while,** передаючи керування операторові, що входить в умову циклу.

```
void code(void)
{
      char  done, ch;
      done = 0;
      while (!done) {
            ch = getchar();
            if (ch == '$') {
                  done = 1;
                  continue;
            }
            putchar(ch + 1);   /*   Перейти   до   наступної   букви   алфавіту   */
      }
   }
```

Функція **code** кодує повідомлення, додаючи одиницю до коду кожного символу. Наприклад, у повідомлені англійською мовою буква A замінюється буквою B и т. Функція припиняє свою роботу, якщо користувач увів символ $. Після цього вивід повідомлень на екран припиняється, оскільки змінна **done** приймає щире значення, і, відповідно, умова циклу стає помилковим.

Використана література

1. Г. Шилдт, Полный справочник по C++, 4-е видання, в-во «Вильямс», 2006
2. Х.М.Дейтел, Как програмировать на C++, 4-е видання, в-во «Бином-Пресс» 2009.
3. Р. Лафоре, Объектно-ориентированное программирование в C++, в-во «Питер», 2004, с 924.

Масиви
План

1. Одновимірні масиви.
2. Створення вказівника на масив.
3. Передача одновимірного масиву у функцію.
4. Двовимірні масиви
5. Багатовимірні масиви
6. Індексація вказівників
7. Ініціалізація масиву

Масив (array) — це сукупність змінних, що мають однаковий тип й об'єднаних під одним ім'ям. Доступ до окремого елемента масиву здійснюється за допомогою індексу. Відповідно до правил мови C/C++ всі масиви складаються із суміжних комірок пам'яті. Молодша адреса відповідає першому елементу масиву, а старший - останньому. Масиви можуть бути одновимірними й багатовимірними. Найпоширенішим масивом є рядок, що завершується нульовим байтом. Вона являє собою звичайний масив символів, останнім елементом якого є нульовий байт.

Масиви й вказівники тісно зв'язані між собою. Важко описувати масиви, не згадуючи вказівники, і навпаки.

Одновимірні масиви

Оголошення одновимірного масиву виглядає в такий спосіб.

тип ім'я_змінної[розмір]

Як й інший змінній, масив повинен оголошуватися явно, щоб компілятор міг виділити пам'ять для нього. Тут *тип* повідомляє базовий тип масиву, тобто тип його елементів, а *розмір* визначає, скільки елементів утримується в масиві. От як виглядає оголошення масиву з ім'ям **balance,** що має тип **double** і складається з **100** елементів.

double balance[100] ;

Доступ до елемента масиву здійснюється за допомогою імені масиву й індексу. Для цього індекс елемента вказується у квадратних дужках після імені масиву. Наприклад, оператор, наведений нижче, привласнює третьому елементу масиву **balance** значення 12.23.

balance[3] = 12.23;

Індекс першого елемента будь-якого масиву в мові C/C++ дорівнює нулю. Отже, оператор

char p[10];

повідомляє масив символів, що складає з 10 елементів — від **p[0]** до **p[9].** Наступна програма заповнює цілочисельний масив числами від 0 до 6.

```
#include <iostream>
#include "windows.h"
using namespace std;

int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int ABC[6];
	for (int i = 0; i < 6; i++)
```

```
        ABC[i] = i;

 for (int i = 0; i < 6; i++)
 {
        cout << "Елемент ABC[" << i << "]=";
        cout << ABC[i] << endl;
 }

 cout << endl;
 system("pause");
 return 0;
}
```

Обсяг пам'яті, необхідний для зберігання масиву, залежить від його типу й розміру. Розмір одновимірного масиву в байтах обчислюється по формулі:

кількість_байтів = **sizeof***(базовий_тип)* * *кількість_елементів*

У мові C/C++ не передбачена перевірка виходу індексу масиву за межі припустимого діапазону. Іншими словами, під час виконання програми можна помилково вийти за межі пам'яті, відведеної для масиву, і записати дані в сусідні осередки, у яких можуть зберігатися інші змінні й навіть програмний код. Відповідальність за запобігання подібних помилок лежить на програмістові. Наприклад, фрагмент програми, наведений нижче, буде скомпільований без помилок, однак під час виконання програми індекс масиву вийде за межі припустимого діапазону.

```
int  count[10],    i;
/*  Вихід індексу масиву  за межі припустимого діапазону  */
for(i=0;  i<100;  i++)  count[i]  =  i;
```

Власне кажучи, одновимірний масив являє собою список змінних, що мають однаковий тип і зберігаються в суміжних комірках пам'яті в порядку зростання їхніх індексів. На мал. 1 показано, як зберігається в пам'яті масив a, що починається з адреси 1000 й оголошений за допомогою оператора

```
char a [7];
```

Елемент	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]
Адреса	1000	1001	1002	1003	1004	1005	1006

Рис. 1. Масив із семи символів, що починається з адреси 1000

Створення вказівника на масив

Ім'я масиву є вказівником на перший його елемент. Допустимо, масив оголошений за допомогою оператора

```
 int  sample [10];
```

Вказівником на його перший елемент при цьому є ім'я **sample**. Таким чином, у наступному фрагменті вказівнику привласнюється адреса першого елемента масиву **sample**.

```
int   *p;
int  sample[10] ;
p  =   sample;
```

Адресу першого елемента масиву можна також обчислити за допомогою оператора &. Наприклад, вираження **sample** й **&sample[0]** еквівалентні. Однак у професійно написаних програмах мовою C/C++ ви ніколи не зустрінете вираження **&sample[0].**

Передача одновимірного масиву у функцію

У мові C/C++ весь масив не можна передати як аргумент функції. Замість цього можна передати вказівник на масив, тобто ім'я масиву без індексів. Наведений нижче фрагмент програми передає у функцію **func1()** індекс **i**.

```
int main(void)
{
   int   i[10];
   func1(i) ;
   .
   .
}
```

Якщо аргументом функції є одновимірний масив, її формальний параметр можна оголосити трьома способами: як вказівник, як масив фіксованого розміру і як масив невизначеного розміру. Наприклад, щоб надати функції **func1()** доступ до масиву **i**, можна використати три варіанти. По-перше, у якості її аргументу можна оголосити вказівник.

```
void fund(int  *x)   /*  Вказівник  */
{
   .
   .
   .
}
```

По-друге, можна передати вказівник на масив фіксованого розміру.

```
void  fund(int x[10])   /* Масив фіксованого розміру  */
{
   .
   .
   .
}
```

І, нарешті, можна використати масив невизначеного розміру.

```
void  func1(int x[ ])   /*  Масив невизначеного розміру  */
{
   .
   .
   .
}
```

Ці три оголошення еквівалентні один одному, оскільки їхній зміст зовсім однаковий: у функцію передається вказівник на цілочисельну змінну. В першому випадку дійсно використовується вказівник. У другому застосовується стандартне оголошення масиву. В останньому варіанті оголошується, що у функцію буде переданий цілочисельний масив невизначеної довжини. Розмір масиву, переданого у функцію, не має ніякого значення, оскільки перевірка виходу індексу за межі припустимого діапазону в мові C/C++ не передбачена. При вказівці розміру масиву можна написати будь-яке число - це нічого не змінить, оскільки у функцію буде переданий не масив, що складається з 32 елементів, а лише вказівник на його перший елемент.

```
void func1(int x[32])   /*  Масив фіксованого розміру  */
```

{

.

.

.

}

Двовимірні масиви

У мові C/C++ передбачені багатовимірні масиви. Найпростішим з них є двовимірний. Власне кажучи, двовимірний масив - це масив одновимірних масивів. Оголошення двовимірного масиву **d**, що складає з 10 рядків й 20 стовпців, виглядає в такий спосіб.

int d[10][20];

Повідомляючи двовимірні масиви, варто проявляти обережність. У деяких мовах програмування розмірності масивів відокремлюються комами. На відміну від них, у мові C/C++ розмірності масиву беруться у квадратні дужки. Звертання до елемента двовимірного масиву виглядає так:

d[l][2]

Урахуйте також, що відношення "більше" й "менше" між рядками розуміються в лексикографічному змісті. Наприклад, значення **stremp**("А","Я") дорівнює —31 (тобто рядок "А" менше рядка "Я"), а значення **stremp("Я","А")** дорівнює 31. Зверніть увагу на те, що значення **stremp** (**"АЯ"**, **"ЯА"**) знову дорівнює —31, іншими словами, функція **stremp** повертає різниці між ASCII-кодами перших незбіжних між собою символів. Властиво, саме так упорядковуються слова за абеткою.

Наступна програма заповнює двовимірний масив числами від 1 до 19 і виводить їх на екран по рядках.

```
#include <iostream>
using namespace std;
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;
      /*   Вивід  на  екран   */
      for (t = 0; t < 3; ++t) {
            for (i = 0; i < 4; ++i)
                  cout << num[t][i] << "\t";
            cout<<endl;
      }
      return 0;
}
```

У даному прикладі елемент **num[0] [0]** дорівнює **1, num[0] [l]** дорівнює 2, **num[0] [2]** дорівнює 3 і т.д. Значення елемента **num[2] [3]** дорівнює 12. Масив **num** можна зобразити в такий спосіб.

num[t][i]

	0	**1**	**2**	**3**
0	**1**	**2**	**3**	**4**
1	**5**	**6**	**7**	**8**
2	**8**	**10**	**11**	**12**

Двомірний масив зберігається у вигляді матриці, у якій перший індекс задає номер рядка, а другий - номер стовпця. Таким чином, при обході елементів у

порядку їхнього розміщення в пам'яті правий індекс змінюється швидше, ніж лівий. Графічна схема розміщення двомірного масиву в пам'яті показана па мал. 4.2.

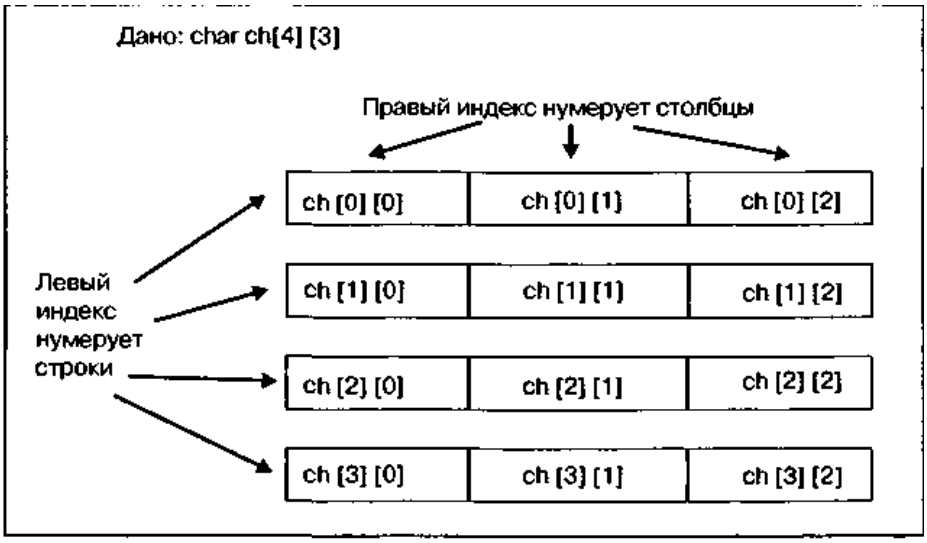


Рис. 4.2. Двухмерный массив

Обсяг пам'яті, займаний двовимірним масивом, виражений у байтах, задається наступною формулою:

*кількість байтів=кількість рядків**
*кількість_стовпців****sizeof****(базовий_тип)*

Вважаючи, що ціле число займає 4 байт, можна обчислити, що для зберігання масиву, що складає з 10 рядків й 5 стовпців, необхідно 10*5*4=200 байт.

Якщо двовимірний масив використовується як аргумент функції, то в неї передається тільки вказівник на його перший елемент. Однак при цьому необхідно вказати, принаймні, кількість стовпців. (Можна, зрозуміло, задати й кількість рядків, але це не обов'язково.) Кількість стовпців необхідно компіляторові для того, щоб правильно обчислити адресу елемента масиву усередині функції, а для цього повинна бути відома довжина рядка. Наприклад, функція, що одержує як аргумент двовимірний масив, що складається з 10 рядків й 10 стовпців, може виглядати так.

```
void func1(int x[ ][10])
{
   .
   .
   .
}
```

Компілятор повинен знати кількість стовпців, інакше він не зможе правильно обчислювати вираження, подібні наступному:

```
x[2][4]
```

Якби довжина рядка була невідома, компілятор не знайшов би початок третього рядка.

Розглянемо коротку програму, у якій двовимірний масив використається для зберігання оцінок, отриманих студентами. Передбачається, що вчитель викладає в трьох групах, у яких учаться не більше 30 студентів. Зверніть увагу на те, як відбувається обхід у масиві **grade** у кожній функції.

```
     /*  Проста база даних, що   містить оцінки студентів.   */
#include <iostream>
using namespace std;
#define CLASSES     3
#define  GRADES     30

int  grade[CLASSES][GRADES];
void enter_grades(void);
int  get_grade(int  num);
void disp_grades(int  g[][GRADES]);

int main()
{
      char  ch, str[80];
      for (;;) {
            do {
                  cout<<"(В)від оцінок"<<endl;
                  cout<<"(Д)рук оцінок"<<endl;
                  cout<<"(К)інець"<<endl;
                  gets_s(str);
                  ch = toupper(*str); //повертає велику букву з рядкового типу в
символьний
            } while (ch != 'В' && ch != 'Д' && ch != 'К');
            switch (ch) {
            case 'В':
                  enter_grades();
                  break;
            case 'D':
                  disp_grades(grade);
                  break;
            case 'К':
                  exit(0);
            }
      }
      return 0;
}
/* Введення оцінок. */
void enter_grades()
{
      int t, i;
      for (t = 0; t < CLASSES; t++) {
            cout << "Class №: " << t + 1 << endl;
            for (i = 0; i < GRADES; ++i)
                  grade[t][i] = get_grade(i);
            }
}
/* Зчитування оцінки. */
int get_grade(int num)
{
      char s[80];
      cout<<"Введіть оцінку студента № : "<<num + 1<<endl;
      gets_s(s);
      return(atoi(s)); // atoi – переводить число в рядковому форматі і інтовий
}
/* Вивід оцінок. */
void disp_grades(int g[][GRADES])
{
      int t, i;
      for (t = 0; t < CLASSES; ++t) {
            cout<<"Група № : "<< t + 1<<endl;
            for (i = 0; i < GRADES; ++i)
                  cout<<"Студент №"<< i + 1<<" is "<< g[t][i]<<endl;
      }
     }
```

Багатовимірні масиви

У мові С/С++ масиви можуть мати більше двох розмірностей. Максимально припустима кількість розмірностей задасться компілятором. Загальний вид оголошення багатовимірного масиву такий.

тип ім'я[*Розмір1*] [*Розмір2*][*Розмір3*]... [*Розмір4*]

Масиви, що мають більше трьох розмірностей, використовуються рідко, оскільки вони займають занадто великий обсяг пам'яті. Наприклад, чотиривимірний масив символів розмірністю 10x6x9x4 займає 2160 байт. Якби масив містив двохбітові цілі числа, треба було б 4320 байт. Якби елементи масиву мали тип **double** і займали б 8 байт кожний, то для масиву треба було б 17280 байт. Розмір пам'яті, виділеної для масиву, експоненціально зростає зі збільшенням кількості розмірностей, наприклад, якщо до попереднього масиву додати п'ятий вимір, у якому розташовується 10 елементів, то для нього знадобився б 172800 байт.

При звертанні до багатовимірних масивів компілятор обчислює кожен індекс. Отже, доступ до елементів багатовимірного масиву відбувається значно повільніше, ніж до елементів одновимірного масиву.

Передаючи багатовимірний масив у функції, варто вказувати всі розміри, крім першого. Наприклад, якщо масив **m** оголошений оператором

```
int  m[4][3][6][5];
```

то функція **func1(),** що одержує його як аргумент, може виглядати так:

```
void func1 (int d[ ][3][6][5])
{
.
.
.
}
```

Зрозуміло, при бажанні можна вказати й перший розмір, але це не обов'язково

Ініціалізація масиву

У мові С/С++ допускається ініціалізація масивів при їхньому оголошенні. Загальний вид ініціалізації масиву не відрізняється від ініціалізації звичайних змінних.

тип_масиву ім'я_масиву[*розмір1*]...[*розмірN*] = *{список_значень}*

Список значень являє собою список констант, розділених комами. Тип констант повинен бути сумісним з *типом масиву.* Перша константа присвоюється першому елементу масиву, друга - другому й т.д. Зверніть увагу на те, що після фігурної дужки, що закривається, } обов'язково повинна стояти крапка з комою.

Розглянемо приклад, у якому цілочисельний масив, що складається з 10 елементів, ініціалізується числами від 1 до 10.

```
int  i[10]  =  {1,  2,  3,  4,  5,  б,  7,  8,  9,  10};
```

Тут елементу **i** [0] привласнюється значення 1, а елементу **i** [9] — число 10. Символьні масиви можна ініціалізувати рядковими константами:

char *ім'я масиву[розмір]=" рядок";*

Багатовимірні масиви ініціалізуються аналогічно. Розглянемо приклад ініціалізації масиву **str** числами від 1 до 10 й їхніми квадратами.

```
int  sqrs[10][2]  =  {
```

```
1,   1,
2,   4,
3,   9,
4,   16,
5,   25,
6,   36,
7,   49,
8,   64,
9,   81,
10,   100
};
```

Ініціалізуючи багатовимірний масив, можна брати елементи списку у фігурні дужки. Цей спосіб називається *субагрегатним групуванням* (subaggregate grouping). Наприклад попередню ініціалізацію можна переписати наступним чином.

```
int  sqrs[10][2]  =  {
{1,  1},
{2,  4},
{3,  9},
{4,  16},
{5,  25;
{6,  36}.
{7,  49}
{8,  64},
{9,  81},
{10,  100}
};
```

Якщо при групованні даних зазначені не всі початкові значення, що залишилися елементи групи автоматично заповнюються кулями.

Ініціалізація безрозмірного масиву

Уявіть собі ініціалізацію таблиці повідомлень про помилки, кожне з яких зберігається в одномірному масиві.

```
char el [12] = "Помилка при читанні\n";
char e2[13] = "Помилка при записі";
char e3[18]  =  "Неможливо відкрити  файл";
```

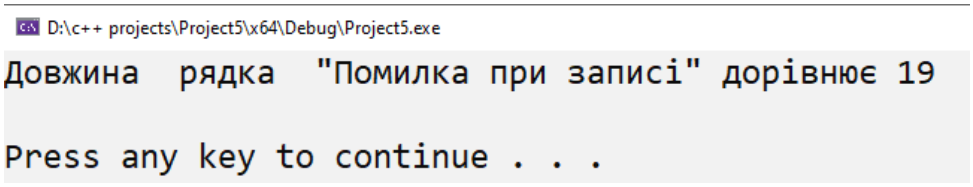
Щоб задати правильний розмір масивів, довелося б підраховувати точну кількість символів у кожному повідомленні. На щастя, компілятор може сам визначити розмір масиву. Якщо в операторі ініціалізації не зазначений розмір масиву, компілятор автоматично створить масив, що вміщає всі початкові значення. Такий масив називається *безрозмірним* (unsized array). Використовуючи цей підхід, можна переписати попередній фрагмент програми інакше.

```
char el[ ] = "Помилка при читанні";
char e2[ ] = "Помилка при записі";
char e3[ ]  =  "Неможливо відкрити файл";
```

В цьому випадку оператор:

```
cout<<"Довжина  рядка  \""<<e2<<"\" дорівнює "<<sizeof(e2)<<endl;
```

виведе на екран повідомлення:



Ініціалізація безрозмірних масивів не тільки полегшує програмування, але й дозволяє змінювати повідомлення, не піклуючись про розмір масивів.

Цей спосіб застосуємо й до багатовимірних масивів. Для цього потрібно вказати всі розміри, крім першого. (Інші розміри потрібні для того, щоб компілятор правильно виконував індексацію масиву.) Це дозволяє створювати таблиці змінної довжини, оскільки компілятор автоматично розміщає їх у пам'яті. Наприклад, оголошення масиву **sqrt** як безрозмірного виглядає так:

```
int  sqrs[ ][2]  =  {
{1,  1},
{2,  4},
{3,  9},
{4,  16},
{5,  25},
{6,  36},
{7,  49},
{8,  64},
{9, 81},
{10,  100}
};
```

Перевага такого оголошення складається в тому, що розмір таблиці можна змінювати, не міняючи розміри масиву.

Цикл foreach

Використання циклу з параметром дає гнучкий механізм реалізації циклічних обчислень та виконання операції, які однакові для значного об’єму даних.

Цикл **for** логічніше використовувати і при застосуванні масивів. Проте при використанні кількох масивів, головно коли аргументи одного масиву беруться з іншого, використання циклу **for** може призвести до технічних помилок, до неврахування певних параметрів.

Тому в С ++ 11 додали новий тип циклу - **foreach** (або «цикл, заснований на діапазоні»), який надає більш простий і безпечний спосіб ітерації по масиву (або будь-який інший структурі даних).

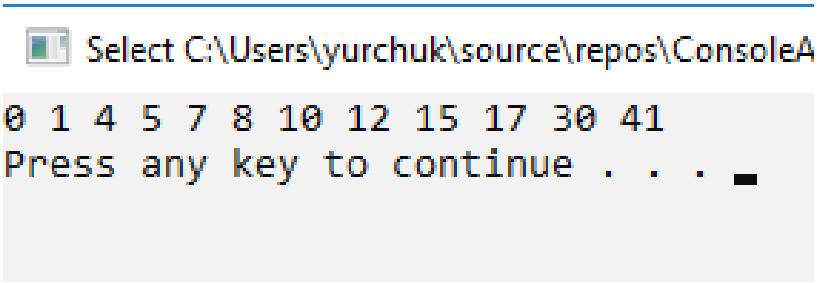
Синтаксис циклу **foreach** наступний:

```
for (оголошення_елемента: масив)
  команди;
```

Виконується ітерація по кожному елементу масиву, привласнюючи значення поточного елемента масиву змінній, оголошеної як елемент (оголошення_елемента). З метою кращої продуктивності голошений елемент повинен бути того ж типу, що й елементи масиву, інакше станеться неявне перетворення типу.

Розглянемо простий приклад з використанням циклу **foreach** для виведення всіх елементів масиву **math**:

```
#include "windows.h"

#include <iostream>
using namespace std;
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int math[] = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41 };
	for (int number : math)
		cout << number << ' ';

	cout << endl;
	system("pause");
	return 0;

	}
```



Розглянемо докладніше, як це все працює. При виконанні циклу **foreach** змінній **number** присвоюється значення першого елемента (тобто 0). Далі програма виконує набір команд в тілі циклу. В нашому випадку це виведення значення змінної **number**, тобто нуля. Потім цикл **for** виконується знову і значенням **number** вже є 1 (другий елемент масиву). Вивід значення **number** виконується знову. Цикл продовжує виконання до тих пір, поки в масиві не залишиться непройденого елементів. В кінці виконання програма повертає 0 в операційну систему (оператор return).

Зверніть увагу, змінна **number** не є індексом масиву. Їй просто присвоюється значення елемента масиву в поточній ітерації циклу.

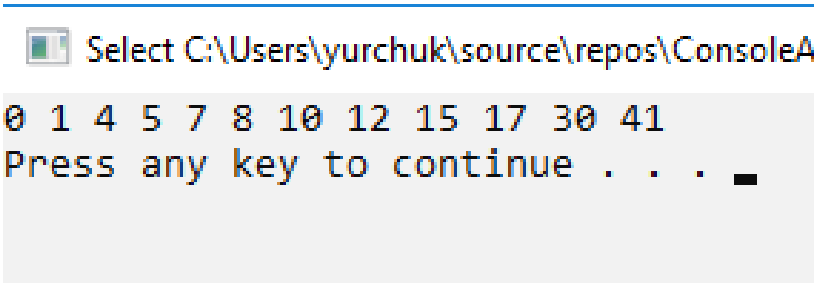
Цикл foreach і ключове слово auto

Оскільки оголошений елемент циклу **foreach** повинен бути того ж типу, що й елементи масиву, то це ідеальний випадок для використання ключового слова **auto**, коли ми дозволимо C ++ обчислити тип даних елементів масиву за нас.

Наприклад:

```
#include "windows.h"
```

```
#include <iostream>
using namespace std;
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int math[] = { 0, 1, 4, 5, 7, 8, 10, 12, 15, 17, 30, 41 };
	for (auto number : math)
		cout << number << ' ';

	cout << endl;
	system("pause");
	return 0;

	}
```



Цикл foreach і посилання

У прикладах вище оголошений елемент завжди оголошується в якості змінної (значення):

```
int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
for (auto element : array) // element буде копією поточного елемента масиву
cout << element << '';
```

Тобто кожен оброблений елемент масиву копіюється в змінну element. А це копіювання може виявитися витратним, в більшості випадків ми можемо просто посилатися на вихідний елемент за допомогою посилання:

```
int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
for (auto & element : array) // символ амперсанда робить element посиланням на
                             поточний елемент масиву, запобігаючи створення копії
cout << element << '';
```

В наведеному вище прикладі в якості оголошується елемента циклу **foreach** використовується посилання на поточний елемент масиву, при цьому копіювання цього елемента не відбувається. Але із зазначенням звичайного посилання будь-які зміни елемента будуть впливати на сам масив, що не завжди може бути бажано.

Звичайно ж, гарною ідеєю буде зробити оголошений елемент const, тоді ви зможете його використовувати в режимі «тільки для читання»:

```
int array[7] = { 10, 8, 6, 5, 4, 3, 1 };
for (const auto &element : array) // element - це константная посилання на поточний
                                  елемент масиву в ітерації
cout << element << '';
```

Правило: Використовуйте звичайні посилання або константні посилання в якості оголошується елемента в циклі **foreach** (в цілях поліпшення продуктивності).

Використана література

1. Г. Шилдт, Полный справочник по C++, 4-е видання, в-во «Вильямс», 2006
2. Х.М.Дейтел, Как програмировать на C++, 4-е видання, в-во «Бином-Пресс» 2009.
3. Р. Лафоре, Объектно-ориентированное программирование в C++, в-во «Питер», 2004, с 924.

Лекція 6
Основні поняття

План

1.П'ять основних типів даних

У C++ існують п'ять елементарних типів даних: символ, ціле число, число із плаваючою крапкою, число із плаваючою крапкою подвоєної точності й змінна, що не має значень. Їм відповідають наступні ключові слова: **char, int, float, double** і **void.** Всі інші типи даних у мові C++ створюються на основі елементарних типів, зазначених вище. Розмір змінних і діапазон їхніх значень залежить від типу процесора й компілятора. Однак у всіх випадках розмір символу дорівнює 1 байт. Розмір цілочисельної змінної звичайно дорівнює довжині машинного слова, прийнятої в конкретній операційній системі. У більшості 16-бітових операційних систем, наприклад, DOS і Windows 3.1, розмір цілочисельної змінної дорівнює 16 біт. У більшості 32-бітових операційних систем, наприклад Windows 2000, цей розмір дорівнює 32 біт. Однак, прагнучи до машино незалежності програм, варто уникати конкретних припущень про розмір цілочисельних змінних. Важливо чітко розуміти, що й у мові З, і в мові C++ обмовляється лише *мінімальний діапазон* у якому змінюються значення змінних кожного типу, а не їхній розмір у байтах.

Точне подання чисел із плаваючою крапкою залежить від їхньої конкретної реалізації. Розмір цілого числа звичайно дорівнює довжині машинного слова, прийнятої в операційній системі. Значення змінних типу **char,** як правило, використовуються для подання символів, передбачених у системі кодування ASCII. Значення, що виходять за межі припустимого діапазону, на різних комп'ютерах обробляються по-різному.

Діапазон зміни змінних типу **float** і **double** залежить від способу подання чисел із плаваючою крапкою. У кожному разі, цей діапазон досить широкий. Стандарт мови З визначає мінімальний діапазон зміни чисел із плаваючою крапкою: від 1E-37 до 1E+37. Мінімальна кількість цифр, що визначають точність чисел із плаваючою крапкою, зазначене в табл. 2.

Таблица 2. Характеристики основних типів данних C++

Ім’я типа	Байти	Діапазон значень
int	4	від -2 147 483 648 до 2 147 483 647
unsigned int	4	від 0 до 4 294 967 295
__int8	1	від -128 до 127
unsigned __int8	1	від 0 до 255
__int16	2	від -32 768 до 32 767

Ім’я типа	Байти	Діапазон значень
unsigned __int16	2	від 0 до 65 535
__int32	4	від -2 147 483 648 до 2 147 483 647
unsigned __int32	4	від 0 до 4 294 967 295
__int64	8	від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned __int64	8	від 0 до 18 446 744 073 709 551 615
bool	1	false або true
char	1	від -128 до 127
signed char	1	від -128 до 127
unsigned char	1	від 0 до 255
short	2	від -32 768 до 32 767
unsigned short	2	від 0 до 65 535
long	4	від -2 147 483 648 до 2 147 483 647
unsigned long	4	від 0 до 4 294 967 295
long long	8	від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
unsigned long long	8	від 0 до 18 446 744 073 709 551 615
double	8	1,7E +/- 308 (15 знаків)
float	4	3.4E+/- 38
long doublt	10	3.4E +/-4932
wchar_t	2	від 0 до 65 535

Тип Void використовується для визначення функції, що не повертає ніяких значень, або для створення узагальненого покажчика (generic pointer).

2. Модифікація основних типів

За винятком типу void, основні типи даних можуть мати різні *модифікатори* (modifiers), які використовуються для більше точного настроювання. От їхній список.

signed

unsigned

long
short

Цілочисельні типи можна модифікувати за допомогою ключових слів signed, short, long і unsigned. Символьні типи можна уточнювати за допомогою модифікаторів unsigned і signed. Крім того, тип double можна модифікувати ключовим словом long Застосування модифікатора signed допускається, але є зайвим, оскільки за замовчуванням всі цілі числа мають знак. Найбільш важливий цей модифікатор при уточненні типу char у тих реалізаціях мови З, де тип char за замовчуванням знака не має.

Різниця між цілими числами, що мають і не мають знак, полягає в інтерпретації біта в старшому розряді. Якщо в програмі визначене ціле число зі знаком, то компілятор генерує код, у якому старший біт інтерпретується як ознака знака *(*sign flag). Якщо ознака знака дорівнює 0, число вважається позитивним, якщо він дорівнює 1 - негативним.

Інакше кажучи, негативні числа представляються за допомогою *додаткового коду* (two's complement), у якому всі біти числа (за винятком старшого розряду) інвертовані, потім до отриманого числа додається одиниця, а ознака знака встановлюється рівним 1.

Цілі числа зі знаком грають досить важливу роль у дуже багатьох алгоритмах, але діапазон їхньої зміни вдвічі коротше, ніж у цілих чисел, що не мають знака. Розглянемо двійкове подання числа 32767.

0 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1

Допустимо, старший біт дорівнює 1, виходить, це число стане рівним —1. Однак, якби ця змінна була оголошена як **unsigned int,** її значення стало б рівним 65535. Якщо модифікатор типу використовується окремо (тобто без вказівки елементарного типу), за замовчуванням передбачається, що повідомлена змінна має тип int. Отже, зазначені нижче специфікатори типів еквівалентні.

Хоча одного специфікатора **int** цілком достатньо, багато програмістів воліють явно вказувати його модифікації.

Особливості роботи з числам з плаваючою крапкою

Це пов’язано з особливостями переведення чисел в двійкове представлення (вся інформація представляється двійковими числами). Наприклад дійсне число 3.0 не зовсім точно 3.0.

```
#include <iostream>
#include "windows.h"
using namespace std;

int main()
{
     double x, y;


     for (x = 0.3; x <= 3.5; x += 0.3)
```

```
        {
            if (x == 3) cout << "Yes";
        }
    cout << endl;
    system("pause");
    return 0;
}
```

"Yes" – не виведеться

```
#include <iostream>
#include "windows.h"
using namespace std;
 int main()
{
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);
    double d1(100 - 99.99); // повинно бути 0.01
    double d2(10 - 9.99); // повинно бути 0.01

    if (d1 == d2)
        cout << "d1 == d2" << "\n";
    else if (d1 > d2)
        cout << "d1 > d2" << "\n";
    else if (d1 < d2)
        cout << "d1 < d2" << "\n";

    cout << endl;
    system("pause");
    return 0;
}
```

Результат:
d1 > d2

У програмі вище, d1 = 0.0100000000000005116, а d2 = 0.0099999999999997868. Обидва цих числа дуже близькі до 0.1, але d1 більше d2. Вони не є рівними.

Іноді порівняння чисел типу з плаваючою крапкою буває неминучим. В такому випадку слід використовувати оператори >, <, >= і <=, тільки якщо значення не дуже близькі. А ось якщо два операнди дуже близькі за значеннями, то результат вже може бути несподіваний. В наведеному вище прикладі наслідки неправильного результату незначні, а ось з оператором рівності справи йдуть гірше, так як навіть при найменшій неточності результат відразу змінюється на протилежний очікуваному. Не рекомендується використовувати

оператори == чи != з числами типу з плаваючою крапкою. Замість них краще використовувати функцію, яка обчислює, наскільки близькі ці два числа. Якщо вони “достатньо близькі”, то ми вважаємо їх рівними. Значення, що використовується для представлення терміну “достатньо близькі”, називається епсилоном. Воно, зазвичай, невелике (наприклад, 0.0000001).

```
bool isAlmostEqual(double a, double b, double epsilon)
{
    // Якщо різниця між a і b менше значення епсилону, то тоді
a і b - "достатньо близькі"
    return fabs(a - b) <= epsilon;
}
```

3. Кваліфікатори const і volatile

Кваліфікатор const

Змінні типу **const** не можуть змінюватися, однак їх можна ініалізувати. Компілятор може помістити ці змінні в постійний запам'ятовувальний пристрій (Random Access Memory - ROM). Розглянемо приклад.

```
const  int  a=10;
```

Це оголошення створює цілочисельну змінну з ім'ям **a** й початковим значенням, рівним числу 10, що в іншій частині програми змінити неможливо. Однак змінну **a** можна використовувати в інших вираженнях. Змінна типу **const** може одержати своє значення або під час ініціалізації, або за допомогою інших машинозалених засобів.

Кваліфікатор **const** можна використовувати для захисту об'єктів, переданих у функцію як аргументи. Іншими словами, якщо у функцію передається покажчик, вона може модифікувати фактичне значення, на яке він посилається. Але якщо покажчик уточнити кваліфікатором **const,** функція не зможе змінити значення у відповідному осередку. Наприклад, функція **sp_to_dash ()**, розглянута нижче, виводить на друк тире, заміняючи ним пробіли в рядку, що є її аргументом. Інакше кажучи, "рядок для перевірки" буде надрукований як " рядок-для-перевірки". Використання кваліфікатора **const** в оголошенні параметра гарантує, що код, поміщений усередині функції, не зможе модифікувати об'єкти, на які посилається покажчик.

```
#include <stdio.h>
void sp_to_dash(const char *str);
int main(void)
{
  sp_to_dash("рядок для перевірки");
  return 0;
}
void sp_to_dash(const char *str)
{
  while(*str) {
  if(*str== ‘ ‘) printf("%c", ‘-‘);
```

```
  else printf ("%c", *str);
  str++;
  }
}
```

Якби ми спробували модифікувати строку всередині функції **sp_to_dash(),** компілятор видав би повідомлення про помилку. Наприклад, якщо написати функцію **sp_to_dash()** так, як показано нижче, то виникла б помилка компіляції.

```
/*  Помилка   */
void sp_to_dash(const char *str)
{
  while(*str) {
  if(*str== ' ') *str = '-'; /* Не можна! Аргумент str
є константою */
  printf("%c", *str);
  str++;
  }
}
```

Багато функцій із стандартної бібліотеки використовують кваліфікатор **const** в оголошеннях своїх параметрів. Наприклад, прототип функції **strlen()** виглядає наступним чином.

```
size_t  strlen(const  char   *str);
```

Оголосивши покажчик **str** константою, автори захистили від змін рядок, на котрий він посилається. В принципі, якщо в стандартній функції немає необхідності модифікувати об'єкт, на який посилається її аргумент, то цей аргумент оголошується константним.

Кваліфікатор const можна також використовувати для запобігання модифікації змінних. Пам'ятайте, що змінну типу const можна модифікувати лише за межами програми. Наприклад, її значення можна змінити за допомогою апаратних пристроїв. Однак, якщо змінна оголошена константою, можна гарантувати, що її зміна може відбутися лише внаслідок зовнішнього втручання.

Кваліфікатор volatile

Кваліфікатор **volatile** повідомляє компілятору, що значення змінної може змінюватися неявно. Наприклад, адрес глобальної змінної можна передати таймеру операційної системи і використовувати його для відліку реального часу. В цьому випадку вмістиме змінної змінюється без явного виконання будь-якого оператора присвоювання. Це дуже важливо, оскільки більшість компіляторів мови C / C ++ автоматично оптимізують деякі вирази, вважаючи, що значення змінних не змінюються, якщо вони не вказані в лівій частині оператора присвоєння. Таким чином, їх значення немає сенсу перевіряти ще при кожному зверненні. Крім того, деякі компілятори змінюють порядок обчислення виразів в ході компіляції. Кваліфікатор **volatile** запобігає такі зміни.

Кваліфікатори **const** і **volatile** можна використовувати одночасно. Наприклад, якщо 0x30 - значення, що зберігається в порте, яке змінюється тільки під впливом зовнішніх обставин, то наступне оголошення запобіжить непередбачені побочні ефекти.

```
const volatile char  *port =  (const volatile char  *)  0x30;
```

4. Специфікатори зберігання

В мові C++ існують специфікатори зберігання.

extern
static
register
auto

Ці специфікатори повідомляють компілятору, як зберігати оголошену змінну. Загальний вид оголошення, що використовує ці специфікатори, наведено нижче.

специфікатор_зберігання тип ім'я_змінної

Зверніть увагу на те, що специфікатори зберігання передують всім іншим елементам оголошення.

Специфікатор extern

Перш ніж розглянути специфікатор **extern**, опишемо систему зв'язків у мові C / C ++. У мові C / C ++ передбачено три категорії: зовнішні, внутрішні зв'язки і їх відсутність. Як правило, функції та глобальні змінні мають зовнішні зв'язки. Це означає, що вони доступні з будь-якої частини програми. Глобальні об'єкти, оголошені за допомогою специфікатора **static** (описаного в наступному розділі), мають внутрішні зв'язки. Вони доступні лише всередині файлу, в якому описані. Локальні змінні не мають зв'язків і, отже, видимі лише всередині свого блоку.

Головне призначення специфікатора **extern** - вказати, що і оголошений об'єкт володіє зовнішніми зв'язками в рамках всієї програми. Щоб зрозуміти, чому це так важливо, необхідно розрізняти оголошення (declaration) та визначення (definition). В оголошенні зазначаються ім'я та тип об'єкта. Визначення виділяє пам'ять для об'єкта. У програмі може бути декілька оголошень одного і того ж об'єкта, але тільки одне визначення.

У більшості випадків оголошення змінної одночасно є її визначенням. Однак, вказавши перед ім'ям змінної специфікатор **extern**, можна оголосити її, не визначаючи. Таким чином, щоб звернутися до змінної, визначеної в другій частині програми, її слід оголосити, використовуючи специфікатор extern.

Розглянемо приклад використання специфікатора extern. Зверніть увагу на те, що глобальні змінні **first** і **last** оголошені після функції **main ()**.

```
#include <stdio.h>
int main(void)
{
 extern int first; /* Глобальні змінні */
 printf("%d %d", first, last);
 return 0;
}
/* Глобальне визначення змінних first і last */
int first = 10, last = 20;
```

Ця програма виведе на друк числа 10 20, оскільки глобальні змінні **first** і **last,** які використовуються при виклику функції **printf** (), ініціалізовані саме цими значеннями. Оскільки специфікатор **extern** у функції **main** () повідомляє компілятора, що змінні **first** і **last** оголошені в іншому місці (у цьому випадку

наприкінці поточного файлу), програма буде скомпільована без помилок, навіть якщо змінні **first** і **last** будуть використані раніше своїх визначень.

Варто мати на увазі, що оголошення змінних у якості зовнішніх, як показано в попередній програмі, необхідно лише тому, що змінні **first** і **last** не були оголошені до свого використання у функції **main()**. **У** противному випадку необхідності в специфікаторі **extern** не було б. Виявивши змінну, раніше не оголошену в поточному блоці, компілятор перевіряє, чи не оголошена вона в блоках, що охоплюють її. Якщо ні, компілятор перевіряє глобальні змінні. Виявивши збіг, компілятор уважає, що дане посилання ставиться до раніше оголошеній змінного. Якщо необхідно використовувати змінну, оголошену пізніше в тім же файлі, варто застосовувати специфікатор **extern.**

Як вказувалося вище, специфікатор **extern** дозволяє оголосити змінну, не визначаючи неї. Однак, якщо при цьому виконується ініціалізація змінної, оголошення **extern** стає визначенням. Це важливо, оскільки об'єкт може мати кілька оголошень, але тільки одне визначення.

Специфікатор **extern** відіграє важливу роль у програмах, що складаються з декількох файлів. Програму, написану мовою C/C++, можна розділити на кілька файлів, скомпілювати їх окремо, а потім відредагувати зв'язки між ними. Для цього необхідно мати можливість повідомити всі файли про глобальних змінних, існуючих у програмі. Найкраще оголосити всі глобальні змінні в одному файлі, а в інші оголосити їх за допомогою специфікатора **extern,** як показано нижче. Це підсилить машинонезависимость програми.

Перший файл	***Другий файл***
int x, y;	extern int x, y;
char ch;	extern char ch;
int main(void)	void func22(void)
{	{
/*...*/	x = y / 10;
}	}
void func1 (void)	void func23 (void)
{	{
x = 123;	y = 10;
}	}

Список глобальних змінних копіюється з першого файлу в другий, а потім додається специфікатор **extern.** Він повідомляє компілятора, що імена й типи змінних, наступних далі, оголошені в іншому місці. Іншими словами, специфікатор **extern** повідомляє компілятора типи й імена глобальних змінних, не виділяючи для них пам'ять ще раз. Всі посилання на зовнішні змінні розпізнаються в процесі редагування зв'язків.

На практиці оголошення зовнішніх змінних звичайно зберігаються в заголовних файлах, які просто включаються у вихідний код. Це й легше, і надійніше, ніж дублювати всі оголошення зовнішніх змінних у кожному файлі вручну.

Статичні змінні

Статичні змінні, на відміну від глобальних, невідомі поза своєю функцією або файлом, і зберігають свої значення між викликами. Це дуже корисно при створенні узагальнених функцій і бібліотек функцій, які можуть використовуватися іншими програмістами. Статичні змінні відрізняються як від локальних, так і від глобальних змінних.

Локальні статичні змінні

Якщо локальна змінна оголошена за допомогою специфікатора **static,** компілятор виділить для неї постійне місце зберігання, як і для глобальної змінної. Принципова відмінність локальної статичної змінної від глобальної полягає в тому, що перша залишається доступної лише усередині свого блоку. Простіше говорячи, локальна статична змінна - це локальна змінна, що зберігає свого значення між викликами функції.

Такі змінні дуже корисні при створенні ізольованих функцій, оскільки між викликами їх можна використовувати в інших частинах програми. Якби статичних змінних не було, довелося б застосовувати глобальні змінні, породжуючи неминучі побічні ефекти. Прикладом удалого застосування статичних змінних є генератор чисел, що породжує нове число, використовуючи попереднє значення. Для зберігання цього числа можна було б використовувати глобальну змінну, однак при кожному новому виклику її довелося б повідомляти заново, постійно перевіряючи, чи не конфліктує вона з іншими глобальними змінними. Застосування статичної змінної легко вирішує цю проблему.

```
int  series(void)
{
  static   int  series_num;
  series_num =  series_num+23;
  return  series_num;
}
```

У цьому прикладі змінна **series_num** продовжує існувати між викликами функції, а локальна змінна щораз створювалася б при вході й знищувалася при виході з функції. Таким чином, кожний виклик функції **series ()** породжує новий елемент ряду, використовуючи попереднє значення й не прибігаючи до глобальній змінного.

Локальну статичну змінну можна ініціалізувати. Початкове значення привласнюється лише один раз, а не при кожному вході в блок, як це відбувається з локальними змінними. Наприклад, у наведеній нижче версії функції **series()** змінна **series_num** ініціалізується числом 100.

```
int series(void)
{
static   int   series__num  =   100;
series_num =  series_num+23;
return  series_num;
}
```

Тепер ряд буде завжди починатися із числа 123. Хоча в багатьох додатках це цілком прийнятно, звичайно генератори чисел надають користувачеві право вибору початкового значення. Для цього можна зробити змінну **series_num** глобальної. Однак саме для того, щоб уникнути цього, і були створені статичні змінні. Це приводить до другого способу використання статичних змінних.

Глобальні статичні змінні

Застосування специфікатора **static** до глобальної змінної змушує компілятор створити глобальну змінну, видиму тільки в межах поточного файлу. Незважаючи на те що ця змінна залишається глобальною, в інших файлах вона не існує. Отже, змінити її значення шляхом втручання ззовні неможливо. Це запобігає побічним ефектам. У деяких ситуаціях, у яких локальні статичні змінні виявляються недостатніми, можна створити невеликий файл, що містить лише функції, які використовують конкретну глобальну статичну змінну, окремо скомпілювати його й застосовувати без ризику виникнення побічних ефектів.

Щоб проілюструвати застосування глобальної статичної змінної, перепишемо генератор чисел з попереднього розділу таким чином, щоб початкове значення задавалося при виклику функції **seriesstartO**. Весь файл, що містить функції **series () series_start ()** і **series_num()**, показаний нижче.

```
/*  Всі  функції повинні перебувати  в  тому самому  файлі.  */
static  int  series_num;
void series_start(int  seed);
int  series (void) ;

int  series(void)
{
  series_num =  series_num+23;
  return series_num;
}
/*  Ініціалізація змінної series_num */
void series_start(int  seed)
{
series_num =  seed;
}
```

Виклик функції **series_start()** ініціалізує генератор чисел. Після цього наступний елемент ряду породжується новим викликом функції **series ()**.

До відома: локальні статичні змінні видимі лише в межах блоку, де вони оголошені, а глобальні статичні змінні — у межах файлу. Якщо помістити функції **series ()** і **series_start ()** у бібліотеку, то змінна **series_num** стане невидимої. Більше того, у програмі можна оголосити нову змінну **series_num** (зрозуміло, в іншому файлі). По суті, модифікатор **static** дозволяє створювати змінні, видимі лише в межах функцій, не породжуючи побічних ефектів.

Модифікатор **static** дозволяє приховувати частину програми від інших модулів. Це надзвичайно важливо при розробці більших і складних програм.

Специфікатор register

Специфікатор **register** спочатку застосовувався лише до змінних типу **int, char** або покажчикам. Однак згодом його тлумачення було розширено, і тепер його можна застосовувати до змінних будь-якого типу.

Специфікатор **register** був розроблений для того, щоб компілятор зберігав значення змінної в регістрі центрального процесора, а не пам'яті, як звичайно. Це означає, що операції над реєстровими змінними виконуються набагато швидше.

У цей час зміст специфікатора **register** трактується дуже широко. У стандарті мови C++ говориться, що специфікатор **register** — це "підказка компіляторові, що даний об'єкт використовується дуже інтенсивно". На практиці в регістрах центрального процесора зберігаються символи й цілі числа. Більші об'єкти на зразок масивів, мабуть, там не помістяться, але вони зберігають можливість пріоритетної обробки компілятором. Залежно від реалізації компілятора й операційної системи обробка реєстрової змінної здійснюється по-різному. З технічної точки зору компілятор може просто ігнорувати специфікатор **register** і обробляти змінну як звичайно, але на практиці це трапляється рідко.

Специфікатор **register** можна застосовувати тільки до локальних змінних і формальних параметрів. До глобальних змінних він не застосовується. У розглянутому нижче прикладі функція обчислює значення M^e для цілих чисел.

```
int   int pwr(register  int  m,   register  int  e)
{
  register  int  temp;
  temp =  1;
  for (;   e;   e--)   temp  =  temp  *  m;
  return  temp;
}
```

У даному прикладі змінні **e, m** і **temp** оголошені як реєстрові, оскільки вони використовуються усередині циклів. Той факт, що реєстрові змінні оптимізовані по швидкості, робить їх ідеально підходящими для використання в циклах. Звичайно реєстрові змінні застосовуються там, де до однієї й тієї же змінної відбувається дуже багато звернень. Це дуже важливо, оскільки реєстровими можна оголосити скільки завгодно змінних, але не всі вони будуть забезпечувати однакову швидкість доступу.

Кількість реєстрових змінних у кожному блоці оптимізується по швидкості й залежить як від операційної системи, так і від реалізації мови C/C++. Турбуватися про кількість реєстрових змінних не варто, оскільки компілятор автоматично перетворює "зайві" реєстрові змінні у звичайні. (Це забезпечує машинонезалежність програм у рамках досить широкого спектра процесорів.)

Звичайно в реєстрах центрального процесора одночасно можуть розташовуватися, принаймні, два змінні типи **char** або **int.** Оскільки вибір операційних систем дуже широкий, універсальних рекомендацій з використання реєстрових змінних не існує. У кожному конкретному випадку варто звертатися до документації компілятора.

У мові З адресу реєстрової змінної за допомогою оператора & обчислити неможливо (причини обговорюються нижче). Це цілком логічно, оскільки реєстрові змінні зберігаються, як правило, у процесорі, а його пам'ять не адресується. Однак на мову C++ це обмеження не поширюється, хоча в цьому

випадку обчислення адреси реєстрової змінної може перешкодити оптимізації програми.

Незважаючи на те що в цей час зміст специфікатора **register** істотно розширений у порівнянні із традиційним, на практиці він як і раніше застосовується в основному до цілочисельних і символьних змінних. Таким чином, не слід очікувати від специфікатора **register** істотного прискорення роботи програми, якщо він застосовується до змінних інших типів.

5. Заголовочні файли.

Як тільки програми стають більшими і їх код уже не поміщується в декількох файлах, записувати кожен раз **попередні оголошення** для функцій, які ми хочемо використовувати, але які знаходяться в інших файлах, стає все нудніше і нудніше. Добре було б, якби всі попередні оголошення знаходилися в одному місці, чи не так?

Файли *.cpp* не є єдиними файлами в проектах. Є ще один тип файлів — **заголовкові файли** (або **"заголовки"**), які мають розширення *.h*. Метою заголовків є зручне зберігання набору оголошень об'єктів для їх подальшого використання в інших програмах.

Заголовкові файли зі Стандартної бібліотеки C++

Розглянемо наступну програму:

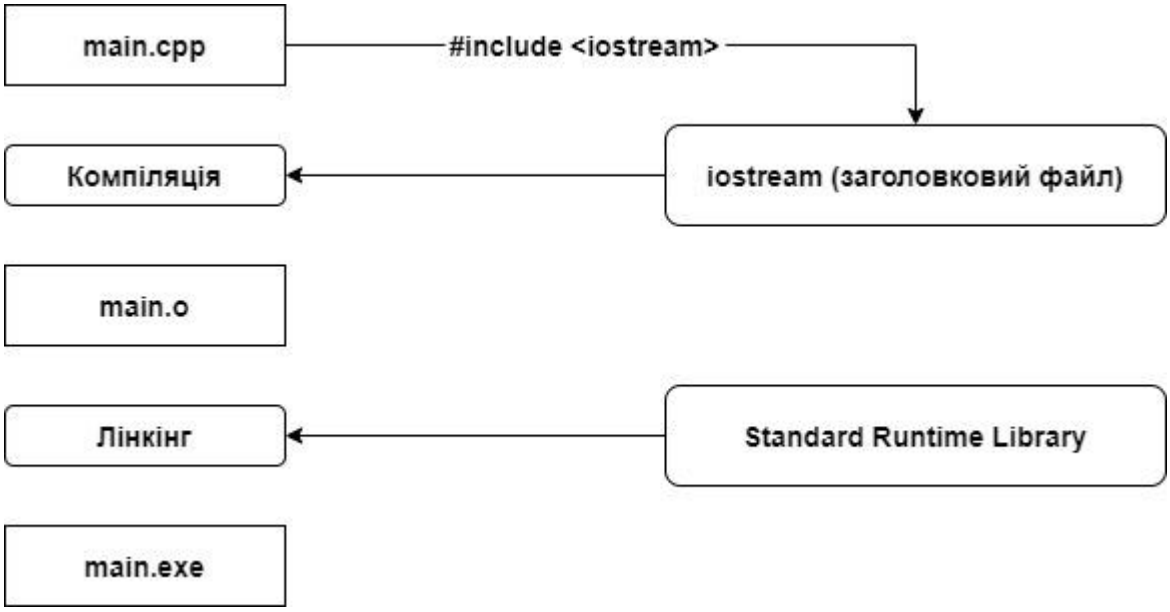
```
1    #include <iostream>
2
3    int main()
4    {
5        std::cout << "Hello, world!" << std::endl;
6        return 0;
7    }
```

Результат виконання програми:

```
Hello, world!
```

У цій програмі ми використовуємо **об'єкт std::cout**, який ніде не визначаємо. Як компілятор знає, що це таке? Справа в тому, що std::cout оголошений в заголовку iostream. Коли ми пишемо #include <iostream>, ми робимо запит, щоб весь вміст заголовка iostream було скопійовано в наш файл. Таким чином, весь вміст iostream стає доступним для використання.

Як правило, в заголовкових файлах записуються тільки оголошення, без визначень. Отже, якщо std::cout тільки оголошений в заголовку iostream, де ж він визначається? Відповідь: "В Стандартній бібліотеці C++, яка автоматично підключається до вашого проекту на етапі лінкінгу".



Подумайте, що сталося б, якби заголовкового файлу iostream не було? Кожен раз, при використанні std::cout, вам доводилося би вручну копіювати всі попередні оголошення, пов'язані з std::cout, в верхню частину вашого файлу! Добре ж, що можна просто #include <iostream>, чи не так?

Пишемо свої власні заголовкові файли

Тепер давайте повернемося до прикладу, який ми обговорювали на попередньому уроці. У нас було два файли: add.cpp і main.cpp.

add.cpp:

```
1   int add(int x, int y)
2   {
3       return x + y;
4   }
```

main.cpp:

```
1
2   #include <iostream>
3
4   int add(int x, int y);
5   int main()
6   {
7       std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
8       return 0;
9   }
```

Ми використовували попереднє оголошення, щоб повідомити компілятор, що таке add(). Як ми вже говорили, записувати в кожному файлі попередні оголошення використовуваних функцій — справа не дуже захоплююча.

І тут нам на допомогу приходять заголовки. Досить просто написати один заголовок і його можна буде повторно використовувати в будь-якій кількості програм. Також вносити зміни в такий код (наприклад, додати ще один **параметр**) набагато легше, ніж вносити зміни у всі наявні файли, де використовується цей код.

Написати свій власний заголовок не так вже й складно. Заголовкові файли складаються з двох частин:

Директиви препроцесора — зокрема, header guards, які запобігають викликам заголовка більше одного разу з одного і того ж файлу.

Вміст заголовкового файлу — набір оголошень.

Всі ваші заголовки (які ви написали самостійно) повинні мати розширення .h.

add.h:

```
// Починаємо з директив препроцесора. ADD_H – це довільне
1 унікальне ім'я (зазвичай використовується ім'я заголовка)
2   #ifndef ADD_H
3   #define ADD_H
4
5   // А це вже вміст заголовка
6   int add(int x, int y); // прототип функції add() (пам'ятайте про крапку з
7 комою в кінці!)
8
9   // Закінчуємо директивою препроцесора
    #endif
```

Щоб використовувати цей файл в main.cpp, вам спочатку потрібно буде підключити його до проекту.

main.cpp, в якому ми підключаємо add.h:

```
1   #include <iostream>
2   #include "add.h"
3
4   int main()
5   {
6       std::cout << "The sum of 3 and 4 is " << add(3, 4) << std::endl;
7       return 0;
8   }
```

add.cpp залишається без змін:

```
1   int add(int x, int y)
2   {
3       return x + y;
4   }
```

Коли компілятор зустрічає #include "add.h", то він копіює весь вміст add.h в поточний файл. Таким чином, ми отримуємо попереднє оголошення функції add().

Примітка: При підключенні заголовкового файлу, весь його вміст вставляється відразу ж після рядка #include ....

Якщо ви отримали помилку від компілятора, що add.h не знайдено, то переконайтеся, що ім’я вашого файлу точно add.h. Цілком можливо, що ви могли зробити помилку, наприклад: просто add (без .h) або add.h.txt або add.hpp.

Якщо ви отримали помилку від лінкера, що функція add() не визначена, то переконайтеся, що ви коректно підключили add.cpp до вашого проекту (і до компіляції також)!

Кутові дужки (<>) vs. Подвійні лапки (“”)

Використовуючи кутові дужки, ми повідомляємо компілятору, що

заголовковий файл, який підключається, написаний не нами (він є “системним”, тобто тим, який надається Стандартною бібліотекою C++), так що шукати цей заголовок слід в системних директоріях. Подвійні лапки повідомляють компілятор, що ми підключаємо наш власний заголовок, який ми написали самостійно, тому шукати його слід в поточній директорії нашого проекту. Якщо файлу там не виявиться, то компілятор почне перевіряти інші шляхи, у тому числі і системні директорії.

Правило: Використовуйте кутові дужки для підключення “системних” заголовкових файлів і подвійні лапки для всього іншого (ваших власних заголовків).

Варто відзначити, що одні заголовки можуть підключати інші заголовки.

Чому iostream пишеться без закінчення .h?

Ще одне часте питання: “Чому iostream (або будь-який інший зі стандартних заголовкових файлів) при підключенні пишеться без закінчення .h?”. Справа в тому, що є два окремих файла: iostream.h (заголовок) і просто iostream! Для пояснення потрібен короткий екскурс в історію.

Коли мова C++ тільки створювалася, всі файли бібліотеки Runtime мали закінчення *.h*. Оригінальні версії cout і cin оголошені в *iostream.h*. При стандартизації мови C++ комітетом ANSI, вирішили перенести всі функції з бібліотеки Runtime в простір імен std, щоб запобігти можливості виникнення конфліктів імен з ідентифікаторами користувачів (що, між іншим, є хорошою ідеєю!). Проте, виникла проблема: якщо всі функції перемістити в простір імен std, то старі програми переставали працювати!

Для забезпечення зворотної сумісності ввели новий набір заголовків з тими ж іменами, але без закінчення .h. Весь їх функціонал знаходиться в просторі імен std. Таким чином, старі програми з #include <iostream.h> не потрібно було переписувати, а нові програми вже могли використовувати #include <iostream>.

Коли ви підключаєте заголовки зі Стандартної бібліотеки C++, переконайтеся, що ви використовуєте версію без *.h* (якщо вона існує). В іншому випадку ви будете використовувати застарілу версію заголовкового файлу, який вже більше не підтримується.

Крім того, багато бібліотек, успадковані від мови Сі, які до сих пір використовуються в C++, також були продубльовані з додаванням префікса c (наприклад, stdlib.h став cstdlib). Функціонал цих бібліотек також був переміщений в простір імен std, щоб уникнути можливості виникнення конфліктів імен з користувацькими ідентифікаторами.

Правило: При підключенні заголовкових файлів зі Стандартної бібліотеки C++, використовуйте версію без “.h” (якщо вона існує). Користувацькі заголовки ж повинні мати закінчення “.h”.

Чи можна записувати визначення в заголовкових файлах?

Мова C++ не скаржитиметься, якщо ви це зробите, але так робити не прийнято.

Як вже було сказано вище, при підключенні заголовкового файлу, весь його вміст вставляється відразу ж після рядка #include. Це означає, що будь-які визначення, які є в заголовку, скопіюються в ваш файл.

Для невеликих проектів, це, ймовірно, не буде проблемою. Але для більш масштабних проектів це може збільшувати час компіляції (оскільки код повторно

компілюватиметься) і розмір виконуваного файлу. Якщо внести зміни до визначень, які знаходяться в файлі *.cpp*, то перекомпілювати доведеться тільки цей файл. Якщо ж внести зміни до визначень, які записані в заголовку, то перекомпілювати доведеться кожен файл, який підключає цей заголовок, використовуючи директиву препроцесора #include. І ймовірність того, що через одну невелику правку вам доведеться перекомпілювати весь проект, різко зростає!

Іноді робляться винятки для простих функцій, які навряд чи зміняться (наприклад, де визначення складається всього лише з одного рядка).

Поради

Ось декілька порад щодо написання власних заголовкових файлів:

Завжди використовуйте директиви препроцесора.

Не визначайте змінні в заголовкових файлах, якщо це не константи. Заголовкові файли слід використовувати тільки для оголошень.

Не визначайте функції в заголовкових файлах.

Кожен заголовок повинен виконувати свою певну роботу і бути якомога більш незалежним. Наприклад, ви можете помістити всі ваші оголошення, пов'язані з файлом *A.cpp* в файл *A.h*, а всі ваші оголошення, пов'язані з *B.cpp* — в файл *B.h*. Таким чином, якщо ви працюватимете тільки з *A.cpp*, то вам достатньо буде підключити тільки *A.h* і навпаки.

Використовуйте імена ваших робочих файлів в якості імен для ваших заголовків (наприклад, *grades.h* працює з *grades.cpp*).

Не підключайте одні заголовки з других заголовків.

Не підключайте файли *.cpp*, використовуючи директиву препроцесора #include.

6. Ініціалізація змінних

При оголошенні змінної їй можна привласнити початкове значення. Загальний вид ініціалізації виглядає в такий спосіб.

тип ім'я змінної = значення;

Розглянемо кілька прикладів.

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

Глобальні й локальні статичні змінні ініціалізуються тільки при запуску програми. Локальні змінні (за винятком статичних) ініціалізуються щораз при вході в блок, де вони описані. Неініціалізовані локальні змінні мають невизначене значення, поки до них не буде застосований оператор присвоювання. Неініціалізовані глобальні змінні й локальні статичні змінні автоматично встановлюються рівними нулю.

7. Константи

Константами (constants) називаються фіксовані значення, які програма не може змінити. Спосіб подання константи залежить від її типу. Іноді константи також називають *літералами* (literal).

Символьні константи кладуть в одинарні лапки. Наприклад, символи 'a' і '%' є константами. У мовах С и С++ передбачені розширені символи, які дозволяють використовувати інші мови, крім англійського. Їхня довжина дорівнює 16 біт. Для того щоб визначити розширену символьну константу, перед символом варто поставити букву L. Розглянемо приклад.

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

Тут змінній **wc** привласнюється розширена символьна константа, еквівалентна букві А. Розширені символи мають тип **wchar_t. У** мові С++ цей тип відноситься до елементарних типів.

Цілочисельними константи вважаються числами, що не мають дробової частини. Наприклад, числа 10 і -100 є цілочисельними константами. Константи із плаваючою крапкою містять дробову частину, що відділяється десятковою крапкою. Наприклад, число 11.123 являє собою константу із плаваючою крапкою. Крім того, у мові С++ числа із плаваючою крапкою можна представляти за допомогою наукового формату.

У мові С передбачено два типи для подання чисел із плаваючою крапкою: **float** і **double.** Використовуючи модифікатори, можна створити ще кілька варіантів основних типів. За замовчуванням компілятор приводить числові константи до найменшого підходящого типу. Таким чином, якщо припустити, що ціле число займає 16 біт, то число 10 за замовчуванням має тип **int,** а число 103000— тип **long.** Незважаючи на те що число 10 можна привести до символьного типу, компілятор не порушить границі типів. Єдине виключення із цього правила являє собою константа із плаваючою крапкою, що за замовчуванням має тип **double.**

У більшості програм угоди, прийняті за замовчуванням, є цілком розумними. Однак, використовуючи суфікси, типи констант можна задавати явно. Якщо після константи із плаваючою крапкою поставити суфікс **F**, вона буде мати тип **float.** Якщо замість букви **F** поставити букву **L**, константа одержить тип **long double.** Для цілочисельних типів суфікс **U** означає **unsigned,** а суфікс L — **long.** От кілька прикладів на цю тему.

Тип дані	***Приклади констант***
int	1 123 21000 -234
long int	35000L -34L
unsigned int	10000U 987U 40000U
float	123.23F 4.34e-3F
double	123.23 1.0 -0.9876324
long double	1001. 2L

Строкові константи

У мові С/С++ є ще один вид констант — строкові. *Рядок* (string) — це послідовність символів, укладена в подвійні лапки. Наприклад, "приклад рядка" — це рядок. Незважаючи на те, що в мові С можна визначати рядкові константи у ньому немає окремого типу даних для рядків. (У той же час, у мові С++ існує стандартний клас **string.)**

Не слід плутати символи й рядки. Окрема символьна константа береться в одинарні лапки, наприклад 'a'. Якщо прописати букву а в подвійні лапки, одержимо рядок "a", що складається з однієї букви.

Керуючі символьні константи

Практично всі символи можна вивести на друк, уклавши їх в одиночні лапки. Однак деякі символи, наприклад, символ переходу на новий рядок, неможливо ввести в рядок із клавіатури. Для цього в мові C/C++ передбачені спеціальні *керуючі символьні константи* (backslash character constants), зазначені в табл. 2.2. Вони називаються *эскейп-послідовностями* (escape sequences). Для того щоб гарантувати машинонезалежність, необхідно використовувати не ASCII-коди, еквівалентні керуючим символьним константам, а саме эскейп-послідовностям.

Таблиця 2.2. Керуючі символьні константи

Код	***Значення***
\b	Пробіл
\f	Прогін паперу
\n	Новий рядок
\r	Повернення каретки
\t	Горизонтальна табуляція
\ "	Подвійні лапки
\'	Одинарні лапки
\0	Нуль
\ \	Зворотна коса риса
\v	Вертикальна табуляція
\a	Звуковий сигнал
\?	Знак питання
\N	Восьмирічна константа N
\xN	Шістнадцятирічна константа N

Програма, наведена нижче, виконує перехід на новий рядок, виводить символ табуляції, а потім друкує рядок "Перевірка висновку".

```
#include  <stdio.h>

int main(void)
{
cout<<" \n\tПеревірка виведення. " ;

  return 0;
}
```

Форматне введення виведення

Мова C++ підтримує дві повноцінні системи вводу-виводу: одна з них успадкована від мови C, а інша є об’єктно-орієнтованою і визначена в мові C++

(із цього моменту ми будемо називати її просто *системою вводу-виводу мови* C++). Системи вводу-виводу мови C и C++ є абсолютно сумісними. Різні компоненти кожної із систем, наприклад, засоби для роботи з консоллю або файлами, просто по-різному реалізують той самий механізм.

Оскільки система вводу-виводу, успадкована від мови C, дуже різноманітна й ефективна, виникає питання: "Навіщо знадобилося створювати ще одну систему вводу-виводу?". Відповідь проста: система вводу-виводу мови C нічого не знає про об'єкти.Отже, щоб система вводу-виводу повністю відповідала принципам об’єктно-орієнтованого програмування, необхідні засоби для роботи з об'єктами, певними користувачем.Крім цього, система вводу-виводу мови C++ володіє ще рядом переваг, навіть якщо програма не дуже широко використовує об'єкти.Чесно говорячи, всі нові програми повинні використовувати тільки систему вводу-виводу мови C++.Стара система, успадкована від мови C, застосовується тільки для підтримки сумісності програм.

Порівняння старої й нової систем вводу-виводу

У цей час використовуються дві бібліотеки об’єктно-орієнтованого виводу: стара, заснована на вихідних специфікаціях мови C++, і сучасним, певним стандартом мови C++. Стара бібліотека вводу-виводу підтримується за допомогою заголовка <iostream.h>. Нова система вводу-виводу забезпечується заголовком <iostream>. У багатьох відносинах ці системи нічим не відрізняються, оскільки сучасна бібліотека просто є модифікованою й поліпшеною версією старої бібліотеки.Основні розходження між ними сховані від спостерігачів, оскільки ставляться до механізмів реалізації, а не до способів застосування засобів вводу-виводу.

З погляду програміста, між старою й новою системами вводу-виводу мови C++ є два розходження. По-перше, нова бібліотека має додаткові властивості й визначає кілька нових типів даних. Отже, стара бібліотека є підмножиною нової. Практично всі програми, написані з використанням старої бібліотеки вводу-виводу, без проблем компілюються новими компіляторами. По-друге, стара бібліотека вводу-виводу перебувала в глобальному просторі імен. (Нагадаємо, що простір імен **std** використовується всіма стандартними бібліотеками мови C++.) Оскільки ця бібліотека морально застаріла, у книзі використається тільки нова система вводу-виводу, хоча всі програми можна скомпілювати й зі старою бібліотекою.

Потоки

Система вводу-виводу мови C++, який її аналогу мові C, оперує потоками. П*отік* (stream) — це логічний пристрій, що одержує або передає інформацію. Потік пов'язаний з фізичним пристроєм вводу-виводу. Всі потоки функціонують однаково, хоча фізичні пристрої, з якими вони зв'язані, можуть бути різними.Оскільки всі потоки однакові, одна функція вводу-виводу може працювати з різними типами фізичних пристроїв.Наприклад, за допомогою однієї функції можна виводити дані як на принтер, так і на екран.

Класи потоків у мові C++

Як відомо, для системи вводу-виводу необхідний заголовок <iostream>. У цьому заголовку визначена досить складна ієрархія класів, що підтримують

операції вводу-виводу. Спочатку визначаються шаблонові класи вводу-виводу.Шаблоновий клас являє собою схему, у якій не уточнюється тип даних, якими вона оперує. Після визначення шаблонових класів можна створювати їхньої конкретизації. Стандарт мови C++ створює дві спеціалізації шаблонових класів вводу-виводу: одну для восьмибітових символів, а іншу - для розширених.

Система вводу-виводу мови C++ побудована на основі двох родинних, але різних ієрархій шаблонових класів. В основі першої ієрархії лежить клас Basic_streambuf, призначений для низькорівневого вводу-виводу. У звичайних додатках частіше застосовується ієрархія класів, побудована на основі класу basic_ios, що забезпечує високорівневі операції вводу-виводу, перевірку помилок й аналіз інформації про статус потоків

Вбудовані потоки в мові C++

На початку виконання програми мовою C++ автоматично відкриваються чотири потоки.

Потоки	***Значення***	***Пристрій за замовчуванням***
cin	Стандартне введення	Клавіатура
cout	Стандартний вивід	Екран
cerr	Стандартне виведення помилок	Екран
clog	Варіант потоку cerr, що буферизує	Екран

Потоки **cin, cout** і **cerr** відповідають потокам **stdin, stdout** й **stderr.**

За замовчуванням стандартні потоки використовуються для взаємодії з консоллю.Однак в операційних системах, що підтримують перенаправлення потоків вводу-виводу (таких як DOS, Unix, OS/2 й Windows), стандартні потоки можна зв'язати з іншими пристроями або файлами.

Крім того, мова C++ визначає чотири додаткових потоки: **win, wout, werr** й **wlog.** Це версії потоків для вводу-виводу розширених символів. Для подання розширених символів використовується тип **wchar_t** й 16-бітові значення. Як правило, розширені символи застосовуються для підтримки деяких природних мов.

Форматне уведення-виведення

Мова C++ дозволяє виконувати операції форматованого вводу-виводу. Наприклад, можна задати ширину поля, указати основу числення або визначити кількість цифр після десяткової крапки. Для форматування даних можна застосовувати два схожих, але різних способу. По-перше, можна прямо звернутися до членів класу **ios.** Зокрема,можна самостійно задавати різні опції форматування, визначені у середині класу **ios,** або викликати різноманітні функції-члени. По-друге, у виразах вводу-виводу можна використати спеціальні функції, які називаються *маніпуляторами* (manipulators).

Розглянемо спочатку засоби форматованого вводу-виводу за допомогою опцій і функцій — членів класу **ios.**

Форматування за допомогою членів класу ios

Кожен потік пов'язаний з набором опцій формату, керуючих поданням інформації. Клас **ios** повідомляє бітову маску за назвою **fmtflags,** у якій визначаються наступні значення. (З технічної точки зору ці значення визначені в класі **ios_base,** що є базовим стосовно класу **ios.)**

adjustfield	basefield	boolalpha	dec
fixed	floatfield	hex	internal
left	oct	right	scientific
showbase	showpoint	showpos	skipws
unitbuf	uppercase		

Ці значення використовуються для встановлення й скидання опцій форматування. При роботі зі старими компіляторами неможливо визначити перерахування **fmtflags. У** такому випадку опції формату кодуються за допомогою значень, що мають тип **long.**

Якщо встановлено опцію **skipws**, при уведенні даних з потоку роздільники (пробіли, знаки табуляції й символи переходу на новий рядок) ігноруються. Якщо цей опцію скинутий, роздільники враховуються.

Якщо встановлено опцію **left**, рядки виводу вирівнюються по лівому краю. Якщо встановлено опцію **right** - по правому краю. Якщо встановлено опцію **internal**, між знаком числа і його першою цифрою пробіли уставляються так, щоб число заповнило собою все поле виводу.Якщо жодна із цих опцій не встановлена, за замовчуванням виконується вирівнювання по правому краї.

За замовчуванням числові значення виводяться в десятковому виді.Однак основу системи числення можна змінити. Для виводу вісімкових чисел призначена опція **oct**. Встановлення опції **hex** дозволяє виводити числа в шістнадцятковому форматі.Вивід чисел у десятковому форматі забезпечується опцією **dec**.

Встановлення опції **showbase** дозволяє вивести на екран основу системи числення. Наприклад, при виводі шістнадцяткових чисел значення 1F буде відображене як Ox1F.

При виводі чисел у науковому форматі буква “e” за замовчуванням виводиться як рядкова. Крім того, буква “x” у вісімковому поданні чисел також вважається рядковою.Якщо необхідно вивести ці букви як прописні, варто встановити опцію **uppercase**.

Встановлення опції **showpos** дозволяє вивести знак перед позитивними числами.

Встановлення опції **showpoint** дозволяє виводити десяткову крапку й незначущі нулі при відображенні десяткових чисел.

Якщо встановлено опцію **scintiefic**, число виводиться в науковому форматі. Якщо встановлено опцію **fixed**, десяткове число виводиться у звичайному виді.Якщо жодна із цих опцій не встановлена, компілятор сам вибирає підходяще подання чисел.

Якщо встановлено опцію **unitbuf**, то після кожної операції вставки буфер очищається.

Опцію **boolalpha** дозволяє вводити й виводити булеві значення true й false.

Оскільки числа звичайно виводяться в десятковому, вісімковому й шістнадцятковому форматі, поля dec, oct й hex називають загальним ім'ям

basefield. Аналогічно поля left, right й internal називають **adjustfield**. Крім того, поля scientific й fixed поєднують загальним ім'ям **floatfield**.

Встановленняопції формату

Для встановлення опції використовується функція **setf ()**. Ця функція є членом класу **ios**. Вона має такий вигляд.

fmt flagssetf(fmt flags *flags);*

Дана функція повертає поточний стан опцій формату, відзначених параметром *flags* і встановлює їх. Наприклад, щоб установити опцію **showpos**, можна застосувати наступний оператор.

stream.setf (ios : : showpos) ;

Тут ім'я ***stream*** означає потік, на який ви хочете вплинути. Зверніть увагу на префікс ios:: перед опцією **showpos**. Він необхідний, оскільки опцію **showpos** є перелічувальною константою, визначеною в класі **ios**.

Наступна програма виводить на екран число 100, установлюючи опції **showpos** і **showpoint.**

```
#include<iostream>
using namespace std;
int main()
{
   cout.setf(ios::showpoint);
   cout.setf(ios::showpos);
   cout << 100.0; // Виводимо число +100.0
   return 0;
}
```

Варто пам'ятати, що функція **setf ()** є членом класу **ios** і впливає на потоки, створені цим класом. Отже, будь-який виклик функції **setf ()** пов'язаний з конкретним потоком.Сама по собі функція **setf ()** ніколи не викликається. Інакше кажучи, у мові C++ немає концепції глобального статусу формату. Кожен потік підтримує свій власний статус.

Хоча попередня програма є синтаксично правильною, її можна переписати й зробити більше ефективною.Замість декількох викликів функції setf() можна застосувати до її аргументів логічну операцію "АБО".Наприклад, що попередні виклики можна замінити одним.

cout.setf (ios : :showpoint | ios :: showpos);

Скидання опцій формату

Антиподом функції **setf ()** є функція **unsetf ().**Ця функція - член класу **ios**-використовується для скидання одного або декількох параметрів формату й має такий вигляд.

void unsetf (fmtflags *flags);*

Дана функція скидає опції, задані своїм параметром.(Всі інші опції зберігають свій колишній стан.)

Функція **unsetf ()** ілюструється наступною програмою. Спочатку вона встановлює опції uppercase й scientific, потім виводить число 100.12 у науковому

форматі.У цьому випадку науковий формат числа містить прописну букву "E".Після цього програма скидає опцію uppercase і знову виводить число 100.12 у науковому форматі, цього разу використовуючи малу літеру "e".

```
#include<iostream>
using namespace std;
int main()
{
   cout.setf (ios::uppercase | ios::scientific);
   cout << 100.12;  // Виводимо число 1.0012E+02
   cout.unsetf (ios::uppercase); // Скидаємо опцію uppercase
   cout << " \n" << 100.12; // Виводимо число 1.0012e+02
   return 0;
 }
```

Перевантажена форма функції setf()

Функція **setf**() має перевантажену форму.

fmt flagssetf (fmtflags *опції 1,* fmtflags *опції 2)* ;

У цій версії змінюються тільки опції, задані параметром *опції2.* Спочатку вони скидаються, а потім встановлюються у відповідності зі статусами опцій, заданих параметром *опції 1.* Зверніть увагу на те, що навіть якщо параметри *опції 1* і *опції 2* ставляться до різних опцій, змінюються тільки опції, задані параметром *опції 2.* Функція повертає попередній стан опцій.

```
#include<iostream>
using namespace std;
int main( )
{
   cout.setf(ios::showpoint | ios::showpos, ios::showpoint);
   cout << 100.0; // Виводить на екран число 100.0, а не +100.0
   return 0;
 }
```

Програма встановлює опцію **showpoint,** а не **showpos,** оскільки опцію **showpos** незазначений у другому параметрі.

```
#include<iostream>
using namespace std;
int main()
{
   cout.setf(ios::hex, ios::basefield);
   cout << 100; // Виводить на екран число 64
   return 0;
}
```

У цій програмі спочатку скидаються опції **basefield** (тобто опції **dec, oct** й **hex),** а потім установлюється опцію **hex.**

Стан опцій, заданих параметром *опції 1,* впливає тільки на опцій, заданих параметром *опції 2.* Наприклад, у наступній програмі перша спроба встановити опцію *showpos* виявляється невдалою.

```
// Ця програма не працює,
```

```
#include<iostream>
using namespace std;
int main()
{
   cout.setf(ios::showpos, ios::hex); // Помилка, опцію
                                      // showpos не встановлений.
   cout << 100 << '\n'; // Виводить на екран число 100, а не +100
   cout.setf(ios::showpos, ios::showpos); // Правильно
   cout << 100; // Тепер на екран виводиться число +100.
   return  0;
}
```

В більшості випадків для скидання опцій використовується функція unsetf(), а для встановлення опції викликається функція setf() з одним параметром. У більше складних випадках, наприклад, при виводі основи системи числення варто викликати функцію setf (fmtflags, fmtflags). Крім того, іноді програмісти створюють шаблон всіх опцій і в формату, що описує подання чисел, а потім намагаються змінити одну або дві опції. У цьом увипадку шаблон задають за допомогою параметра *опції 1*, а змінювані опції перераховують у параметрі *опції 2*.

Перевірка опцій форматування

Іноді необхідно точно визначити поточний стан формату, не змінюючи його опції. Для цього в класі ios використається функція-член flags(). Яка повертає поточний стан кожноїопції формату.Її прототип має такий вигляд.

fmt flagsflags();

У наступній програмі ця функція застосовується для встановленняопцій форматування, пов'язаних з потоком cout. Зверніть особливу увагу на функцію showflags (). Вона може виявитися корисною.

```
#include<iostream>
using namespace std;
void showflags ;
intmain()
{
     // Показати стан опцій формату,
     // передбачити за замовчуванням.
     showflags();
     cout.setf(ios::right | ios::showpoint | ios::fixed);
     showflags();
     return 0;
}
// Функція виводить на екран стан опцій формату.
void showflags()
{
     ios::fmtflags f;
     long i;
     f = (long) cout.flags(); // Одержати стан опцій.
     // Перевірити коженої опції
```

```
    for(i=0x4000; i; i = i>>l)
    if(i & f) cout << "1 "
    else cout <<« "0 ";
    cout << " \n";
}
```

Нижче показані результати роботи цієї програми.(Ці установки залежать від компілятора.)

1000001000000001 010001010010001

Встановлення всіх опцій

Функція **flags ()** має другу форму, що дозволяє встановлювати всі опції формату, пов'язані з потоком.Прототип цієї версії функції **flags ()** показаний нижче.

fmtflagsflags(fmtflags*f) ;*

У цій версії параметр *f* задає бітовий шаблон опцій формату, пов'язаних з конкретним потоком.Таким чином, змінюються відразу всі опції форматування.Функція повертає попередній стан опцій.

Ця версія функції **flags()**ілюструється наступною програмою.Спочатку вона створює бітову маску, у якій установлені опції **showpos**, **showbase**, **oct** й **right**. Всі інші опції скинуті. Потім для установки опцій форматування, пов'язаних з потоком **cout**, використовується функція **flags()**. Функція **show-flags()** виводить на екран стан опцій форматування.(Це та ж функція, що була визначена в попередньому прикладі.)

```
#include<iostream>
using namespace std;
void showflags();
int main()
{
    // Виводимо верб екран стан опцій формату,
    // передбачене за замовчуванням.
    showflags();
    // Опції showpos, showbase, oct й right установлені,
    // інші скинуті.
    long f = ios::showpos | ios::showbase | ios::oct | ios::right;
    cout.flags(f);  // Задаємо стан всіх опцій.
    showflags();
    return 0;
}
```

Застосування функцій width(), precision() і fill()

У класі **ios** передбачені три функції-члени, що дозволяють змінювати ширину поля виводу, точність і символ-заповнювач. Вони називаються **width()**, **precision()** і **fill ()** відповідно.Розглянемо кожну з них окремо.

За замовчуванням кількість позицій, які займає число при виводі, дорівнює кількості символів, з яких воно складається. Однак існує можливість змінити мінімальну ширину поля виводу, використовуючи функцію **width()**. Її прототип показаний нижче.

streamsize width(streamsize w);

Функція повертає попередню ширину поля, а її нове значення задається параметром *w*. У деяких реалізаціях ширина поля повинна задаватися при кожному виводі. Якщо цього не зробити, використовується її значення, передбачене за замовчуванням. Тип **streamsize** являє собою один з варіантів типу **int,** причому конкретний вибір залежить від компілятора.

Якщо число не цілком заповнює поле, воно доповнюється символом-заповнювачем (за замовчуванням - пробілом), щоб довжина числа збігалася із шириною поля.Якщо число не міститься в поле виводу, воно однаково виводиться повністю.Усікання чисел не виробляється.

При виводі дійсних чисел можна задавати кількість цифр після десяткової крапки (точність числа), використовуючи функцію **precision**(). Її прототип представлений нижче.

| streamsize precision(streamsize *p) ;*

Функція повертає старе значення, а нова кількість цифр після десяткової крапки задається параметром *p.* За замовчуванням після десяткової крапки виводиться 6 цифр. У деяких реалізаціях це значення варто задавати перед кожним виводом, інакше буде використовуватися точність, передбачена за замовчуванням.

Крімтого, якщо поле виводу незаповнено, воно автоматично доповнюється пробілами. Символ-заповнювач можна змінити, використавши функцію **fill().** Її прототип має такий вигляд.

char fill (char*ch) ;*

Функція **fill**() повертає старий символ-заповнювач, а новим заповнювачем стає символ *ch.*

Розглянемо програму, що ілюструє застосування цих функцій.

```
#include<iostream>
using namespace std;
int main ()
{
    cout.precision(4) ;
    cout.width(l0);
    cout << 10.12345 << "\n” //  потрібно cout<<fixed<<10.12345 <<endl;
    cout.fill(' *' ) ;
    cout.width(10);
    cout <<10.12345 << "\n"; // Виводить на екран *****10.12
    // Ширина поля виводу поширюється й на рядки
    cout.width(l);
    cout << "Hi!" << "\n"; // Виводить *******ні!
    cout.width(10);
    cout.setf(ios::left); // Вирівнює по лівому краї
    cout << 10.12345; // Виводить на екран 10.12*****
return 0;
}
```

Результат роботи цієї програми виглядає так.

```
10.12
*****10.12
*******ні!
```

10.12*****

Функції **width(), precision()** і**fill()** мають перевантажені версії, що одержують, але не відповідні параметри, що змінюють, потоку.

char fill () ;

streamsize width;

streamsize precision();

Застосування маніпуляторів формату

Інший спосіб зміни формату заснований на застосуванні спеціальних функцій, називаних *маніпуляторами* (manipulators). Їх можна включати в оператори вводу-виводу.Стандартні маніпулятори перераховані в табл. 20.1.

Таблиця 20.1. Маніпулятори

Маніпулятори	*Призначення*	*Ввід-вивід*
boolalpha	Установлює опцію boolalpha	Ввід-вивід
dec	Установлює опцію dec	Ввід-вивід
endl	Виводить символ переходу на новий рядок й очищає буфер	Вивід
ends	Виводить нульовий байт	Вивід
fixed	Установлює опцію fixed	Вивід
flush	Очищає буфер	
hex	Установлює опцію hex	Ввід-вивід
left	Установлює опцію left	Вивід
noboolalpha	Скидає опцію boolalpha	Ввід-вивід
noshowbase	Скидає опцію showbase	Вивід
noshowpoint	Скидає опцію	Вивід
noshowpos	showpointСкидає опцію	Вивід
noskipws	showpos	Вивід
nounitbuf	Скидає опцію skipws	Вивід
nouppercase	Скидає опцію unitbuf	Вихід
oct	Скидає опцію uppercase	Ввід-вивід
resetiosflags (fmtflags*f)*	Установлює опцію oct	Ввід-вивід
right	Скидає опції, зазначені параметром *f*	Вихід
scientific	Установлює опцію right	Вихід
setbase(int*base)*	Установлює опцію scientific	Ввід-вивід
setfill(int*ch)*	Задає основи системи числення, зазначені параметром *base*	Вихід
setiosflags (fmtflags f)	Задає символ-заповнювачів	Ввід-вивід
setprecision(int p)	Установлює опції, зазначені параметром *f*	Вихід
setw(int*w)*	Задає кількість цифр після десяткової крапки	Вихід
showbase	Задає ширину поля,	
showpoint	зазначену параметром *w*	Вихід

showpos	Установлює опцію showbase	Вихід
skipws	Установлює опцію showpoint	Вихід
unitbuf	Установлює опцію showpos	Вхід
uppercase	Установлює опцію skipws	Вихід
ws	Установлює опцію unitbuf	Вихід
	Установлює опцію uppercase	Вхід
	Ігнорує провідні роздільники	

Для доступу до маніпуляторів, що одержують параметри (наприклад, до функції setw()), необхідно включити в програму заголовок <iomanip>. Розглянемо приклад, що ілюструє застосування маніпуляторів.

```
#include<iostream>
#include<iomanip>
using namespace std;
int main() {
cout<<hex << 100 <<endl;
cout << setfill('?') << setw(10) << 2343.0;
return 0; }
```

Результат роботи програми наведений нижче.

```
164
??????2343
```

Зверніть увагу на те, як маніпулятори включаються в оператори виводу Крім того, якщо маніпулятор не має аргументів (наприклад, endl), дужки після нього не ставляться, оскільки його ім'я є адресою, переданою перевантаженому операторові "<<".

Для порівняння приведемо функціонально еквівалентну версію попередньої програми, у якій використаються функції - члени класу ios.

```
#include<iostream>
 #include<iomanip>
using namespace std;
int main() {
cout.setf(ios::hex, ios::basefield);
cout<< 100 << "\n";  // Число 100 у шістнадцятковому форматі
cout.fill('?'); cout.width(10); cout << 2343.0;
return 0; }
```

Як бачимо, основна перевага маніпуляторів над функціями - членами класу ios полягає в компактності коду.

Використовуючи маніпулятор **setiosflags()**, можна встановити відразу всі опції формату, пов'язані з потоком.Наприклад, у наступній програмі маніпулятор **setiosflags ()** застосовується для установки опціюів**showbase** й **showpos**.

```
#include<iostream>
#include<iomanip>
using namespace std;
intmain
{
    cout<<setiosflags(ios::showpos);
    cout<<setiosflags(ios::showbase);
```

```
    cout<< 123 << " " <<hex<< 123;
    return 0;
}
```

Маніпулятор **setiosflags**() і функція **setf**() виконують однакові операції. Маніпулятор **boolalpha** цікавіший. Він дозволяє вводити й виводити логічні значення, використовуючи ключові слова **true** й **false,** а не числа.

```
#include<iostream>
using namespace std;
int main()
{
    bool b;
    b = true;
    cout<< b << " " <<boolalpha<< b <<endl;
    cout<< "Введіть булеве значення: ";
    cin>>boolalpha>> b;
    cout<< "Ви ввели наступне значення:  " << b;
    return 0;
}
```

Розглянемо приклад її роботи.

```
1 true
Введіть булеве значення:  false
Ви ввели наступне значення:  false
```

Переповення типів

Переповнення

Питання: *“Що станеться, якщо ми спробуємо використовувати значення поза діапазону допустимих значень?”*. Відповідь: *“Переповнення”*.

Переповнення (англ. **“overflow”**) трапляється при втраті біт через те, що змінній не було виділено достатньо пам’яті для їх зберігання.

Дані зберігаються в бінарному (двійковому) форматі і кожен біт може мати тільки 2 можливих значення (0 або 1).

Приклади переповнення

Розглянемо змінну unsigned, яка складається з 4 біт.

Але що станеться, якщо ми спробуємо присвоїти значення, яке займає більше 4 біт? Правильно! Переповнення. Наша змінна зберігатиме тільки 4 найменш значущих (ті, що праворуч) біти, а всі інші біти — загубляться.

Наприклад, якщо ми спробуємо помістити число 21 в нашу 4-бітну змінну:

Число 21 займає 5 біт (10101). 4 біти справа (0101) помістяться в змінну, а крайній лівий біт (1) просто загубиться. Тобто наша змінна міститиме 0101, що дорівнює 101 (нуль спереду не рахується), а це вже число 5, а не 21.

Тепер давайте розглянемо приклад на практиці (тип short займає 16 біт):

```
1       #include <iostream>
2
3       int main()
4       {
```

```
5       unsigned short x = 65535; // найбільше значення, яке може зберігати 16-бітна змінна unsigned
6       std::cout << "x was: " << x << std::endl;
7       x = x + 1; // 65536 - це число більше нашого максимально допустимого числа з діапазону допустимих значень. Отже, відбудеться переповнення, тому що змінна x не може зберігати 17 біт
8       std::cout << "x is now: " << x << std::endl;
9       return 0;
10  }
```

Результат виконання програми:

```
x was: 65535
x is now: 0
```

Що сталося? А сталося переповнення, оскільки ми спробували присвоїти змінній x значення, яке вона не може містити.

Для тих, хто хоче знати більше: Число 65 535 в двійковій системі числення представлено як 1111 1111 1111 1111. 65 535 — це найбільше число, яке може зберігати 2-байтова (16 біт) цілочисельна змінна unsigned, оскільки це число використовує всі 16 біт. Коли ми додаємо 1, то отримуємо число 65 536. Число 65 536 представлено в двійковій системі як 1 0000 0000 0000 0000 і займає 17 біт! Отже, найголовніший біт (яким є 1) втрачається, а всі 16 біт праворуч — залишаються. Комбінація 0000 0000 0000 0000 відповідає десятковому 0, що і є нашим результатом.

Аналогічним чином, ми отримаємо переповнення, використавши число, менше мінімального з діапазону допустимих значень:

```
1   #include <iostream>
2
3   int main()
4   {
        unsigned short x = 0; // найменше значення, яке 2-байтова змінна unsigned може містити
5
6       std::cout << "x was: " << x << std::endl;
7       x = x - 1; // переповнення!
8       std::cout << "x is now: " << x << std::endl;
9       return 0;
10  }
```

Результат виконання програми:

```
x was: 0
x is now: 65535
```

Таким чином: max + 1 = min, а min - 1 = max

Переповнення призводить до втрати інформації, а це ніколи не є добре. Якщо є хоча б найменша підозра або припущення, що значенням змінної може бути число, яке виходить за рамки допустимих значень, то відразу використовуйте більший тип даних!

Правило: Ніколи не допускайте можливості виникнення переповнення в ваших програмах!

Чому розмір всіх цілочисельних типів не є фіксованим?

Якщо говорити в загальному, то все почалося ще з мови Сі, коли продуктивність мала першочергове значення. У мові Сі навмисно залишили розмір цілочисельних типів нефіксованим для того, щоб компілятор міг самостійно підібрати найбільш підходящий розмір для певного типу даних в залежності від комп'ютерної архітектури.

Цілочисельні типи фіксованого розміру

Щоб вирішити питання кросплатформенності, в мові C++ додали набір цілочисельних типів фіксованого розміру, які гарантовано мають один і той же розмір на будь-якій архітектурі:

Назва	Тип	Діапазон значень
int8_t	1 байт signed	від -128 до 127
uint8_t	1 байт unsigned	від 0 до 255
int16_t	2 байти signed	від -32 768 до 32 767
uint16_t	2 байти unsigned	від 0 до 65 535
int32_t	4 байти signed	від -2 147 483 648 до 2 147 483 647
uint32_t	4 байти unsigned	від 0 до 4 294 967 295
int64_t	8 байт signed	від -9 223 372 036 854 775 808 до 9 223 372 036 854 775 807
uint64_t	8 байт unsigned	від 0 до 18 446 744 073 709 551 615

Починаючи з C++11 доступ до цих типів здійснюється через підключення **заголовку** cstdint (знаходяться ці типи даних в **просторі імен std**). Розглянемо приклад на практиці:

```
1   #include <iostream>
2   #include <cstdint>
3
4   int main()
5   {
6       std::int16_t i(5); // пряма ініціалізація
7       std::cout << i << std::endl;
8       return 0;
9   }
```

Оскільки цілочисельні типи фіксованого розміру були додані ще до C++11, то деякі старі компілятори надають доступ до них через підключення заголовку stdint.h.

Якщо ваш компілятор не підтримує cstdint чи stdint.h, то ви можете завантажити кросплатформний заголовковий файл **pstdint.h**. Просто підключіть його до вашого проекту, і він самостійно визначить цілочисельні типи фіксованого розміру для вашої системи/архітектури.

Попередження щодо std::int8_t і std::uint8_t

З певних причин в мові C++ більшість компіляторів визначають і обробляють типи int8_t і uint8_t ідентично типам char signed і char unsigned, але це відбувається далеко не у всіх випадках. Отже, **std::cin** і **std::cout** можуть працювати не так, як ви очікуєте. Наприклад:

```
1   #include <iostream>
2   #include <cstdint>
3
4   int main()
5   {
```

```
6           std::int8_t myint = 65;
7           std::cout << myint << std::endl;
8
9           return 0;
10      }
```

На більшості архітектур результат виконання цієї програми наступний:

A

Тобто вищенаведена програма опрацьовує myint як змінну типу char. Однак на деяких комп'ютерах результат може бути наступним:

65

Тому ідеальним варіантом буде уникати використання std::int8_t і std::uint8_t взагалі (використовуйте замість них std::int16_t або std::uint16_t). Однак, якщо ви використовуєте std::int8_t або std::uint8_t, то ви повинні бути обережні з будь-якою функцією, яка може інтерпретувати std::int8_t або std::uint8_t в символьний тип даних, замість цілочисельного (наприклад, з об'єктами std::cin і std::cout).

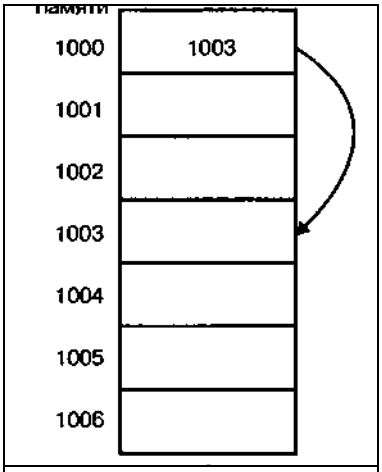
Правило: Уникайте використання std::int8_t і std::uint8_t. Якщо ви використовуєте ці типи, то будьте уважні, так як в деяких випадках вони можуть оброблятися як тип char.

Недоліки цілочисельних типів фіксованого розміру

Цілочисельні типи фіксованого розміру можуть не підтримуватися на певних архітектурах (де вони не мають можливості бути представлені). Також ці типи можуть бути менш продуктивними, ніж фундаментальні типи даних, на певних архітектурах.

Використана література

1. Г. Шилдт, Полный справочник по C++, 4-е видання, в-во «Вильямс», 2006
2. Х.М.Дейтел, Как програмировать на C++, 4-е видання, в-во «Бином-Пресс» 2009.
3. Р. Лафоре, Объектно-ориентированное программирование в C++, в-во «Питер», 2004, с 924.



Мал. 1

Існує три причини, по яких неможливо написати гарну програму мовою C/C++ без використання вказівників. По-перше, вказівники дозволяють функціям змінювати свої аргументи. По-друге, за допомогою вказівників здійснюється динамічний розподіл пам'яті. І, по-третє, вказівники підвищують ефективність багатьох процедур. Як ми побачимо надалі, у мові C++ вказівники мають додаткові переваги.

Вказівники - один із самих потужних й, у той же час, самих небезпечних засобів мови C/C++. Наприклад, неініціалізовані вказівники (або вказівники, що містять невірні адреси) можуть знищити операційну систему комп'ютера. І, що ще гірше, неправильне використання вказівників породжує помилки, які вкрай важко виявити.

Оскільки вказівники настільки важливі й небезпечні, ми розглянемо їх досить докладно.

Вказівник (pointer) — це змінна, у якій зберігається адреса іншого об'єкта (як правило, інший змінної). Наприклад, якщо одна змінна містить адресу іншої змінної, говорять, що перша змінна *посилається* (point) на другу. Ця ситуація показана на мал. 1.

Вказівники Змінна, що зберігає адресу комірки пам'яті, повинна бути оголошена як вказівник. Оголошення вказівника складається з імені базового типу, символу * й імені змінної. Загальна форма цього оголошення така.

*тип_вказівника * ім'я_вказівника;*

Тут *тип_вказівника* означає базовий тип вказівника. Ним може бути будь-який допустимий тип.

Базовий тип вказівника визначається типом змінної, на яку він може посилатися. З формальної точки зору вказівник будь-якого типу може посилатися на будь-яке місце в пам'яті. Однак операції адресної арифметики тісно пов'язані з базовим типом вказівників, тому дуже важливо правильно їх оголосити. (Адресна арифметика обговорюється нижче.)

Оператори для роботи з вказівниками

Як відомо, існують два спеціальних оператори для роботи з вказівниками: оператор розіменування вказівника * й оператор одержання адреси &. Оператор & є унарним і повертає адресу свого операнда. (Не забувайте: унарні оператори мають тільки один операнд.) Наприклад, оператор присвоювання

m = &ccount,-

записує в вказівник m адресу змінної **count.** Ця адреса комірки пам'яті, що займає змінна **count**. Адреса й значення змінної ніяк не зв'язані один з одним. Оператор & означає "адресу".

Щоб глибше розібратися в механізмі, що лежить в основі попереднього оператора присвоювання, припустимо, що змінна **count** зберігається в осередку з номером 2000.

Оператор розіменування вказівника * є антиподом оператора &. Цей унарний оператор повертає значення, що зберігається по зазначеній адресі. Наприклад, якщо вказівник m містить адреса змінної **count,** те оператор присвоювання

q = *m;

помістить значення **count** у змінну q. Отже, змінна q стане рівна 100, оскільки саме це число записане в осередку 2000, адреса якої зберігається в вказівнику m. Символ * можна інтерпретувати як "значення, що зберігається за адресою". У цьому випадку попередній оператор означає: "присвоїти змінній q значення, що зберігається за адресою m".

Пріоритет операторів & й * вище, ніж пріоритет всіх арифметичних операторів, за винятком унарного мінуса.

Необхідно мати гарантії, що вказівник завжди посилається на змінну правильного типу. Наприклад, якщо в програмі оголошений вказівник на змінну цілого типу, компілятор думає, що адреса, що у ньому зберігається, ставиться до змінного типу **int,** незалежно від того, чи не так це насправді. Оскільки вказівнику можна привласнити будь-яку адресу, наступна програма буде скомпільована без помилок, але бажаного результату не дасть.

```
#include <iostream>
using namespace std;
int main(void) {
double x = 100.1, y;
int  *p;
/* Наступний оператор змусить вказівник p (який маї цілий тип) що посилатися на змінну типу double. */
```

```
p = &x;
/* Наступний оператор працює не так, як задумано. */
y = *p;
cout<<y; /* Не виводить число 100.1 */
return 0; }
```

Ця програма ніколи не присвоїть змінній y значення змінної x. Оскільки вказівник p є целочисленным, у змінну в будуть скопійовані лише 4 байт (оскільки цілі числа займають 4 байт), а не 8.

Вирази, що містять вказівники

Як правило, вирази, що містять вказівники, підкоряються загальноприйнятим правилам. Однак у них є свої особливості.

Присвоювання вказівників

Вказівник можна присвоїти іншому вказівнику. Розглянемо приклад.

```
#include <iostream>
using namespace std;
int main() {
int x;
int *pl, *p2;
pl = &x;
p2 = p1;
cout<<p2; /* Виводить адресу змінної x, а не її значення! */
return 0; }
```

Тепер на змінну x посилаються обидва вказівники p1 і p2.

Адресна арифметика

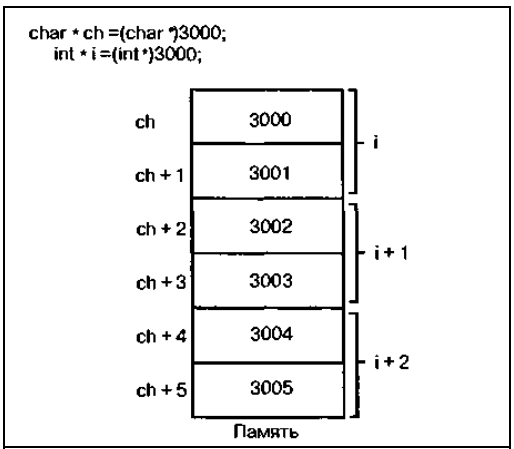
До вказівників можна застосовувати тільки дві арифметичні операції: додавання й віднімання. Припустимо, що вказівник p1 посилається на змінну цілого типу, розміщену за адресою 2000. Крім того, будемо вважати, що цілі числа займають 2 байт у пам'яті комп'ютера. Після обчислення виразу

```
p1++
```

змінна p1 буде дорівнює не 2001, а 2002, оскільки при збільшенні вказівника p1 на одиницю він посилається на наступне ціле число. Це ж стосується й оператора декрементації. Наприклад, якщо вказівник p1 дорівнює 2000, вираз

```
p1--
```

присвоїть вказівнику p1 значення 1998.



Мал. 2

Узагальнюючи наведений вище приклад, сформулюємо наступні правила адресної арифметики. При збільшенні вказівник посилається на осередок, у якій зберігається наступний елемент базового типу. При зменшенні він посилається на попередній елемент. Для вказівників на символи зберігаються правила "звичайної" арифметики, оскільки розмір символів дорівнює 1 байт. Всі інші вказівники збільшуються або зменшуються на довжину відповідних змінних, на які вони посилаються. Це гарантує, що вказівники завжди будуть посилатися на елемент базового типу Описана вище концепція проілюстрована на мал. 2.

Дії над вказівниками не обмежуються операторами інкрементації й декрементації. Наприклад, до них можна додавати й цілі числа. Вираз

```
pi = pi + 12;
```

зміщає вказівник pi на 12-й елемент базового типу, розташований після поточної адреси.

Крім того, вказівники можна віднімати. Це дозволяє визначити кількість об'єктів базового типу, розташованих між двома вказівниками. Всі інші арифметичні операції заборонені.

Порівняння вказівників

Вказівники можна порівнювати між собою. Наприклад, в прикладі нижче порівнювання вказівників p і q, є зовсім правильним.

```
if(p<q) cout<<"Вказівник p містить менше адресу, ніж; вказівник q;
```

Як правило, вказівники порівнюються між собою, коли вони посилаються на той самий об'єкт, наприклад масив.

Вказівники й масиви

Вказівники й масиви тісно зв'язані між собою. Розглянемо наступний фрагмент програми.

```
char  str[80],  *рш;
pi  =  Str;
```

Тут вказівнику рі привласнена адреса першого елемента масиву str. Щоб одержати доступ до п'ятого елемента цього масиву, варто виконати один із двох операторів:

```
str[4]
```

або

```
*(pl+4)
```

Обоє оператора повернуть значення п'ятого елемента масиву str. Нагадаємо, що індексація масивів починається з нуля, тому індекс п'ятого елемента масиву str дорівнює 4. Крім того, п'ятий елемент масиву можна одержати, додавши 4 до вказівника рі, що спочатку посилається на перший елемент.

Цей приклад можна узагальнити. По суті, у мові C/C++ існують два способи звертання до елементів масиву: індексація й адресна арифметика. Хоча індексація масиву наочніше, адресна арифметика іноді виявляється ефективніше.

У наступному фрагменті програми наведені два варіанти функції putstr(), що ілюструє доступ до елементів масиву. У першому варіанті використається індексація, а в другому - адресна арифметика. Функція putstr призначена для посимвольного запису рядка в стандартний потік виводу.

```
/* Індексація вказівника. */
void putstr(char *s) <
register int t;
for(t=0; s[t]; ++t) putchar(s[t]); }

/* Доступ за допомогою вказівника. */
void putstr(char *s)
{   while(*s) putchar(*s++);
 }
```

Більшість професійних програмістів порахують другу версію більш зрозумілою й наочною. На практиці саме цей спосіб доступу до елементів масиву розповсюджений найбільше широко.

Масиви вказівників

Як і всі звичайні змінні, вказівники можна поміщати в масив. Оголошення масиву, що складає з 10 цілочисельних вказівників, виглядає в такий спосіб.

```
int   *x[10];
```

Щоб присвоїти адреса цілочисленной змінної var третьому елементу масиву вказівників, потрібно виконати оператор

```
x[2]   =  &var;
```

Щоб витягти значення змінної var, використовуючи вказівник x[2], необхідно його розіменувати:

```
*x[2]
```

Масив вказівників передається у функцію як звичайно - досить указати його ім'я як параметр. У наступному фрагменті коду на вхід функції надходить масив вказівників.

```
void display_array (int   *q[])
{
int   t ;
for(t=0;   t<10;    t++)
cout<<*q[t];
}
```

Нагадаємо, що змінна q не є вказівником на цілочисельні змінні, її варто інтерпретувати як вказівник на масив цілочисельних вказівників Отже, необхідно оголосити, що параметр q являє собою масив цілочисельних вказівників. Це оголошення зроблене в заголовку функції.

Масиви часто містять вказівники на рядки. Спробуємо створити функцію, що виводить на екран повідомлення про помилку із заданим номером.

```
void svntax_error(int num)
static char *err[] = { "Неможливо відкрити файл\n", "Помилка при читанні\n", "Помилка при записі\n", "Відмова апаратури"};
cout<<  err[num]; }
```

Вказівники на рядки знаходяться в масиві **err.** Вивід відбувається усередині функції **syntax_error** () і виводить на екран рядок із зазначеним номером. Наприклад, якщо параметр num дорівнює 2, на екрані з'явиться повідомлення "Помилка при записі".

Відзначимо, що аргумент командного рядка **argv** також являє собою масив вказівників на символьні змінні (див. главу 3).

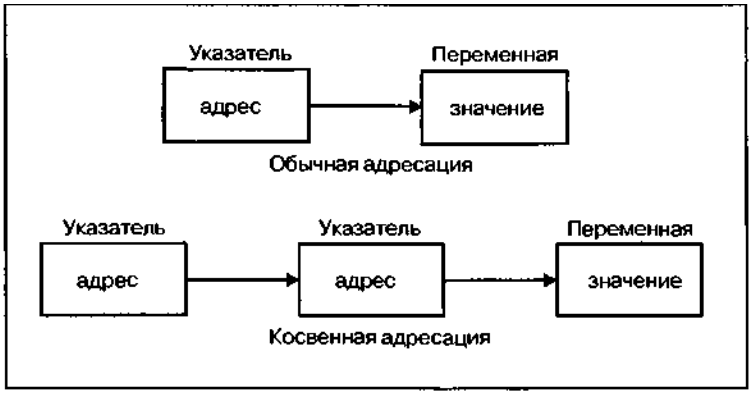


Рис. 5.3. Обычная и косвенная адресация

Непряма адресація

Іноді вказівник може посилатися на інший вказівник, що, у свою чергу, містить адресу звичайної змінної. Така схема називається *непрямою адресацією* (multiple inderection), або *вказівником на вказівник* (pointer to pointer). Застосування непрямої адресації знижує наочність програми. Концепція непрямої адресації проілюстрована на мал. 5.3. Як бачимо, звичайний вказівник зберігає адресу об'єкта, що містить необхідне значення. У випадку непрямої адресації один вказівник посилається на інший вказівник, а той, у свою чергу, містить адресу об'єкта, у якому записане потрібне значення.

Глибина непрямої адресації не обмежена, однак майже завжди можна обійтися "вказівником на вказівник". Більше громіздкі схеми адресації складніше зрозуміти, отже, імовірність помилок при їхньому використанні різко зростає.

Змінна, що представляє собою вказівник на вказівник, оголошується в такий спосіб: перед її ім'ям записується додаткова зірочка. Наприклад, у наступному операторі змінна **newbalance** оголошується як вказівник на вказівник на число із плаваючою крапкою.

```
float **newbalance;
```

Варто пам'ятати, що змінна **newbalance** не є вказівником на число із плаваючою крапкою, вона лише посилається на вказівник цього типу.

Щоб витягти значення змінної за допомогою непрямої адресації, не обходи мо двічі застосувати оператор рзіменування.

```
#include <iostream>
using namespace std;
int main(void) {
int x,   *p,   **q;
x = 10; p = &x; q =  &p;
cout<< **q;  /*  Вивід числа x  */
return  0; }
```

Тут змінна p оголошена як цілочисельний вказівник, а змінна q являє собою вказівник на цілочисельний вказівник. В результаті виконання на екран виведеться число 10.

Ініціалізація вказівників

Якщо нестатичний локальний вказівник оголошений, але не ініціалізований, його значення залишається невизначеним. (Глобальні й статичні локальні вказівники автоматично ініціалізуються нулем.) При спробі застосувати вказівник, що містить невизначену адресу, може зруйнуватися як програма, так і вся операційна система - гірше не придумаєш!

При роботі з вказівниками більшість професійних програмістів мовою C/C++ дотримуються наступного правила: вказівник, що не посилається на конкретну комірку пам'яті, повинен бути рівний нулю. По визначенню будь-який нульовий вказівник ні на що не посилається й не повинен використовуватися. Однак це ще не гарантує безпеки. Використання нульового вказівника - усього лише загальноприйняте правило. Це зовсім не правило, обумовлене мовою C або C++. Наприклад, якщо нульовий вказівник помістити в ліву частину оператора присвоювання, ризик руйнування програми й операційної системи залишається високим.

Оскільки передбачається, що нульовий вказівник в обчисленнях не використається, його можна застосовувати для підвищення наочності й ефективності програм. Наприклад, його можна використати як ознаку кінця масиву, що містить вказівники. Це дозволить запобігти виходу за межі допустимого діапазону. Подібна ситуація ілюструється на прикладі функції **search().**

```
/*  Пошук імені */
int search(char  *p[],   char  *name)
{
register int t;
for(t=0;   p[t];   ++t)
if(!strcmp(p[t], name))   return  t;
return -1;   /*  Ім'я не знайдене  */ }
```

Цикл **for** усередині функції **search** () виконується доти, поки не буде знайдене необхідне ім'я, або не виявиться нульовий вказівник. Як тільки нульовий вказівник

буде виявлений, умова циклу стане помилковим, і програма передасть керування наступному операторові.

Програмісти на C/C++ часто ініціалізують рядки. Приклад такої ініціалізації наведений нижче:

```
char *p = "Привіт";
```

Як бачимо, вказівник p не є масивом. Спробуємо зрозуміти, чому можливий такий спосіб ініціалізації. Всі компілятори мови C/C++ створюють *таблицю рядків* (string table), у якій зберігаються строкові константи, які використовуються в програмі. Отже, поередній оператор привласнить вказівнику p адресу рядкової константи "Привіт", записаної в таблицю рядків. Вказівник p використається в програмі як звичайний рядок (однак змінювати його небажано). Проілюструємо це наступним прикладом.

```
#include <iostream>
# include <cstring>
using namespace std;
char *p = "Привіт";
int main(void) {
register int t;
/* Виводимо рядок у прямому й зворотному порядку. */
cout<<p;
for(t=strlen(p)-l; t>-l; t-) cout<<p[t];
return 0; }
```

З формальної точки зору в стандарті мови C++ рядковий літерал має тип **const char** *. Однак у мові C++ передбачене автоматичне перетворення в тип **char** *. Таким чином, розглянута вище програма є абсолютно правильною. І все-таки ця особливість мови вважається небажаної, тому це перетворення не слід застосовувати. Створюючи нові програми, рядкові літерали необхідно дійсно вважати константами, а оголошення вказівника p записувати в такий спосіб.

```
const char *p = "Привіт";
```

Вказівники на функції

Особливо малозрозумілим, хоча й діючому механізмі мови C++ є *вказівники на функції* (function pointer). Незважаючи на те що функція не є змінною, вона розташовується в пам'яті, і, отже, її адреса можна присвоювати вказівнику. Ця адреса вважається точкою входу у функцію. Саме вона використається при її виклику. Оскільки вказівник може посилатися на функцію, її можна викликати за допомогою цього вказівника. Це дозволяє також передавати функції іншим функціям як аргументи.

Адреса функції задається її ім'ям, зазначеним без дужок й аргументів. Щоб розібратися в цьому механізмі, розглянемо наступну програму.

```
#include <iostream>
#include <cstring>
using namespace std;
void check(char *a, char *b, int (*cmp)(const char *, const char *));
int main(void) {
char s1[80] , s2[80];
int (*p)(const char *, const char *);
p = strcmp;
gets(s1); gets(s2) ;
check(s1, s2, p);
return 0; }
void check(char *a, char *b, int (*cmp)(const char *, const char *))
{
cout<<"Перевірка рівності"<<endl;
if(!(*cmp)(a, b)) cout<"Рівні";
else cout<<"Не рівні"; }
```

При виклику функція **check**() одержує два вказівники на символьні змінні й вказівник на функцію. Відповідні аргументи оголошені в її заголовку. Зверніть увагу на те, як оголошений вказівник на функцію. Цю форму оголошення варто застосовувати для будь-яких вказівників на функції, незалежно від того, який тип мають їхні аргументи й повертають значення, що. Оголошення *cmp знаходиться в дужках для того, щоб компілятор правильно його інтерпретував. Вираз

```
(*cmp)(a, b)
```

усередині функції **check**() означає виклик функції strcmp, на яку посилається вказівник cmp, з аргументами a й b. Дужки як і раніше необхідні. Це один зі способів викликати функцію за допомогою вказівника. Альтернативний виклик виглядає так:

```
cmp(a, b);
```

Перший спосіб застосовується частіше, оскільки він дозволяє явно продемонструвати, що функція викликається через вказівник. (Інакше кажучи, читач програми відразу побачить, що змінна cmp є вказівником, а не ім'ям функції.) Проте ці два способи еквівалентні.

Зверніть увагу на те, що функцію **check** () можна викликати, безпосередньо вказавши функцію strcmp() в якості її аргументу.

```
check(sl,  s2,  strcmp);
```

У цьому випадку відпадає необхідність у додатковому вказівнику на функцію.

Може виникнути закономірне питання: навіщо викликати функції за допомогою вказівників? На перший погляд, це лише ускладнює програму, не надаючи переваг. Проте іноді вигідніше викликати функції через вказівники й навіть створювати масиви вказівників на функції. Розглянемо як приклад синтаксичний аналізатор - складову частину компілятора, що обчислює вирази. Він часто викликає різні математичні функції (синус, косинус, тангенс й т.д.), засобу вводу-виводу або функції доступу до ресурсів системи. Замість створення великого оператора switch, у якому довелося б перераховувати всі ці функції, можна створити масив вказівників на них. У цьому випадку до функцій можна було б звертатися по індексу. Щоб оцінити ефективність такого підходу, розглянемо розширену версію попередньої програми. У цьому прикладі функція check() перевіряє на рівність рядка, що складаються з букв або цифр. Для цього вона просто викликає різні функції, що виконують порівняння.

```
#include <iostream>
#include <cstring>
using namespace std;
void check(char *a, char *b, int (*cmp)(const char *, const char *));
int numcmp(const char *a, const char *b) ;
int main(void) {
char s1[80], s2[80];
gets(s1); gets(s2);
if(isalpha(*s1))
check(sl, s2, strcmp); else
check(sl, s2, numcmp);
return 0; }
void check(char *a, char *b, int (*cmp)(const char *, const char *)) {
cout<<"Перевірка рівності.\n";
if(!(*cmp)(a, b)) cout<<"Рівні";
else cout<<"Не рівні"; }
int numcmp(const char *a, const char *b) {
if (atoi (a) ==atoi (b) ) return 0;
else return 1;}
```

Якщо користувач уведе рядок, що складається з букв, функція **check ()** викличе функцію **strcmp**, одержавши вказівник на неї. У противному випадку буде переданий вказівник на функцію **numcmp ().** Отже, у різних ситуаціях функція **check ()** може викликати різні функції.

Оператори динамічного розподілу пам'яті

У мові C++ передбачені два оператори динамічного розподілу пам'яті: **new** й **delete**. Ці оператори виділяють і звільняють пам'ять у ході виконання програми. Динамічний розподіл пам'яті являє собою важливу частину практично всіх реальних програм. У мові C++ є альтернативний спосіб динамічного розподілу пам'яті, заснована на функціях **malloc ()** і **free()**. Вони збережені для того, щоб досягти сумісності з мовою C. Однак у програмах мовою C++ варто застосовувати оператори **new** й **delete**, оскільки вони мають ряд важливих переваг.

Оператор **new** виділяє область пам'яті й повертає вказівник на її перший клітинку. Оператор **delete** звільняє пам'ять, виділену раніше за допомогою оператора **new**. Загальний вид цих операторів такий.

вказівник = new *тип;*
delete *вказівник;*

Тут *вказівнику* присвоюється адреса першої клітинки виділеної області пам'яті, розмір якої визначається *типом.*

Оскільки обсяг виділеної пам’яті обмежений, пам'ять може виявитися вичерпаною. У цьому випадку оператор **new** згенерує виняткову ситуацію **bad_alloc,** визначену в заголовку **<new>.** Програма повинна перехопити цю ситуацію й обробити її. Якщо в програмі не передбачена обробка виняткової ситуації **bad_alloc,** її виконання буде перервано.

Дія оператора **new** у виняткових ситуаціях визначено стандартом мови C++. Проблема полягає в тім, що не всі компілятори повністю відповідають стандарту. У ранніх версіях мови C++ оператор **new** у випадку відмови повертав нульовий вказівник. Пізніше він став генерувати виняткову ситуацію. У підсумку було вирішено, що за замовчуванням оператор **new** повинен генерувати виняткову ситуацію, однак повертати нульовий вказівник також не забороняється. Отже, у кожному конкретному випадку поводження оператора **new** регламентується розроблювачами компілятора. Зрозуміло, зрештою всі компілятори будуть змушені дотримуватися стандарту, але поки варто орієнтуватися на їхню документацію.

Всі приклади, описані в нашій книзі, написані у відповідності зі стандартом. Якщо ваш компілятор не повністю відповідає стандарту, програми варто змінити.

Розглянемо приклад, у якому виділяється динамічна пам'ять для цілочисельної змінної.

```
#nclude <iostream>
#include <new>
using namespace std;
int main() {
int *p;
try {
p = new int; // Виділити пам'ять для цілочисельної змінної }
catch (bad_alloc xa) {
cout << "Виняткова ситуація\n";
return 1; }
*p = 100;
cout « "За адресою " « p « " ";
cout « "записане значення " « *p « endl;
delete p;
return 0; }
```

Ця програма присвоїть змінній p адресу динамічної пам'яті, у якій може зберігатися цілочислова змінна. Потім у цю область пам'яті записується число 100, а її вміст виводиться на екран. На закінчення програма звільняє виділену пам'ять. Врахуйте, якщо ваш компілятор реалізує оператор **new** так, що у випадку відмови він повертає нульовий вказівник, цю програму прийдеться модифікувати.

Оператор **delete** варто застосовувати тільки до результату оператора new. В іншому випадку можуть виникнути проблеми, наприклад, крах операційної системи.

Оператори **new** й **delete** аналогічні функціям **malloc ()** і **free ()**, але в порівнянні з ними володіють декількома перевагами. По-перше, оператор new автоматично виділяє кількість пам'яті, необхідна об'єкту зазначеного типу. Оператор **sizeof** тепер застосовувати не слід. Оскільки розмір виділюваної пам'яті обчислюється автоматично, помилки виключені. По-друге, оператор **new** автоматично повертає вказівник заданого типу. Тепер не потрібно виконувати приведення типу, як це було при використанні функції **malloc ()**. Оператори **new** й **delete** можна перевантажувати, що дозволяє створювати налаштовувані системи, для виділення динамічної пам'яті.

Хоча формальних правил на цей рахунок не існує, оператори **new** й **delete** не слід змішувати з функціями **malloc ()** і **free ()** у рамках однієї й тієї ж програми, оскільки вони можуть виявитися несумісними.

Ініціалізація виділюваної пам'яті

Виділювану пам'ять можна проініціалізувати заданим значенням. Для цього варто вказати початкове значення після імені типу в операторі new. Загальний вид оператора new у цьому випадку виглядає в такий спосіб.

вказівник = new *тип* *(початкове_значення)*

Зрозуміло, тип початкового значення повинен бути сумісним з типом даних, для яких виділяється пам'ять.

Розглянемо програму, що записує у виділену пам'ять число 87.

```
#include <iostream>
#include <new>
using namespace std;
int main() {
int *p;
try {
p = new int (87); // ініціалізуємо числом 87 }
catch (bad_alloc xa) {
cout « "Виняткова ситуація\n";
return 1; }
cout « "За адресою " « p << " ";
cout « "записане число " « *p « "\n";
delete p;
return 0; }
```

Виділення пам'яті для масивів

За допомогою оператора new можна виділяти пам'ять для масивів. У цьому випадку оператор **new** має такий вигляд.

вказівник = new *тип_масиву [розмір]* ;

Тут розмір задає кількість елементів розташовуваного масиву.

Для звільнення пам'яті, зайнятий масивом, застосовується наступна форма оператора **delete**:

delete [] *вказівник;*

Наприклад, що випливає програма виділяє пам'ять для цілочисельного масиву, що складає з 10 елементів.

```
#include <iostream>
#include <new> using namespace std;
int main() {
    int *p, i;
    try {
        p = new int [10]; /* Виділяємо пам'ять для цілочисленного масиву, що складає з 10
елементів*/
    } catch (bad_alloc xa) {
        cout << "Виняткова ситуація\n";
        return 1; }
    for(i=0; i<10; i++ ) p[i] = i;
    for(i=0; i<10; I++) cout << p[i] << " ";
    delete [] p; // Звільняємо пам'ять, зайняту масивом
    return 0; }
```

Зверніть увагу на оператор **delete**. Як уже вказувалося, при звільненні пам'яті, виділеної для масиву за допомогою оператора **new**, розмір масиву не вказується. (Як буде показано в наступному розділі, ця властивість особливо важливо при роботі з масивами об'єктів.)

При створенні динамічного багатовимірного масиву необхідно в операції **new** вказати всі його розмірності (перший може бути змінною), наприклад:

int n = 5; *// n — количество строк*

int **m = (int **) new int [n][5];.

Розглянемо більш універсальний і безпечний спосіб виділення динамічної пам’яті під двовимірний масив, коли обидві його розмірностізадаються на етапі виконання програми. Наприклад, розподіл динамічної пам’яті для матриці, що має **n** рядків і **m** стовпців та елементи цілоготипу, можна здійснити так:

int n, m;

cout << ” Введіть кількість рядків та стовпців: “;

cin >> n >> m;

int **a = new int *[n]; — оголошення змінної тип «покажчи на покажчик на int» і виділення пам’яті для масиву покажчиків на рядки матриці;

for (int i = 0; i < n; i++)

a[i] = new int [m]; — кожному елементу масиву покажчиків на рядки присвоюється адреса початку ділянки пам’яті, виділеної для рядка матриці.

Виділення пам'яті для об'єктів

Використовуючи оператор **new**, можна динамічно виділяти пам'ять для об'єктів. У цьому випадку оператор поверне вказівник на створений об'єкт. Динамічно створений об'єкт нічим не відрізняється від інших. При його створенні також викликається конструктор (якщо він передбачений), а при звільненні пам'яті викликається відповідний деструктор.

Розглянемо невелику програму, у якій визначений клас **balance**, призначений для зберігання імені людини й суми, що лежить на його банківському рахунку. Усередині функції **main()** об'єкт класу **balance** створюється динамічно.

```
#include <iostream>
#include <new>
#include <cstring>
using namespace std;
class balance { double cur_bal; char name[80];
public:
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s); }
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80] ;
double n;
try {
p = new balance; } catch (bad_alloc xa){
cout « "Виняткова ситуація\n";
return 1; }
p->set(12387. 87, "Ральф Уилсон");
p->get_bal(n, s) ;
cout « s << ": сума = " << n,-cout « "\n";
delete p;
return 0; }
```

Оскільки змінна р є вказівником на об'єкт, для доступу до його членів використається оператор "->".

Як відомо, динамічні об'єкти можуть містити конструктори й деструктор. Крім того, конструктори можуть мати параметри. Розглянемо модифікацію попередньої програми.

```
#include <iostream>
#include <new>
#nclude <cstring>
using namespace std;
class balance { double cur_bal; char name[80]; public:
balance(double n, char *s) { cur_bal = n; strcpy(name, s); }
~balance() {
cout « "Знищення об'єкта ";
cout << name « "\n"; } void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80];
double n;
// У цій версії використається ініціалізація
try {
p = new balance (12387.87, "Ральф Уилсон"),-} catch (bad_alloc xa) {
cout << "Виняткова ситуація\n" ,-
return 1; }
p->get_bal(n, s) ;
cout « s « ": сума = " « n; cout « "\n";
delete p;
return 0; }
```

Зверніть увагу на те, що параметри конструктора об'єкта зазначені після імені типу, як при звичайній ініціалізації.

У динамічній пам'яті можна розмістити масиви об'єктів, однак варто мати на увазі одну пастку. Масиви, розміщені в динамічній пам'яті за допомогою оператора **new**, не можна ініціалізувати, тому варто переконатися, що клас містить конструктор, що не має параметрів. У противному випадку компілятор мови C++ не знайде підходящого конструктора й при спробі розмістити масив у динамічній пам'яті видасть повідомлення про помилку.

У наступній версії нашої програми в динамічній пам'яті розміщається масив **balance**, і викликається конструктор, що не має параметрів.

```
#include <iostream>
#include <new>
#nclude <cstring>
using namespace std;
class balance { double cur_bal; char name[80]; public:
balance(double n, char *s) { cur_bal = n; s trcpy(name, s); }
balance() {} // Конструктор без параметрів
~balance() {
cout « "Знищення об'єкта ";
cout « name « "\n"; }
void set(double n, char *s) {
cur_bal = n;
strcpy(name, s); }
void get_bal(double &n, char *s) {
n = cur_bal;
strcpy(s, name); } };
int main() {
balance *p;
char s[80];
double n;
int i ;
try {
p = new balance [3]; // Розміщення масиву } catch (bad_alloc xa) {
cout « "Виняткова ситуація\n";
return 1; }
// Використається оператор ".", а не ->
p[0].set(12387.87, "Ральф Уилсон");
p[l].set(144.00, "А. С. Коннерс"),
p[2].set(-11.23, "И. М. Овердроун");
for(i=0; i<3; i++) { p[i].get_bal(n, s);
cout << s « ": сума = " « n; cout « "\n"; }
delete [] p;
 return 0; }
```

У результаті роботи цієї програми на екран будуть виведені наступні рядки.

```
Ральф Уилсон: сума = 12387.9
А. С. Коннерс: сума = 144
И. М. Овердроун: сума = -11.23
Знищення об'єкта И. М. Овердроун
Знищення об'єкта А. С. Коннерс
Знищення об'єкта Ральф Уилсон
```

Для знищення масиву динамічних об'єктів варто застосовувати оператор **delete [ ]**, щоб можна було викликати деструктор для кожного об'єкта окремо.

Альтернатива nothrow

У випадку недостачі пам'яті оператор **new** може не генерувати виняткову ситуацію, а повертати нульовий вказівник. Цей вид оператора **new** виявляється досить корисним при роботі зі старими компіляторами. Особливо корисно, що виклики функції malloc () можна замінити оператором new. (Це часто доводиться робити при перекладі програм з мови С на мову С++.) Для цього застосовується наступна форма оператора new.

вказівник = new (nothrow) *тип;*

Ця форма оператора **new** нагадує його ранні версії. Оскільки у випадку недостачі пам'яті він не генерує виняткову ситуацію, а повертає нульовий вказівник, його можна впроваджувати в старі програми. Однак у нових програмах краще використати новий варіант оператора new, що генерує виняткову ситуацію **bad_alloc.** Щоб мати можливість використати опцію **nothrow,** варто включити в програму заголовок <new>.

Розглянемо програму, що демонструє альтернативу **new(nothrow).**

```
// Демонстрація альтернативи new(nothrow)
#include <iostream>
#include <new>
using namespace std;
```

```
int main() {
int   *p,    i;
p = new(nothrow) int[32]; // Застосування опції nothrow
if(!p) {
cout << "Виняткова ситуація.\n";
return 1; }
for(i=0; i<32; i++) p[i] = i;
for(i=0; i<32; i++) cout « p[i] « "
delete [] p; // Звільняємо пам'ять
return 0; }
```

Як показує ця програма, при використанні альтернативи **nothrow** варто перевіряти вказівник, що повертає оператором new.

Буферизований оператор new

У мові C++ існує особлива форма оператора **new**, яку можна застосовувати для вказівки альтернативного способу розподілу динамічної пам'яті. Ця форма називається *буферизованим оператором* **new** (replacement form of new). Найбільш корисний цей засіб виявляється при перевантаженні оператора **new** в особливих ситуаціях. Його загальний вид такий.

вказівник = new (*список_аргументів*) *тип;*

Тут *список аргументів* являє собою перерахування значень, розділених комами, які передаються перевантаженому оператору new.

Функції

Функція — це послідовність стейтментів для виконання певного завдання. Часто ваші програми переривають виконання одних функцій заради виконання інших. Ви це постійно робите в реальному житті, наприклад, ви читаєте книгу і згадали, що повинні були зробити телефонний дзвінок. Ви залишаєте *закладку* в своїй книзі, берете телефон і набираєте номер. Після того, як ви вже поговорили, ви повертаєтеся до тієї сторінки в книзі, на якій ви зупинилися.

Програми в C++ працюють схожим чином. Іноді, коли програма виконує код, вона може зіткнутися з викликом функції. **Виклик функції** — це **вираз**, який вказує процесору перервати виконання поточної функції і приступити до виконання іншої функції. Процесор "залишає закладку" в поточній точці виконання, а потім виконує функцію, що викликається. Коли виконання функції, що викликається, — завершено, то процесор повертається до "закладки" і відновлює виконання перерваної функції.

Функція, в якій знаходиться виклик, називається **викликаючою функцією** (англ. *"caller"*). Наприклад:

```
#include <iostream> // для std::cout i std::endl

// Оголошення функції doPrint(), яку ми будемо викликати
void doPrint() {
	std::cout << "In doPrint()" << std::endl;
}

// Оголошення функції main()
int main()
{
	std::cout << "Starting main()" << std::endl;
	doPrint(); // перериваємо виконання main() викликом функції doPrint(). Функція main() в
даному випадку є викликаючою функцією
	std::cout << "Ending main()" << std::endl;
	return 0;
  }
```

Результат виконання програми:

```
Starting main()
In doPrint()
Ending main()
```

Ця програма починає своє виконання з першого рядка функції main(), в якому виводиться на екран Starting main(). Другий рядок функції main() викликає функцію doPrint(). На цьому етапі виконання стейтментів у функції main() призупиняється і процесор переходить до виконання стейтментів всередині функції doPrint(). Перший (і єдиний) рядок в doPrint() виводить текст In doPrint().

Коли процесор завершує виконання doPrint(), він повертається назад в функцію main() до тієї точки, на якій зупинився. Отже, наступним стейтментом є вивід рядка Ending main() на екран.

Зверніть увагу, для виклику функції потрібно вказати її ім'я і список параметрів в круглих дужках (). У вищенаведеному прикладі параметри не використовуються, тому круглі дужки порожні. Ми поговоримо детально про параметри функцій на відповідному уроці.

Правило: Не забувайте вказувати круглі дужки () при виклику функцій.

Значення, що повертаються

Коли функція main() завершує своє виконання, вона повертає цілочисельне значення назад в операційну систему, використовуючи **оператор return**.

Функції, які ми пишемо, також можуть повертати значення. Для цього потрібно вказати **тип повернення значення**. Він вказується при оголошенні функції, перед її ім'ям. Зверніть увагу, що тип повернення не вказує, яке саме значення повертатиметься. Він вказує тільки тип цього значення.

Після цього всередині функції, що викликається, ми використовуємо оператор return, щоб вказати фактичне **значення, що повертається**.

Розглянемо просту функцію, яка повертає цілочисельне значення:

```
#include <iostream>

// int означає, що функція повертає цілочисельне значення у викликаючу функцію
int return7()
{
	// Ця функція повертає цілочисельне значення, тому ми повинні використовувати оператор return
	return 7; // повертаємо число 7 у викликаючу функцію
}

int main()
{
	std::cout << return7() << std::endl; // на екран виведеться 7
	std::cout << return7() + 3 << std::endl; // на екран виведеться 10

	return7(); // значення 7, що повертається, - ігнорується, тому що main() з ним нічого не робить

	return 0;
}
```

Результат виконання програми:

```
7
10
```

Тепер давайте розберемося детально:

- Перший виклик функції return7() повертає 7 у викличаючу функцію, де воно потім передається в **std::cout** для виводу на екран.
- Другий виклик функції return7() знову повертає 7 у викликаючу функцію. Вираз 7 + 3 генерує результат 10, який виводиться на екран.
- Третій виклик функції return7() знову повертає 7 у викликаючу функцію. Проте main() нічого з ним не робить, тому нічого і не відбувається (значення, що повертається, просто ігнорується).

Примітка*:* Значення, що повертаються, не виводяться на екран, якщо їх не передавати об'єкту std::cout. В останньому виклику функції return7() значення не надсилається в std::cout, тому нічого і не відбувається.

Тип повернення void

Функції можуть і не повертати значення. Щоб повідомити компілятору, що функція не повертає значення, потрібно використати **тип повернення void**. Погляньмо ще раз на функцію doPrint() з вищенаведеного прикладу:

```
void doPrint() // void - це тип повернення
{
	std::cout << "In doPrint()" << std::endl;
	// Ця функція не повертає значення, тому і оператор return тут не потрібен
```

```
}
```

Ця функція має тип повернення void, який означає, що функція не повертає значення. Оскільки значення не повертається, то і оператор return тут не потрібен.

Ось ще один приклад використання функції типу void:

```
#include <iostream>

// void означає, що функція не повертає значення
void returnNothing()
{
	std::cout << "Hi!" << std::endl;
	// Ця функція не повертає значення, тому і оператор return тут не потрібен
}

int main()
{
	returnNothing(); // функція returnNothing() викликається, але в main() нічого не повертається

	std::cout << returnNothing(); // помилка: цей рядок не скомпілюється (вам потрібно буде його закоментувати)
	return 0;
}
```

У першому виклику функції returnNothing () виводиться Hi!, але нічого не повертається назад у викликаючу функцію. Точка виконання повертається назад в main(), де програма продовжує своє виконання.

Другий виклик функції returnNothing () навіть не скомпілюється. Функція returnNothing() має тип повернення void, який означає, що ця функція не повертає значення. Однак main() намагається відправити це значення (яке не повертається) в std::cout для виведення на екран. std::cout не може опрацювати цей випадок, оскільки не було надано значення для виводу. Отже, компілятор видасть помилку. Вам потрібно буде **закоментувати** цей рядок, щоб компіляція пройшла успішно.

Повернення значень функцією main()

Тепер у вас є розуміння того, як працює функція main(). Коли програма виконується, операційна система викликає функцію main() і починається її виконання. Стейтменти в main() виконуються послідовно. В кінці функція main() повертає цілочисельне значення (зазвичай 0) назад в операційну систему. Тому main() оголошується як int main().

Чому потрібно повертати значення назад в операційну систему? Справа в тому, що значення, що повертається функцією main(), є **кодом стану**, який повідомляє операційній системі про те, чи успішно було виконання програми. Зазвичай, значення 0 (нуль) означає що все пройшло успішно, тоді як будь-яке інше значення означає невдачу/помилку.

Зверніть увагу, за стандартами мови C++ функція main() повинна повертати цілочисельне значення. Однак, якщо ви не вкажете return в кінці функції main(), то компілятор поверне 0 автоматично, якщо ніяких помилок не буде. Проте все ж рекомендується вказувати оператор return в кінці функції main() і використовувати тип повернення int для функції main().

Детальніше про значення, що повертаються

По-перше, якщо тип повернення функції не є void, то вона повинна повертати значення зазначеного типу (використовуючи оператор return). Єдиний виняток — це функція main(), яка повертає 0, якщо не надано інше значення.

По-друге, коли процесор зустрічає в функції оператор return, він негайно виконує повернення значення назад у викликаючу функцію і точка виконання також переходить у викликаючу функцію. Будь-який код, який знаходиться за оператором return у функції — ігнорується.

Функція може повертати тільки одне значення через оператор return у викликаючу функцію. Це може бути або число (наприклад, 7), або значення змінної, або вираз (який генерує результат), або певне значення з набору можливих значень.

Є способи обійти правило повернення одного значення, але про це трішки пізніше.

Нарешті, автор функції вирішує, що означатиме значення, яке повертає функція. Деякі функції використовують повернені значення в якості кодів стану для надання інформації щодо результату

виконання функції (успішне виконання чи ні). Інші функції повертають певне значення з набору можливих значень, ще інші функції взагалі нічого не повертають.

Повторне використання функцій

Одну і ту ж функцію можна викликати декілька разів, навіть в різних програмах, що дуже корисно:

```
#include <iostream>

// Функція getValueFromUser() отримує значення від користувача, а потім повертає його назад у викликаючу функцію
int getValueFromUser()
{
	std::cout << "Enter an integer: ";
	int x;
	std::cin >> x;
	return x;
}

int main()
{
	int a = getValueFromUser(); // перший виклик функції getValueFromUser()
	int b = getValueFromUser(); // другий виклик функції getValueFromUser()

	std::cout << a << " + " << b << " = " << a + b << std::endl;

	return 0;
  }
```

Результат виконання програми:

Enter an integer: 4
Enter an integer: 9
4 + 9 = 13

Тут виконання функції main() переривається 2 рази. Зверніть увагу, в обох випадках отримане значення від користувача зберігається в змінній x, а потім передається назад в функцію main() за допомогою оператора return, де присвоюється змінній a або b!

Також main() не є єдиною функцією, яка може викликати інші функції. Будь-яка функція може викликати будь-яку іншу функцію!

```
#include <iostream>

void printO()
{
	std::cout << "O" << std::endl;
}

void printK()
{
	std::cout << "K" << std::endl;
}

// Функція printOK() викликає як printO(), так і printK()
void printOK()
{
	printO();
	printK();
}

// Оголошення функції main()
int main()
{
	std::cout << "Starting main()" << std::endl;
	printOK();
	std::cout << "Ending main()" << std::endl;
	return 0;
  }
```

Результат виконання програми:

```
Starting main()
OK
Ending main()
```

Вкладені функції

В мові C++ одні функції не можуть бути оголошені всередині інших функцій (тобто бути вкладеними). Наступний код викличе помилку компіляції:

```
#include <iostream>

int main()
{
	int boo() // ця функція знаходиться всередині функції main(), що є заборонено
	{
		std::cout << "boo!";
		return 0;
	}

	boo();
	return 0;
  }
```

Правильно ось так:

```
#include <iostream>

int boo() // тепер вже не в функції main()
{
	std::cout << "boo!";
	return 0;
}

int main()
{
	boo();
	return 0;
  }
```

Параметри і аргументи функції

У багатьох випадках нам потрібно буде передавати дані в функцію, що викликається, щоб вона могла з ними якось взаємодіяти. Наприклад, якщо ми хочемо написати функцію множення двох чисел, то нам потрібно якимось чином повідомити функцію про те, що це будуть за числа. В іншому випадку, як вона дізнається, що на що множити? Тут нам на допомогу приходять параметри і аргументи.

Параметр функції — це змінна, яка використовується в функції і значення якої надає викликаюча функція. Параметри вказуються при оголошенні функції в круглих дужках. Якщо їх багато, то вони перераховуються через кому, наприклад:

```
// Ця функція не має параметрів
void doPrint()
{
	std::cout << "In doPrint()" << std::endl;
}

// Ця функція має один параметр цілочисельного типу - a
void printValue(int a)
{
	std::cout << a << std::endl;
}

// Ця функція має два параметри цілочисельного типу - a і b
int add(int a, int b)
{
	return a + b;
```

```
}
```

Параметри кожної функції дійсні тільки всередині цієї функції. Тому, якщо і printValue(), і add() мають параметр a, то це не означає, що відбудеться конфлікт імен. Ці параметри вважаються окремими і ніяк не взаємодіють один з одним.

Аргумент функції — це значення, яке передається з викликаючої функції в функцію, що викликається, і яке вказується в круглих дужках при виклику:

```
printValue(7); // 7 – це аргумент функції printValue()
  add(4, 5); // 4 і 5 – це аргументи функції add()
```

Зверніть увагу, аргументи також перераховуються через кому. Кількість аргументів має збігатися з кількістю параметрів, інакше компілятор видасть повідомлення про помилку.

Як працюють параметри і аргументи функцій?

При виклику функції, всі її параметри створюються як локальні змінні, а значення кожного з аргументів копіюється у відповідний параметр (локальну змінну). Цей процес називається *"передачею по значенню"*, наприклад:

```
#include <iostream>

// Ця функція має два параметри типу int: a і b.
// Значення змінних a і b визначає викликаюча функція
void printValues(int a, int b)
{
	std::cout << a << std::endl;
	std::cout << b << std::endl;
}

int main()
{
	printValues(8, 9); // тут два аргументи: 8 і 9

	return 0;
  }
```

При виклику функції printValues() аргументи 8 і 9 копіюються в параметри a і b. Параметру a присвоюється значення 8, а параметру b — значення 9.

Результат:

```
8
9
```

Як працюють параметри і значення, що повертаються?

Використовуючи параметри і значення, що повертаються, ми можемо створювати функції, які можуть приймати і опрацьовувати дані, а потім повертати результат назад у викликаючу функцію.

Наприклад, ось проста функція, яка приймає два цілих числа і повертає їх суму:

```
#include <iostream>

// Функція add() приймає два цілих числа в якості параметрів і повертає їх суму.
// Значення a і b визначає викликаюча функція
int add(int a, int b)
{
	return a + b;
}

// Функція main() не має параметрів
int main()
{
	std::cout << add(7, 8) << std::endl; // аргументи 7 і 8 передаються в функцію add()
	return 0;
  }
```

При виклику функції add(), параметру a присвоюється значення 7, а параметру b — значення 8. Після цього функція add() обчислює їх суму і повертає результат назад у функцію main(), де він вже виводиться на екран.

Результат виконання програми:

15

# include <iostream> 	
using namespace std;
 	
void func(int *a, int *b)
{
int c=*a;
*a = *b; *b =c ;
}
void main()
{
int a = 10; int b = 20;
func(&a,&b);
cout <<"a="<<a<<endl;
cout <<"b="<<b<<endl;
}
	# include <iostream>
	using namespace std;
	
	void func(int &a, int &b)
	{
	int c=a;
	a = b; b =c ;
	}
	void main()
	{
	int a = 10; int b = 20;
	func(a,b);
	cout <<"a="<<a<<endl;
	cout <<"b="<<b<<endl;
	}	

Робота з функціями. Продовження

Існує 3 основних способи передачі аргументів у функцію:
передача по значенню;
передача по посиланню;
передача по адресі.

Передача по значенню

За замовчуванням, аргументи в мові C++ передаються по значенню. Коли аргумент **передається по значенню**, то його значення копіюється в параметр функції. Наприклад:

```
#include <iostream>

void boo(int y)
{
	std::cout << "y = " << y << std::endl;
}

int main()
{
	boo(7); // 1-й виклик

	int x = 8;
	boo(x); // 2-й виклик
	boo(x + 2); // 3-й виклик

	return 0;
}
```

У першому виклику функції boo() аргументом є **літерал** 7. При виклику boo() створюється змінна y, в яку копіюється значення 7. Потім, коли boo() завершує своє виконання, змінна y знищується.
У другому виклику функції boo() аргументом вже є змінна x = 8. Коли boo() викликається вдруге, змінна y створюється знову і значення 8 копіюється в y. Потім, коли boo() завершує своє виконання, змінна y знову знищується.
У третьому виклику функції boo() аргументом є вираз x + 2, який обчислюється в значення 10. Потім це значення передається в змінну y. При завершенні виконання функції boo() змінна y знову знищується.
Таким чином, результат виконання програми:
y = 7
y = 8
y = 10

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

```
#include <iostream>

void boo(int y)
{
	std::cout << "y = " << y << '\n';

	y = 8;

	std::cout << "y = " << y << '\n';
} // змінна y знищується тут

int main()
{
	int x = 7;
	std::cout << "x = " << x << '\n';
```

```
	boo(x);

	std::cout << "x = " << x << '\n';
	return 0;
}
```

Результат:
x = 7
y = 7
y = 8
x = 7

На початку функції main() змінна x дорівнює 7. При виклику boo() значення x (7) передається в параметр y функції boo(). Усередині boo() змінній y спочатку присвоюється значення 8, а потім y знищується. Значення x не змінюється, навіть якщо змінити y.
Параметри функції, передані по значенню, також можуть бути **const**. Тоді вже буде 100% гарантія того, що функція не змінить значення параметру.
Плюси і мінуси передачі по значенню

Плюси передачі по значенню:
Аргументи, передані по значенню, можуть бути змінними (наприклад, x), літералами (наприклад, 8), виразами (наприклад, x + 2), **структурами**, класами або **перерахуваннями** (тобто майже будь-чим).
Аргументи ніколи не змінюються функцією, в яку передаються, що запобігає виникненню **побічних ефектів**.

Мінуси передачі по значенню:
Копіювання структур і класів може призвести до значного зниження продуктивності (особливо, коли функція викликається багато разів).

Коли використовувати передачу по значенню:
При передачі фундаментальних типів даних і енумераторів, коли припускається, що функція не повинна змінювати аргумент.

Коли не використовувати передачу по значенню:
При передачі **масивів**, структур і класів.
У більшості випадків, передача по значенню — це найкращий спосіб передачі аргументів фундаментальних типів даних, коли функція не повинна змінювати вихідні значення. Передача по значенню є гнучкою і безпечною, а у випадку фундаментальних типів даних ще й ефективною.

Передача по посиланню

При **передачі змінної по посиланню** потрібно просто оголосити параметри функції як **посилання**, а не як звичайні змінні:

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

При виклику функції змінна x стане посиланням на аргумент. Оскільки посилання на змінну обробляється точно так же, як і сама змінна, то будь-які зміни, внесені в посилання, призведуть до змін вихідного значення аргументу! У наступному прикладі це добре проілюстровано:

```
#include <iostream>

void boo(int& value)
```

```
{
	value = 7;
}

int main()
{
	int value = 6;

	std::cout << "value = " << value << '\n';
	boo(value);
	std::cout << "value = " << value << '\n';
	return 0;
}
```

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

value = 6
value = 7

Як ви можете бачити, функція змінила значення аргументу з 6 на 7!
Ось ще один приклад:

```
#include <iostream>

void addOne(int& x) // x - це змінна-посилання
{
	x = x + 1;
} // x знищується тут

int main()
{
	int a = 7;
	std::cout << "a = " << a << '\n';
	addOne(a);
	std::cout << "a = " << a << '\n';
	return 0;
}
```

Результат виконання програми:
a = 7
a = 8
Зверніть увагу, значення аргументу a було змінено функцією.

Повернення відразу декількох значень

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

```
#include <iostream>
#include <math.h> // для sin() і cos()

void getSinCos(double degrees, double& sinOut, double& cosOut)
{
	// sin() і cos() приймають радіани, а не градуси, тому потрібна конвертація
	const double pi = 3.14159265358979323846; // значення Пі
	double radians = degrees * pi / 180.0;
	sinOut = sin(radians);
	cosOut = cos(radians);
}

int main()
{
	double sin(0.0);
```

```
	double cos(0.0);

	// Функція getSinCos() повертає sin і cos в змінні sin і cos
	getSinCos(30.0, sin, cos);

	std::cout << "The sin is " << sin << '\n';
	std::cout << "The cos is " << cos << '\n';
	return 0;
}
```

Ця функція приймає один параметр (передача по значенню) в якості вхідних даних і «повертає» два параметри (передача по посиланню) в якості вихідних даних. Параметри, які використовуються тільки для повернення значень назад в caller, називаються **параметрами виводу**. Вони дають зрозуміти caller-у, що значення вихідних змінних, переданих у функцію, не настільки значні, так як ми очікуємо, що ці змінні будуть перезаписані.

Давайте розглянемо це детально. По-перше, в функції main() ми створюємо локальні змінні sin і cos. Вони передаються в функцію getSinCos() по посиланню (а не по значенню). Це означає, що функція getSinCos() має прямий доступ до початкових значень змінних sin і cos, а не до їх копій. Функція getSinCos(), відповідно, присвоює нові значення змінним sin і cos (через посилання sinOut і cosOut), перезаписуючи їх старі значення. Потім main() виводить ці оновлені значення.

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

Хоча цей спосіб хороший, але він також має свої нюанси. По-перше, синтаксис трохи незвичний, оскільки параметри вводу і виводу вказуються разом з викликом функції. По-друге, в caller-і не очевидно, що sin і cos є параметрами виводу, і вони будуть змінені функцією. Це, ймовірно, найнебезпечніша частина даного способу передачі (так як може призвести до помилок). Деякі програмісти вважають це досить великою проблемою, і не радять передавати аргументи по посиланню, віддавши перевагу передачі по адресі, не змішуючи при цьому параметри вводу і виводу.

Особисто я не рекомендую змішувати параметри вводу і виводу саме з цієї причини, але якщо ви це робите, то обов'язково додавайте **коментарі до коду**, описуючи, що і як ви робите.

Неконстантні посилання можуть посилатися тільки на неконстантні **l-values** (наприклад, на неконстантні змінні), тому параметр-посилання не може прийняти аргумент, який є константним l-value або r-value (наприклад, літералом або результатом виразу).

Передача по константному посиланню

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

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

Ви вже знаєте, що **константне посилання** — це посилання на змінну, значення якої змінити через це ж посилання ніяк не вийде. Отже, якщо ми використовуємо константне посилання в якості параметру, то отримуємо 100% гарантію того, що функція не змінить аргумент!
Запустивши наступний фрагмент коду, ми отримаємо помилку компіляції:

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

Використання const корисне з наступних причин:

- Ми отримуємо гарантію від компілятора, що значення, які не повинні бути змінені — не зміняться (компілятор видасть помилку, якщо ми спробуємо зробити щось подібне тому, що було у вищенаведеному прикладі).
- Програміст, бачачи const, розуміє, що функція не змінить значення аргументу. Це може допомогти при **відлагодженні програми**.
- Ми не можемо передати константний аргумент в неконстантне посилання-параметр. Використання константного параметру гарантує, що ми зможемо передавати як неконстантні, так і константні аргументи в функцію.
- Константні посилання можуть приймати будь-які типи аргументів, включаючи l-values, константні l-values і r-values.

Правило: При передачі аргументів по посиланню завжди використовуйте константні посилання, якщо вам не потрібно, щоб функція змінювала значення аргументів.

Плюси і мінуси передачі по посиланню

Плюси передачі по посиланню:

- Посилання дозволяють функції змінювати значення аргументу, що іноді корисно. В іншому випадку, для гарантії того, що функція не змінить значення аргументу, потрібно використовувати константні посилання.
- Оскільки при передачі по посиланню копіювання аргументів не відбувається, то цей спосіб набагато ефективніший і швидший за передачу по значенню, особливо при роботі з великими структурами або класами.
- Посилання можуть використовуватися для повернення відразу декількох значень з функції (через параметри виводу).

Мінуси передачі по посиланню:

- Важко визначити, чи є параметр, переданий по неконстантному посиланню, параметром вводу, параметром виводу чи тим і іншим одночасно. Розумне використання const і суфікса Out для зовнішніх змінних вирішує цю проблему.
- По виклику функції неможливо визначити, чи буде аргумент змінений функцією чи ні. Аргумент, переданий по значенню або по посиланню, виглядає однаково. Ми можемо визначити спосіб передачі аргументу тільки переглянувши оголошення функції. Це може призвести до ситуації, коли програміст не відразу зрозуміє, що функція змінює значення аргументу.

Коли використовувати передачу по посиланню:

- при передачі структур або класів (використовуйте const, якщо потрібно тільки для читання);
- коли потрібно, щоб функція змінювала значення аргументу.

Коли не використовувати передачу по посиланню:

- при передачі фундаментальних типів даних (використовуйте передачу по значенню);
- при передачі звичайних **масивів** (використовуйте передачу по адресу).

Передача по адресі

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

```
#include <iostream>

void boo(int* ptr)
{
	*ptr = 7;
}

int main()
```

```
{
	int value = 4;

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

Результат виконання програми:

value = 4
value = 7

Як ви можете бачити, функція boo() змінила значення аргументу (змінну value) через параметр-вказівник ptr. Передачу по адресі зазвичай використовують з вказівниками на звичайні **масиви**. Наприклад, наступна функція виведе всі значення масиву:

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

Ось приклад програми, яка викликає цю функцію:

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

Результат:
9 8 6 4 3 2 1

Пам’ятайте, що **фіксовані масиви конвертуються у вказівники** при передачі у функцію, тому їх довжину потрібно передавати в якості окремого параметру. Перед розіменуванням параметрів, переданих по адресі, не зайвим буде перевірити — чи не є вони **нульовими вказівниками**. Розіменування нульового вказівника призведе до збою в програмі. Ось функція printArray() з перевіркою (виявленням) нульових вказівників:

```
#include <iostream>

void printArray(int* array, int length)
{
	// Якщо користувач передав нульовий вказівник в якості array
	if (!array)
		return;

	for (int index = 0; index < length; ++index)
		std::cout << array[index] << ' ';
}

int main()
{
	int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
	printArray(array, 7);
}
```

Передача по константній адресі

Оскільки printArray() все одно не змінює значення отриманих аргументів, то гарною ідеєю буде зробити параметр array константою:

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

	for (int index = 0; index < length; ++index)
		std::cout << array[index] << ' ';
}

int main()
{
	int array[7] = { 9, 8, 6, 4, 3, 2, 1 };
	printArray(array, 7);
  }
```

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

```
#include <iostream>

void setToNull(int* tempPtr)
{
    // Ми присвоюємо tempPtr інше значення (ми не змінюємо значення, на яке вказує tempPtr)
    tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}

int main()
{
    // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
    int six = 6;
    int* ptr = &six;

    // Тут виведеться 6
    std::cout << *ptr << "\n";

    // tempPtr отримує копію ptr
    setToNull(ptr);

    // ptr до сих пір вказує на змінну six!

    // Тут виведеться 6
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
  }
```

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

```
6
6
```

Зверніть увагу, хоча сама адреса передається по значенню, ви все одно можете розіменувати її для зміни значення вихідного аргументу. Заплутано? Давайте розберемося детально:

При передачі аргументу по адресі в змінну-параметр функції копіюється адреса з аргументу. У цей момент параметр функції і аргумент вказують на одне і те ж значення.

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

Якщо параметру функції присвоїти іншу адресу, то це ніяк не вплине на аргумент, оскільки параметр функції є копією, а зміна копії не призводить до зміни оригіналу. Після зміни адреси параметра функції, параметр функції і аргумент вказуватимуть на різні значення, тому розіменування параметру і подальша його зміна ніяк не вплинуть на значення, на яке вказує аргумент.

У наступній програмі це все добре проілюстровано:

```
#include <iostream>

void setToSeven(int* tempPtr)
{
    *tempPtr = 7; // ми змінюємо значення, на яке вказує tempPtr (і ptr також)
}

int main()
{
    // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
    int six = 6;
    int* ptr = &six;

    // Тут виведеться 6
    std::cout << *ptr << "\n";

    // tempPtr отримує копію ptr
    setToSeven(ptr);

    // tempPtr змінив значення, на яке вказував, на 7

    // Тут виведеться 7
    if (ptr)
        std::cout << *ptr << "\n";
    else
        std::cout << " ptr is null";

    return 0;
  }
```

Результат виконання програми:

```
6
7
```

Передача адрес по посиланню

Виникає питання: «А що, якщо ми хочемо змінити адресу, на яку вказує аргумент, всередині функції?». Виявляється, це можна зробити дуже легко. Ви можете просто передати адресу по посиланню. Синтаксис посилання на вказівник може здатися трохи дивним, але все ж:

```
#include <iostream>

// tempPtr тепер є посиланням на вказівник, тому будь-які зміни tempPtr призведуть і до зміни
вихідного аргументу!
void setToNull(int*& tempPtr)
{
      tempPtr = nullptr; // використовуйте 0, якщо не підтримується C++11
}

int main()
{
      // Спочатку ми присвоюємо ptr адресі six, тобто *ptr = 6
      int six = 6;
      int* ptr = &six;
```

```
    // Тут виведеться 6
    std::cout << *ptr;

    // tempPtr є посиланням на ptr
    setToNull(ptr);

    // ptr було присвоєно значення nullptr!

    if (ptr)
        std::cout << *ptr;
    else
        std::cout << " ptr is null";

    return 0;
}
```

Результат виконання програми:

6 ptr is null

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

Існує тільки передача по значенню

Тепер, коли ви розумієте основні відмінності між передачею по посиланню, по адресі і по значенню, давайте трохи поговоримо про те, що знаходиться “під капотом”.

На уроці про **посилання** ми згадували, що посилання насправді реалізуються за допомогою вказівників. Це означає, що передача по посиланню є просто передачею по адресі. І трохи вище ми говорили, що передача по адресі насправді є передачею адреси по значенню! З цього випливає, що мова C++ дійсно передає все по значенню!

Плюси і мінуси передачі по адресі

Плюси передачі по адресі:

- Передача по адресі дозволяє функції змінити значення аргументу, що іноді корисно. В іншому випадку, використовуємо const для гарантії того, що функція не змінить аргумент.
- Оскільки копіювання аргументів не відбувається, то швидкість передачі по адресі досить висока, навіть якщо передавати великі **структури** або класи.
- Ми можемо повернути відразу декілька значень з функції, використовуючи **параметри виводу**.

Мінуси передачі по адресі:

- Всі вказівники потрібно перевіряти, чи не є вони нульовими. Спроба розіменувати нульовий вказівник призведе до збою в програмі.
- Оскільки розіменування вказівника виконується повільніше, ніж доступ до значення напряму, то доступ до аргументів, переданих по адресі, виконується також повільніше, ніж доступ до аргументів, переданих по значенню.

Коли використовувати передачу по адресі:

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

Коли не використовувати передачу по адресі:

- при передачі структур або класів (використовуйте передачу по посиланню);
- при передачі фундаментальних типів даних (використовуйте передачу по значенню).

Як ви можете бачити самі, передача по адресі і по посиланню мають майже однакові переваги і недоліки. Оскільки передача по посиланню зазвичай безпечніша, ніж передача по адресі, то в більшості випадків краще використовувати передачу по посиланню.

Правило: Використовуйте передачу по посиланню, замість передачі по адресі, коли це можливо.

Повернення по значенню

Повернення по значенню — це найпростіший і найбезпечніший тип повернення. При поверненні по значенню, копія значення, що повертається, передається назад в caller. Як і у випадку з передачею по значенню, ви можете повертати **літерали** (наприклад, 7), змінні (наприклад, x) або вирази (наприклад, x + 2), що робить цей спосіб дуже гнучким.

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

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

Повернення по значенню ідеально підходить для повернення змінних, які були оголошені всередині функції, або для повернення аргументів функції, які були передані по значенню. Однак, подібно до передачі по значенню, повернення по значенню повільне при роботі зі **структурами** і класами.

Коли використовувати повернення по значенню:

- при поверненні змінних, які були оголошені всередині функції;
- при поверненні аргументів функції, які були передані в функцію по значенню.

Коли не використовувати повернення по значенню:

- при поверненні **стандартних масивів** або **вказівників** (використовуйте повернення по адресі);
- при поверненні великих структур або класів (використовуйте повернення по посиланню).

Повернення по адресі

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

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

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

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

Повернення по адресі часто використовується для повернення динамічно виділеної пам'яті назад в caller:

```
int* allocateArray(int size)
{
	return new int[size];
}

int main()
{
	int* array = allocateArray(20);

	// Робимо що-небудь з array

	delete[] array;
```

```
    return 0;
}
```

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

Коли використовувати повернення по адресі:

- при поверненні динамічно виділеної пам'яті;
- при поверненні аргументів функції, які були передані по адресі.

Коли не використовувати повернення по адресі:

- при поверненні змінних, які були оголошені всередині функції (використовуйте повернення по значенню);
- при поверненні великої структури або класу, які були передані по посиланню (використовуйте повернення по посиланню).

Повернення по посиланню

Подібно передачі по посиланню, значення, які повертаються по посиланню, повинні бути змінними (ви не зможете повернути посилання на літерал або вираз). При **поверненні по посиланню** в caller повертається посилання на змінну. Потім caller може її використовувати для продовження модифікації змінної, що може бути іноді корисно. Цей спосіб також дуже швидкий і при поверненні великих структур або класів.

Однак, як і при поверненні по адресі, ви не повинні повертати локальні змінні по посиланню. Розглянемо наступний фрагмент коду:

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

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

Повернення по посиланню зазвичай використовується для повернення аргументів, переданих у функцію по посиланню. У наступному прикладі ми повертаємо (по посиланню) елемент масиву, який був переданий у функцію по посиланню:

```
#include <iostream>
#include <array>

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

int main()
{
    std::array<int, 20> array;

    // Присвоюємо елементу масиву під індексом 15 значення 7
    getElement(array, 15) = 7;

    std::cout << array[15] << '\n';

    return 0;
}
```

Результат виконання програми:

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

Хоча цей приклад непрактичний, так як ми можемо напряму звернутися до 15 елементу масиву, але як тільки ми будемо розглядати класи, то ви виявите набагато більше застосувань для повернення значень по посиланню.

Коли використовувати повернення по посиланню:

- при поверненні посилання-параметру;
- при поверненні елементу масиву, який був переданий у функцію;
- при поверненні великої структури або класу, який не знищується в кінці функції (наприклад, той, який був переданий у функцію).

Коли не використовувати повернення по посиланню:

- при поверненні змінних, які були оголошені всередині функції (використовуйте повернення по значенню);
- при поверненні стандартного масиву або значення вказівника (використовуйте повернення по адресі).

Змішування значень, що повертаються, з посиланнями

Хоча функція може повертати як значення, так і посилання, caller може неправильно це інтерпретувати. Подивимося, що станеться при змішуванні значень, що повертаються, з посиланнями на значення:

```
int returnByValue()
{
	return 7;
}

int& returnByReference()
{
	static int y = 7; // static гарантує те, що змінна y не знищиться, коли вийде з локальної області видимості
	return y;
}

int main()
{
	int value = returnByReference(); // випадок A: все добре, обробляється як повернення по значенню
	int& ref = returnByValue(); // випадок B: помилка компілятора, так як 7 - це r-value, а r-value не може бути прив'язане до неконстантного посилання
	const int& cref = returnByValue(); // випадок C: все добре, час життя значення, що повертається, продовжується відповідно до часу життя cref
}
```

У випадку A ми присвоюємо посилання на значення, що повертається, змінній, яка сама не є посиланням. Оскільки value не є посиланням, то значення, що повертається, просто копіюється в value так, наче returnByReference() був поверненням по значенню.

У випадку B ми намагаємося ініціалізувати посилання ref копією значення, що повертається з функції returnByValue(). Однак, оскільки значення, що повертається, не має адреси (це **r-value**), ми отримаємо помилку компіляції.

У випадку C ми намагаємося ініціалізувати **константне посилання** cref копією значення, що повертається з функції returnByValue(). Оскільки константні посилання можуть бути ініціалізовані за допомогою r-values, то тут не повинно виникати ніяких проблем. Зазвичай r-values знищуються в кінці виразу, в якому вони створені, однак, при прив’язці до константного посилання, час життя r-value (в даному випадку, значення, що повертається з функції) продовжується відповідно до часу життя посилання (в даному випадку, cref) .

Висновки

У більшості випадків ідеальним варіантом для використання є повернення по значенню. Це також найгнучкіший і найбезпечніший спосіб повернення даних у викликаючий об’єкт. Однак повернення по посиланню або по адресі також може бути корисним при роботі з динамічно виділеною пам’яттю. При використанні повернення по посиланню або по адресі переконайтеся, що ви не повертаєте посилання або адресу локальної змінної, яка вийде з області видимості, коли функція завершить своє виконання!

Перевантаження функцій

Перевантаження функцій — це можливість визначати декілька функцій з одним і тим же ім’ям, але з різними параметрами. Наприклад:

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

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

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

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

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

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

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

Тепер у нас є дві версії функції subtract():

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

Ви можете подумати, що відбудеться **конфлікт імен**, але це не так. Компілятор може визначити сам, яку версію subtract() слід викликати на основі аргументів, які використовуються у виклику функції. Якщо параметрами будуть змінні типу int, то мова C++ розуміє, що ми хочемо викликати subtract(int, int). Якщо ж ми надамо два значення типу з плаваючою крапкою, то мова C++ зрозуміє, що ми хочемо викликати subtract(double, double). Фактично, ми можемо визначити стільки перевантажених функцій subtract(), скільки хочемо, до тих пір, поки кожна з них матиме свої (унікальні) параметри.

Відповідно, функцію subtract() можна визначити і з більшою кількістю параметрів:

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

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

Типи повернення в перевантаженні функцій

Зверніть увагу, **тип повернення функції** НЕ враховується при перевантаженні функції. Припустимо, що ви хочете написати функцію, яка повертає **рандомне число**, але вам потрібна одна версія, яка повертає значення типу int, і друга — яка повертає значення типу double. У вас може виникнути спокуса зробити наступне:

```
int getRandomValue();
  double getRandomValue();
```

Компілятор видасть помилку. Ці дві функції мають однакові параметри (точніше, їх відсутність), і другий виклик функції getRandomValue() розглядатиметься як помилкове перевизначення першого виклику. Імена функцій потрібно буде змінити.

Псевдоніми типів в перевантаженні функцій

Оскільки оголошення **typedef (псевдоніма типу)** не створює новий тип даних, то наступні два оголошення функції print() вважаються ідентичними:

```
typedef char* string;
void print(string value);
  void print(char* value);
```

Виклики функцій

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

Збіг знайдено. Виклик дозволений для відповідної перевантаженої функції.

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

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

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

Крок №1: **C++ намагається знайти точний збіг.** Це той випадок, коли фактичний аргумент точно відповідає типу параметра однієї з перевантажених функцій. Наприклад:

```
void print(char* value);
void print(int value);

  print(0); // точний збіг з print(int)
```

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

Крок №2: **Якщо точного збігу не знайдено, то C++ намагається знайти збіг шляхом подальшого неявного перетворення типів.** На уроці про **неявні конвертації** ми говорили про те, як певні типи даних можуть автоматично конвертуватися в інші типи даних. Якщо коротко, то:

- char, unsigned char і short конвертуються в int;
- unsigned short може конвертуватися в int або unsigned int (в залежності від розміру int);
- float конвертується в double;
- **enum** конвертується в int.

Наприклад:

```
void print(char* value);
void print(int value);

  print('b'); // збіг з print(int) після неявної конвертації
```

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

Крок №3: **Якщо неявне перетворення неможливе, то C++ намагається знайти відповідність за допомогою стандартного перетворення.** У стандартному перетворенні:

- будь-який числовий тип відповідатиме будь-якому іншому числового типу, включаючи unsigned (наприклад, int дорівнює float);
- enum відповідає формальному типу числового типу даних (наприклад, enum дорівнює float);
- нуль відповідає типу вказівника і числовому типу (наприклад, 0 як char * або 0 як float);
- вказівник відповідає **вказівнику типу void**.

Наприклад:

```
struct Employee; // визначення пропустимо
void print(float value);
void print(Employee value);

  print('b'); // 'b' конвертується у відповідність версії print(float)
```

В цьому випадку, оскільки немає print(char) (точного збігу) і немає print(int) (збігу шляхом неявного перетворення), символ b конвертується в тип float і зіставляється з print(float).

Зверніть увагу, всі стандартні перетворення вважаються рівними. Жодне з них не вважається вищим за інші по пріоритету.

Крок №4: **C++ намагається знайти відповідність шляхом користувацької конвертації**. Хоча ми ще не розглядали класи, але вони можуть визначати конвертації в інші типи даних, які можуть бути неявно застосовані до об’єктів цих класів. Наприклад, ми можемо створити клас W і в ньому визначити для користувача конвертацію в тип int:

```
1  class W; // з користувацькою конвертацією в тип int
2
3  void print(float value);
4  void print(int value);
5
6  W value; // оголошуємо змінну value типу класу W
7  print(value); // value конвертується в тип int і відповідає print(int)
```

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

Те, як робити користувацькі конвертації в класах, ми розглянемо на відповідних уроках.

Декілька збігів

Якщо кожна з перевантажених функцій повинна мати унікальні параметри, то як можуть бути можливі декілька збігів? Оскільки всі стандартні і користувацькі перетворення вважаються рівними, то, якщо виклик функції відповідає кільком кандидатам за допомогою стандартної або користувацької конвертації, результатом буде **неоднозначний збіг** (тобто декілька збігів). Наприклад:

```
void print(unsigned int value);
void print(float value);

print('b');
print(0);
  print(3.14159);
```

У випадку з print('b') мова C++ не може знайти точний збіг. Вона намагається конвертувати b в тип int, але версії print(int) також немає. Використовуючи стандартне перетворення, мова C++ може конвертувати b як в unsigned int, так і в float. Оскільки всі стандартні перетворення вважаються рівними, то виходить два збіги.

З print(0) все аналогічно. 0 — це int, а версії print(int) немає. Шляхом стандартного перетворення ми знову отримуємо два збіги.

А ось з print(3.14159) все трохи заплутаніше: більшість програмістів віднесуть його однозначно до print(float). Однак, пам’ятайте, що за замовчуванням всі значення-літерали типу з плаваючою крапкою відносяться до типу double, якщо у них немає закінчення f. 3.14159 — це значення типу double, а версії print(double) немає. Отже, ми отримуємо ту ж ситуацію, що і в попередніх випадках — неоднозначний збіг (два варіанти).

Неоднозначний збіг вважається помилкою типу compile-time. Відповідно, помилка має бути усунута до того, як ваша програма скомпілюється. Є два рішення цієї проблеми:

Рішення №1: Просто визначте нову перевантажену функцію, яка приймає параметри саме того типу даних, який ви використовуєте у виклику функції. Тоді мова C++ зможе знайти точний збіг.

Рішення №2: Явно конвертувати за допомогою **операторів явної конвертації** неоднозначний параметр(и) відповідно до типу функції, яку ви хочете викликати. Наприклад, щоб виклик print(0) відповідав print(unsigned int), вам потрібно зробити наступне:

```
print(static_cast<unsigned int>(0)); // відбудеться виклик print(unsigned int)
```

Висновки

Перевантаження функцій може значно знизити складність програми, в той же час створюючи невеликий додатковий ризик. Хоча цей урок трохи довгий і може здатися складним, але, насправді, перевантаження функцій зазвичай працює прозоро і без будь-яких проблем. Всі неоднозначні випадки компілятор позначатиме, і їх можна буде легко виправити.

Правило: Використовуйте перевантаження функцій для спрощення ваших програм.

Параметри за замовчуванням

Параметр за замовчуванням (або *«необов'язковий параметр»*) — це параметр функції, який має визначене (за замовчуванням) значення. Якщо користувач не передає в функцію значення для параметра, то використовується значення за замовчуванням. Якщо ж користувач передає значення, то це значення використовується замість значення за замовчуванням. Наприклад:

```
#include <iostream>

void printValues(int a, int b = 5)
{
	std::cout << "a: " << a << '\n';
	std::cout << "b: " << b << '\n';
}

int main()
{
	printValues(1); // в якості b буде використовуватися значення за замовчуванням - 5
	printValues(6, 7); // в якості b буде використовуватися значення, надане користувачем - 7
  }
```

Результат виконання програми:

a: 1
b: 5
a: 6
b: 7

У першому виклику функції ми не передаємо аргумент для b, тому функція використовує значення за замовчуванням — 5. У другому виклику ми передаємо значення для b, тому воно використовується замість параметра за замовчуванням.

Параметр за замовчуванням — це відмінний варіант, коли функція потребує значення, яке користувач може перевизначити, а може і не перевизначити. Наприклад, ось декілька **прототипів функцій**, для яких можуть використовуватися параметри за замовчуванням:

```
void openLogFile(std::string filename = "default.log");
int rollDie(int sides = 6);
  void printStringInColor(std::string str, Color color = COLOR_RED); // Color - це перерахування
```

Декілька параметрів за замовчуванням

Функція може мати декілька парамерів за замовчуванням:

```
void printValues(int a = 10, int b = 11, int c = 12)
{
	std::cout << "Values: " << a << " " << b << " " << c << '\n';
```

```
}
```

При наступних викликах функції:

```
printValues(3, 4, 5);
printValues(3, 4);
printValues(3);
  printValues();
```

Результат наступний:
Values: 3 4 5
Values: 3 4 12
Values: 3 11 12
Values: 10 11 12

Зверніть увагу, надати аргумент для параметру c, не надаючи при цьому аргументи для параметрів a і b — не можна (перестрибувати через параметри забороняється). Це пов'язано з тим, що мова C++ не підтримує наступний синтаксис виклику функції: printValues(,,5). З цього випливають наступні два правила:

Правило №1: Всі параметри за замовчуванням в прототипі або у визначенні функції повинні знаходитися праворуч. Наступне викличе помилку:

```
  void printValue(int a = 5, int b); // не дозволяється
```

Правильно:

```
  void printValue(int a, int b = 5);
```

Правило №2: Якщо є більше одного параметра за замовчуванням, то найлівішим параметром за замовчуванням повинен бути той, який з найбільшою ймовірністю (серед всіх інших параметрів) буде явно перевизначений користувачем.

Оголошення параметрів за замовчуванням

Як тільки параметр за замовчуванням оголошено, повторно оголосити його вже не можна. Це означає, що для функції з попереднім оголошенням і визначенням, параметр за замовчуванням оголосити можна або в попередньому оголошенні, або у визначенні функції, але не в обох місцях відразу. Наприклад:

```
void printValues(int a, int b = 15);

void printValues(int a, int b = 15) // помилка: перевизначення параметру за замовчуванням
{
      std::cout << "a: " << a << '\n';
      std::cout << "b: " << b << '\n';
  }
```

Рекомендується оголошувати параметри за замовчуванням в попередньому оголошенні, а не у визначенні функції, так як попередні оголошення можна використовувати в декількох файлах — при такому розкладі параметри за замовчуванням буде легше побачити (особливо, якщо попереднє оголошення знаходиться в заголовку). Наприклад:

boo.h:

```
#ifndef BOO_H
#define BOO_H
void printValues(int a, int b = 15);
  #endif
```

main.cpp:

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

```
void printValues(int a, int b)
{
	std::cout << "a: " << a << '\n';
	std::cout << "b: " << b << '\n';
}

int main()
{
	printValues(7);

	return 0;
  }
```

Зверніть увагу, в прикладі, наведеному вище, використовується параметр за замовчуванням b для функції printValues(), так як main.cpp підключає boo.h, який має попереднє оголошення функції printValues() з оголошеним параметром за замовчуванням.

Правило: Оголошуйте параметри за замовчуванням в попередньому оголошенні функції, в іншому випадку (якщо функція не має попереднього оголошення) — оголошуйте у визначенні функції.

Параметри за замовчуванням і перевантаження функцій

Функції з параметрами за замовчуванням можуть бути **перевантажені**. Наприклад:

```
void print(std::string string);
  void print(char ch = ' ');
```

Якщо користувач викличе просто print() (без параметрів), то виведеться пробіл, що буде результатом виконання print(' ').

Однак, варто зазначити, що параметри за замовчуванням НЕ відносяться до параметрів, які враховуються при визначенні унікальності функції. Відповідно, наступне не допускається:

```
void printValues(int a);
  void printValues(int a, int b = 15);
```

При виклику printValues(10) компілятор не зможе визначити, чи хочете ви викликати printValues(int) чи printValues(int, 15) (зі значенням за замовчуванням).

Висновки

Параметри за замовчуванням — це корисний механізм для вказування параметрів, при якому користувач може перевизначити значення за замовчуванням, або не перевизначати їх взагалі. Вони часто використовуються в мові C++.

Вказівники на функції

Розглянемо наступний фрагмент коду:

```
int boo()
{
	return 7;
  }
```

Ідентифікатор boo — це ім’я функції. Але який її тип? Функції мають свій власний **l-value** тип. У цьому випадку це тип функції, який повертає цілочисельне значення і не приймає ніяких параметрів. Подібно змінним, функції також мають свої адреси в пам’яті.

Коли функція викликається (за допомогою оператора ()), точка виконання переходить до адреси функції, що викликається:

```
int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
```

```
}

int main()
{
	boo(); // переходимо до адреси 002B1050

	return 0;
  }
```

Однією з найпоширеніших помилок початківців є:

```
#include <iostream>

int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
}

int main()
{
	std::cout << boo; // ми хочемо викликати boo(), але замість цього ми просто виводимо boo!

	return 0;
  }
```

Замість виклику функції boo() і виводу значення, що повертається, ми, абсолютно випадково, відправили вказівник на функцію boo() безпосередньо в std::cout. Що станеться в такому випадку?

Результат на моєму комп'ютері:

002B1050

У вас може бути і інше значення, в залежності від того, в який тип даних ваш компілятор вирішить конвертувати вказівник на функцію. Якщо ваш комп'ютер не вивів адресу функції, то ви можете змусити його це зробити, конвертуючи boo у **вказівник типу void** і відправляючи його на вивід:

```
#include <iostream>

int boo() // код функції boo() знаходиться в комірці пам'яті 002B1050
{
	return 7;
}

int main()
{
	std::cout << reinterpret_cast<void*>(boo); // вказуємо C++ конвертувати функцію boo() у вказівник типу void

	return 0;
  }
```

Так само, як можна оголосити неконстантний вказівник на звичайну змінну, можна оголосити і неконстантний вказівник на функцію. Синтаксис створення неконстантного вказівника на функцію, мабуть, один з найбільш "потворних" в мові C++:

```
// fcnPtr - це вказівник на функцію, яка не приймає ніяких аргументів і повертає цілочисельне значення
  int (*fcnPtr)();
```

У прикладі, наведеному вище, fcnPtr — це вказівник на функцію, яка не має параметрів і повертає цілочисельне значення. fcnPtr може вказувати на будь-яку іншу функцію, яка відповідає цьому типу.

Дужки навколо *fcnPtr необхідні для дотримання **пріоритету операцій**, в іншому випадку int *fcnPtr() буде інтерпретуватися як **попереднє оголошення** функції fcnPtr, яка не має параметрів і повертає вказівник на цілочисельне значення.

Для створення константного вказівника на функцію використовуйте **const** після зірочки:

```
int (* const fcnPtr)();
```

Якщо ви розмістите const перед int, то це означатиме, що функція, на яку вказує вказівник, повертає const int.

Присвоювання функції вказівника на функцію

Вказівник на функцію може бути ініціалізований функцією (і неконстантному вказівнику на функцію теж можна присвоїти функцію):

```
int boo()
{
	return 7;
}

int doo()
{
	return 8;
}

int main()
{
	int (*fcnPtr)() = boo; // fcnPtr вказує на функцію boo()
	fcnPtr = doo; // fcnPtr тепер вказує на функцію doo()

	return 0;
}
```

Одна з поширених помилок, яку роблять початківці:

```
fcnPtr = doo();
```

Тут ми фактично присвоюємо значення, що повертається з виклику функції doo(), вказівнику fcnPtr, чого ми не хочемо робити. Ми хочемо, щоб fcnPtr містив адресу функції doo(), а не значення, що повертається з функції doo(). Тому дужки тут не потрібні.

Зверніть увагу, у вказівника на функцію і самої функції повинні збігатися тип, параметри і тип значення, що повертається. Наприклад:

```
// Прототипи функцій
int boo();
double doo();
int moo(int a);

// Присвоювання значень вказівникам на функції
int (*fcnPtr1)() = boo; // ок
int (*fcnPtr2)() = doo; // не ок: тип вказівника і тип повернення функції не співпадають!
double (*fcnPtr4)() = doo; // ок
fcnPtr1 = moo; // не ок: fcnPtr1 не має параметрів, але moo() має
	int (*fcnPtr3)(int) = moo; // ок
```

На відміну від фундаментальних типів даних, мова C++ **неявно конвертує** функцію у вказівник на функцію, якщо це необхідно (тому вам не потрібно використовувати оператор адреси & для отримання адреси функції). Однак, мова C++ НЕ буде неявно конвертувати вказівник на функцію у вказівник типу void або навпаки.

Виклик функції через вказівник на функцію

Ви також можете використати вказівник на функцію для виклику самої функції. Є два способи зробити це. Перший — через явне розіменування:

```
int boo(int a)
{
	return a;
}

int main()
{
```

```
    int (*fcnPtr)(int) = boo; // присвоюємо функцію boo() вказівнику fcnPtr
    (*fcnPtr)(7); // викликаємо функцію boo(7), використовуючи fcnPtr

    return 0;
}
```

Другий — через неявне розіменування:

```
int boo(int a)
{
    return a;
}

int main()
{
    int (*fcnPtr)(int) = boo; // присвоюємо функцію boo() вказівнику fcnPtr
    fcnPtr(7); // викликаємо функцію boo(7), використовуючи fcnPtr

    return 0;
}
```

Як ви можете бачити, спосіб неявного розіменування виглядає так само, як і звичайний виклик функції, так як звичайні імена функцій є вказівниками на функції!

Примітка: **Параметри за замовчуванням** не працюватимуть з функціями, викликаними через вказівники на функції. Параметри за замовчуванням обробляються під час компіляції (тобто вам потрібно надати аргумент для параметра за замовчуванням під час компіляції). Однак вказівники на функції обробляються під час виконання. Отже, параметри за замовчуванням не можуть оброблятися при виконанні функції через вказівник на функцію. В цьому випадку вам потрібно буде явно передати значення для параметрів за замовчуванням.

Передача функцій в якості аргументів іншим функціям

Одна з найбільш корисних речей, яку ви можете зробити з вказівниками на функції — це передати функцію в якості аргументу іншій функції. Функції, які використовуються в якості аргументів для інших функцій, називаються **функціями зворотного виклику**.

Припустимо, що ви пишете функцію для виконання певного завдання (наприклад, сортування **масиву**), але ви хочете, щоб користувач міг визначити, яким чином виконувати це сортування (наприклад, по зростанню чи по спаданню). Розглянемо детально цей випадок.

Всі алгоритми сортування працюють за однаковою схемою: алгоритм виконує ітерацію по списку чисел, порівнює пари чисел і міняє їх місцями, виходячи з результатів цих порівнянь. Отже, змінюючи алгоритм порівняння чисел, ми можемо змінити спосіб сортування, не зачіпаючи при цьому інші частини коду.

Ось наше **сортування методом вибору**, розглянуте на відповідному уроці:

```
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

void SelectionSort(int* array, int size)
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex - це індекс найменшого елементу, який ми виявили до цього моменту
        int smallestIndex = startIndex;

        // Шукаємо найменший елемент серед решти елементів масиву (починаємо з startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше нашого попереднього знайденого найменшого елементу,
            if (array[smallestIndex] > array[currentIndex]) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ ТУТ
                // то це наш новий найменший елемент в цій ітерації
                smallestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим елементом
        std::swap(array[startIndex], array[smallestIndex]);
    }
```

```
}
```

Давайте замінимо порівняння чисел на функцію порівняння. Оскільки наша функція порівняння порівнюватиме два цілих числа і повертатиме **логічне значення** для вказівки того, чи слід виконувати заміну, вона виглядатиме наступним чином:

```
bool ascending(int a, int b)
{
    return a > b; // умова, при якій міняються місцями елементи масиву
}
```

А ось уже сортування методом вибору з функцією ascending() для порівняння чисел:

```
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

void SelectionSort(int* array, int size)
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // smallestIndex - це індекс найменшого елементу, який ми виявили до цього моменту
        int smallestIndex = startIndex;

        // Шукаємо найменший елемент серед решти елементів масиву (починаємо з startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше нашого попереднього знайденого найменшого елементу,
            if (ascending(array[smallestIndex], array[currentIndex])) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ
ТУТ
                // то це наш новий найменший елемент в цій ітерації
                smallestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим елементом
        std::swap(array[startIndex], array[smallestIndex]);
    }
}
```

Тепер, щоб дозволити caller-у вирішити, яким чином виконуватиметься сортування, замість використання нашої функції порівняння, ми дозволяємо caller-у надати свою власну функцію порівняння! Це робиться за допомогою вказівника на функцію.

Оскільки функція порівняння caller-а порівнюватиме два цілих числа і повертатиме логічне значення, то вказівник на цю функцію виглядатиме наступним чином:

```
bool (*comparisonFcn)(int, int);
```

Ми дозволяємо caller-у передавати спосіб сортування масиву за допомогою вказівника на функцію в якості третього параметру в нашу функцію сортування.

Ось готовий код сортування методом вибору з вибором способу сортування в caller-і (тобто в функції main()):

```
#include <iostream>
#include <algorithm> // для std::swap() (використовуйте <utility>, якщо підтримується C++11)

// Зверніть увагу, третім параметром є користувацький вибір виконання сортування
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int))
{
    // Перебираємо кожний елемент масиву
    for (int startIndex = 0; startIndex < size; ++startIndex)
    {
        // bestIndex - це індекс найменшого/найбільшого елементу, який ми виявили до цього
моменту
        int bestIndex = startIndex;
```

```
        // Шукаємо найменший/найбільший елемент серед решти елементів масиву (починаємо з
startIndex+1)
        for (int currentIndex = startIndex + 1; currentIndex < size; ++currentIndex)
        {
            // Якщо поточний елемент менше/більше нашого попереднього знайденого
найменшого/найбільшого елементу,
            if (comparisonFcn(array[bestIndex], array[currentIndex])) // ПОРІВНЯННЯ ВИКОНУЄТЬСЯ
ТУТ
                // то це наш новий найменший/найбільший елемент в цій ітерації
                bestIndex = currentIndex;
        }

        // Міняємо місцями наш стартовий елемент зі знайденим найменшим/найбільшим елементом
        std::swap(array[startIndex], array[bestIndex]);
    }
}

// Ось функція порівняння, яка виконує сортування в порядку зростання (зверніть увагу, це та ж
функція ascending(), що у прикладі, наведеному вище)
bool ascending(int a, int b)
{
    return a > b; // міняємо місцями, якщо перший елемент більше другого
}

// Ось функція порівняння, яка виконує сортування в порядку спадання
bool descending(int a, int b)
{
    return a < b; // міняємо місцями, якщо другий елемент більше першого
}

// Ця функція виводить значення масиву
void printArray(int* array, int size)
{
    for (int index = 0; index < size; ++index)
        std::cout << array[index] << " ";
    std::cout << '\n';
}

int main()
{
    int array[8] = { 4, 8, 5, 6, 2, 3, 1, 7 };

    // Сортуємо масив в порядку спадання, використовуючи функцію descending()
    selectionSort(array, 8, descending);
    printArray(array, 8);

    // Сортуємо масив в порядку зростання, використовуючи функцію ascending()
    selectionSort(array, 8, ascending);
    printArray(array, 8);

    return 0;
}
```

Результат виконання програми:

8 7 6 5 4 3 2 1

1 2 3 4 5 6 7 8

Прикольно, правда? Ми надали caller-у можливість контролювати процес сортування чисел (caller може визначити і будь-які інші функції порівняння):

```
bool evensFirst(int a, int b)
{
      // Якщо a – парне число, а b – непарне число, то a йде першим (ніякої перестановки не
відбувається)
      if ((a % 2 == 0) && !(b % 2 == 0))
            return false;

      // Якщо a – непарне число, а b – парне число, то b йде першим (тут вже потрібна
перестановка)
```

```
	if (!(a % 2 == 0) && (b % 2 == 0))
		return true;

	// В іншому випадку, сортуємо в порядку зростання
	return ascending(a, b);
}

int main()
{
	int array[8] = { 4, 8, 6, 3, 1, 2, 5, 7 };

	selectionSort(array, 8, evensFirst);
	printArray(array, 8);

	return 0;
}
```

Результат виконання програми:

2 4 6 8 1 3 5 7

Як ви можете бачити, використання вказівника на функцію дозволяє caller-у «підключити» свій власний функціонал до чогось, що ми писали і тестували раніше, що сприяє повторному використанню коду! Раніше, якщо ви хотіли відсортувати один масив в порядку спадання, а інший — в порядку зростання, вам знадобилося б написати кілька версій сортування масиву. Тепер же у вас може бути одна версія, яка виконуватиме сортування будь-яким способом, яким ви тільки захочете!

Параметри за замовчуванням у функціях

Якщо ви дозволите caller-у передавати функцію в якості параметру, то корисним буде надати і деякі стандартні функції для зручності caller-а. Наприклад, у вищенаведеному прикладі з сортуванням методом вибору, було б простіше встановити дефолтний (за замовчуванням) спосіб порівняння чисел. Наприклад:

```
// Сортування за замовчуванням виконується в порядку зростання
void selectionSort(int* array, int size, bool (*comparisonFcn)(int, int) = ascending);
```

У цьому випадку, до тих пір, поки користувач викликає selectionSort() як зазвичай (а не через вказівник на функцію), параметр comparisonFcn за замовчуванням відповідатиме функції ascending().

Вказівники на функції і псевдоніми типів

Подивимося правді в очі — синтаксис вказівників на функції потворний. Проте, за допомогою **typedef** ми можемо виправити цю ситуацію:

```
typedef bool (*validateFcn)(int, int);
```

Тут ми визначили псевдонім типу під назвою validateFcn, який є вказівником на функцію, яка приймає два значення типу int і повертає значення типу bool.

Тепер замість написання наступного:

```
bool validate(int a, int b, bool (*fcnPtr)(int, int)); // фу, який синтаксис
```

Ми можемо написати наступне:

```
bool validate(int a, int b, validateFcn pfcn) // ось це інша справа
```

Так набагато краще, правда? Однак синтаксис визначення самого typedef може бути трохи важким для запам’ятовування. У C++11 замість typedef ви можете використовувати псевдонім типу (type alias) для створення псевдоніма типу вказівника на функцію:

```
using validateFcn = bool(*)(int, int); // псевдонім типу
```

Це вже читабельніше, ніж з typedef, так як ім’я псевдонім і його визначення розташовані на протилежних сторонах від оператора =.

Використання type alias ідентичне використанню typedef:

```
bool validate(int a, int b, validateFcn pfcn) // круто, правда?
```

Використання std::function в C++11

У C++11 ввели альтернативний спосіб визначення і зберігання вказівників на функції, який виконується з використанням std::function. **std::function** є частиною заголовку functional Стандартної бібліотеки C++. Для визначення вказівника на функцію за допомогою цього способу вам потрібно оголосити об'єкт std::function наступним чином:

```
#include <functional>

  bool validate(int a, int b, std::function<bool(int, int)> fcn); // вказуємо вказівник на функцію за допомогою std::function, який повертає bool і приймає два int-а
```

Як ви можете бачити, тип повернення і параметри знаходяться в кутових дужках, а параметри ще і всередині круглих дужок. Якщо параметрів немає, то внутрішні дужки можна залишити порожніми. Тут вже зрозуміліше, який тип значення, що повертається, і які очікувані параметри функції.

Приклад задачі з використанням std::function:

```
#include <iostream>
#include <functional>

int boo()
{
	return 7;
}

int doo()
{
	return 8;
}

int main()
{
	std::function<int()> fcnPtr; // оголошуємо вказівник на функцію, який повертає int і не приймає ніяких параметрів
	fcnPtr = doo; // fcnPtr тепер вказує на функцію doo()
	std::cout << fcnPtr(); // викликаємо функцію як зазвичай

	return 0;
  }
```

Висновки

Вказівники на функції корисні, перш за все, коли ви хочете зберігати функції в масиві (або в **структурі**) або коли вам потрібно передати одну функцію в якості аргументу іншій функції. Оскільки синтаксис оголошення вказівників на функції є трохи потворним і вразливим до помилок, то рекомендується використовувати type alias (або std::function в C++11).

Еліпсис

Функції, які використовують еліпсис, виглядають наступним чином:

тип_повернення ім'я_функції(список_аргументів, ...)

список_аргументів — це один або декілька звичайних параметрів функції. Зверніть увагу, функції, які використовують еліпсис, повинні мати принаймні один параметр, який не є еліпсисом.

Еліпсис (англ. *"ellipsis"*), який представлений у вигляді трьох крапок … в мові C++, завжди повинен бути останнім параметром у функції. Про нього можна думати, як про **масив**, який містить будь-які інші параметри, крім тих, які вказані в список_аргументів.

Розглянемо приклад з використанням еліпсиса. Припустимо, що нам потрібно написати функцію, яка обчислює середнє арифметичне переданих аргументів:

```
#include <iostream>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром.
// Змінна count - це кількість переданих аргументів
double findAverage(int count, ...)
{
    double sum = 0;

    // Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
    va_list list;

    // Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який
потрібно ініціалізувати.
    // Другий параметр - це останній параметр, який не є еліпсисом
    va_start(list, count);

    // Перебираємо кожний аргумент еліпсиса
    for (int arg = 0; arg < count; ++arg)
        // Використовуємо va_arg для отримання параметрів з еліпсиса.
        // Перший параметр - це va_list, який ми використовуємо.
        // Другий параметр - це очікуваний тип параметрів
        sum += va_arg(list, int);

    // Виконуємо очищення va_list, коли вже зробили все необхідне
    va_end(list);

    return sum / count;
}

int main()
{
    std::cout << findAverage(4, 1, 2, 3, 4) << '\n';
    std::cout << findAverage(5, 1, 2, 3, 4, 5) << '\n';
  }
```

Результат виконання програми:

```
2.5
3
```

Як ви можете бачити, функція findAverage() приймає змінну count, яка вказує на кількість переданих аргументів. Розглянемо інші компоненти цього прикладу.

По-перше, ми повинні підключити заголовок cstdarg. Цей заголовок визначає va_list, va_start і va_end — макроси, необхідні для доступу до параметрів, які є частиною еліпсиса.

Потім ми оголошуємо функцію, яка використовує еліпсис. Пам'ятайте, що список_аргументів повинен бути представлений одним або декількома фіксованими параметрами. Тут ми передаємо одне цілочисельне значення, яке повідомляє функції, скільки буде параметрів.

Зверніть увагу, в еліпсисі немає ніяких імен змінних! Замість цього ми отримуємо доступ до значень через спеціальний тип — va_list. Про va_list можна думати, як про **вказівник**, який вказує на масив з еліпсисом. Спочатку ми оголошуємо змінну va_list, яку називаємо просто list для зручності використання.

Потім нам потрібно, щоб list вказував на параметри еліпсиса. Робиться це за допомогою va_start(), який має два параметри: va_list і ім'я останнього параметра, який не є еліпсисом. Після того, як va_start() був викликаний, va_list вказує на перший параметр зі списку переданих аргументів.

Щоб отримати значення параметра, на який вказує va_list, потрібно використати va_arg(), який також має два параметри: va_list і тип даних параметра, до якого ми намагаємося отримати доступ. Зверніть увагу, за допомогою va_arg() ми також переходимо до наступного параметру va_list!

Нарешті, коли ми вже все зробили, потрібно виконати очищення: va_end() з параметром va_list.

Чому еліпсис небезпечний?

Еліпсис дає програмісту велику гнучкість для реалізації функцій, які приймають змінну, що вказує на загальну кількість параметрів. Однак ця гнучкість має свої недоліки.

Зі звичайними параметрами функції компілятор використовує перевірку типів для гарантування того, що типи аргументів функції відповідають типам параметрів функції (або аргументи можуть бути **неявно конвертовані** для подальшої відповідності). Це робиться з метою запобігання випадкам, коли ви передаєте в функцію цілочисельне значення, тоді як вона очікує **рядок** (або навпаки). Зверніть увагу, параметри еліпсиса не мають оголошень типу даних. При їх використанні компілятор повністю пропускає перевірку типів даних. Це означає, що можна відправити аргументи будь-якого типу в еліпсис, і компілятор не зможе попередити вас, що це сталося. В кінцевому підсумку, ми отримаємо збій або неправильні результати. При використанні еліпсиса вся відповідальність лягає на caller, і від нього залежить коректність переданих аргументів в функцію. Очевидно, що це є хорошою лазівкою для виникнення помилок. Розглянемо приклад такої помилки:

```
std::cout << findAverage(6, 1.0, 2, 3, 4, 5, 6) << '\n';
```

Хоча на перший погляд все може здатися досить звичайним, але подивіться на другий аргумент типу double — він повинен бути типу int. Хоча все скомпілюється без помилок, але результат наступний:

1.78782e+08

Число не маленьке. Як це сталося?

Як ми вже знаємо з попередніх уроків, комп’ютер зберігає всі дані у вигляді послідовності біт. Тип змінної вказує комп’ютеру, як перевести цю послідовність біт в певне (читабельне) значення. Однак в еліпсисі тип змінної відкидається. Отже, єдиний спосіб отримати нормальне значення назад з еліпсису — вручну вказати va_arg(), який очікуваний тип параметра. Це те, що робить другий параметр в va_arg(). Якщо фактичний тип параметра не відповідає очікуваному типу параметра, то відбудуться погані речі.

У програмі, наведеній вище, за допомогою va_arg(), ми вказали, що всі параметри повинні бути типу int. Отже, кожен виклик va_arg() повертатиме послідовність біт, які будуть конвертовані в тип int.

У цьому випадку проблема полягає в тому, що значення типу double, яке ми передали в якості першого аргументу еліпсиса, займає 8 байт, тоді як va_arg(list, int) повертає тільки 4 байти даних при кожному виклику (тип int займає 4 байти). Отже, перший виклик va_arg повертає першу частину типу double (4 байти), а другий виклик va_arg повертає другу частину типу double (ще 4 байти). В результаті отримуємо сміття.

Оскільки перевірка типів пропущена, то компілятор навіть не скаржитиметься, якщо ми зробимо щось взагалі дике, наприклад:

```
int value = 8;
	std::cout << findAverage(7, 1.0, 3, "Hello, world!", 'G', &value, &findAverage) << '\n';
```

Вірте чи ні, але це дійсно скомпілювалося без помилок і видало наступний результат на моєму комп’ютері:

1.56805e+08

Цей результат підтверджує фразу: «Сміття на вході, сміття на виході».

Мало того, що еліпсис відкидає тип параметрів, він також відкидає і кількість цих параметрів. Це означає, що нам потрібно буде самим розробити рішення для відстеження кількості параметрів, що передаються в еліпсис. Як правило, це робиться одним з наступних трьох способів:

Спосіб №1: **Передати параметр-довжину**. Потрібно, щоб один з фіксованих параметрів, який не входить в еліпсис, відображав кількість переданих параметрів. Це рішення використовувалося у вищенаведеній програмі. Однак навіть тут ми зіткнемося з проблемами, наприклад:

```
std::cout << findAverage(6, 1, 2, 3, 4, 5) << '\n';
```

Результат на моєму комп’ютері:

4.16667

Що трапилося? Ми повідомили findAverage(), що збираємося передати 6 значень, але фактично передали тільки 5. Отже, з першими п'ятьма значеннями, що повертаються з функції va_arg(), все ок. Але ось 6-е значення, яке повертає va_arg() — це просто сміття зі **стеку**, так як ми його не передавали. Отже, і результат відповідний. Принаймні, тут очевидно, що це значення є сміттям. А ось розглянемо більш *підступний* випадок:

```
std::cout << findAverage(6, 1, 2, 3, 4, 5, 6, 7) << '\n';
```

Результат:
3.5

На перший погляд все коректно, але останнє число (7) в списку аргументів ігнорується, так як ми повідомили, що збираємося надати 6 параметрів (а надали 7). Такі помилки буває досить-таки важко виявити.

Спосіб №2: **Використовувати контрольне значення**. Контрольне значення — це спеціальне значення, яке при його виявленні використовується для завершення циклу. Наприклад, нуль-термінатор використовується в рядках для позначення кінця рядка. У еліпсиса контрольне значення передається останнім з аргументів. Ось програма, наведена вище, але вже з використанням контрольного значення -1:

```
#include <iostream>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром
double findAverage(int first, ...)
{
	// Обробка першого значення
	double sum = first;

	// Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
	va_list list;

	// Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який потрібно ініціалізувати.
	// Другий параметр - це останній параметр, який не є еліпсисом
	va_start(list, first);

	int count = 1;
	// Безкінечний цикл
	while (1)
	{
		// Використовуємо va_arg для отримання параметрів з еліпсису.
		// Перший параметр - це va_list, який ми використовуємо.
		// Другий параметр - це очікуваний тип параметрів
		int arg = va_arg(list, int);

		// Якщо поточний параметр є контрольним значенням, то припиняємо виконання циклу
		if (arg == -1)
			break;

		sum += arg;
		count++;
	}

	// Виконуємо очищення va_list, коли вже зробили все необхідне
	va_end(list);

	return sum / count;
}

int main()
{
	std::cout << findAverage(1, 2, 3, 4, -1) << '\n';
	std::cout << findAverage(1, 2, 3, 4, 5, -1) << '\n';
  }
```

Зверніть увагу, нам вже не потрібно явно передавати довжину в якості першого параметру. Замість цього ми передаємо контрольне значення в якості останнього параметру.

Однак тут також є нюанси. По-перше, мова C++ вимагає, щоб ми передавали хоча б один фіксований параметр. У попередньому прикладі для цього використовувалася змінна count. У цьому прикладі перше значення є частиною чисел, використовуваних в обчисленні. Тому, замість обробки першого значення в парі з іншими параметрами еліпсиса, ми явно оголошуємо його як звичайний параметр. Потім нам потрібно це обробити усередині функції (ми присвоюємо змінній sum значення first, а не 0, як у попередній програмі).

По-друге, потрібно, щоб користувач передав контрольне значення останнім у списку. Якщо користувач забуде передати контрольне значення (або передасть неправильне), то функція циклічно працюватиме до тих пір, поки не дійде до значення, яке відповідатиме контрольному (яке не було вказано), тобто сміттю (або станеться збій).

Спосіб №3: **Використовувати рядок-декодер.** Передайте рядок-декодер в функцію, щоб повідомити, як правильно інтерпретувати параметри:

```
#include <iostream>
#include <string>
#include <cstdarg> // потрібно для використання еліпсиса

// Еліпсис повинен бути останнім параметром
double findAverage(std::string decoder, ...)
{
	double sum = 0;

	// Ми отримуємо доступ до еліпсису через va_list, тому оголошуємо змінну цього типу
	va_list list;

	// Ініціалізуємо va_list, використовуючи va_start. Перший параметр - це список, який необхідно ініціалізувати.
	// Другий параметр - це останній параметр, який не є еліпсисом
	va_start(list, decoder);

	int count = 0;
	// Безкінечний цикл
	while (1)
	{
		char codetype = decoder[count];
		switch (codetype)
		{
		default:
		case '\0':
			// Виконуємо очищення va_list, коли вже зробили все необхідне
			va_end(list);
			return sum / count;

		case 'i':
			sum += va_arg(list, int);
			count++;
			break;

		case 'd':
			sum += va_arg(list, double);
			count++;
			break;
		}
	}
}

int main()
{
	std::cout << findAverage("iiii", 1, 2, 3, 4) << '\n';
	std::cout << findAverage("iiiii", 1, 2, 3, 4, 5) << '\n';
	std::cout << findAverage("ididdi", 1, 2.2, 3, 3.5, 4.5, 5) << '\n';
  }
```

У цьому прикладі ми передаємо рядок, в якому вказується як кількість переданих аргументів, так і їх типи (i = int, d = double). Таким чином, ми можемо працювати з параметрами різних типів. Однак слід пам’ятати, що якщо число або типи переданих параметрів не будуть з точністю відповідати тому, що зазначено в рядку-декодері, то можуть відбутися погані речі.

Рекомендації по безпечному використанню еліпсиса

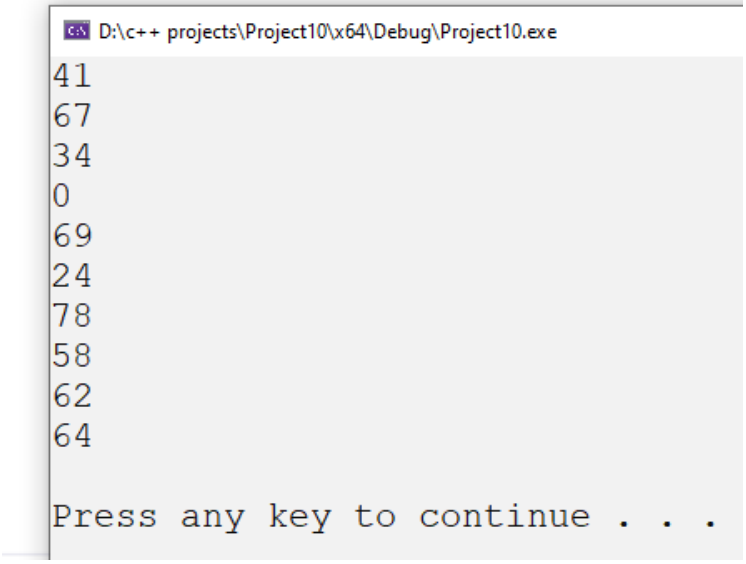
По-перше, якщо це можливо, не використовуйте еліпсис взагалі! Часто доступні інші розумні рішення, навіть якщо вони вимагають трохи більше роботи і часу. Наприклад, у функції findAverage() у вищенаведеній програмі ми могли б передати **динамічно виділений масив** цілих чисел, замість використання еліпсиса. Це забезпечило б перевірку типів (гарантуючи, що caller не спробує зробити щось безглузде), зберігаючи при цьому можливість передавати змінну-довжину, яка вказувала б на кількість всіх переданих значень.

По-друге, якщо ви використовуєте еліпсис, не змішуйте різні типи аргументів в межах вашого еліпсиса, якщо це можливо. Це зменшить ймовірність того, що caller випадково передасть дані не того типу, а va_arg() видасть результат-сміття.

По-третє, використання параметра count або рядка-декодера в якості список_аргументів зазвичай безпечніше, ніж використання контрольного значення. Це гарантує, що цикл еліпсиса буде завершено після чітко визначеної кількості ітерацій.

1. Передача масиву функції. Передається адреса масиву.

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

int* mass(int *a)
{
	for(int i=0; i<10;i++)
	{
		a[i] = rand() % 100;
	}
	return a;
}

int main()
{

	int b[10];
	mass(b);

	for (int i = 0; i < 10; i++)
	{
		cout << b[i]<<endl;
	}

	cout << endl;
	system("pause");
	return 0;
}
```



2. Попередній код має зайвий код. Головно повернення з функції вказівника. Це зайве, оскільки функція ітак працює по адресах переданих з головної програми

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

void mass(int *a)
{
	for(int i=0; i<10;i++)
	{
		a[i] = rand() % 100;
	}
}

int main()
{
	int b[10];
	mass(b);

	for (int i = 0; i < 10; i++)
	{
		cout << b[i]<<endl;
	}

	cout << endl;
	system("pause");
	return 0;
	}
```

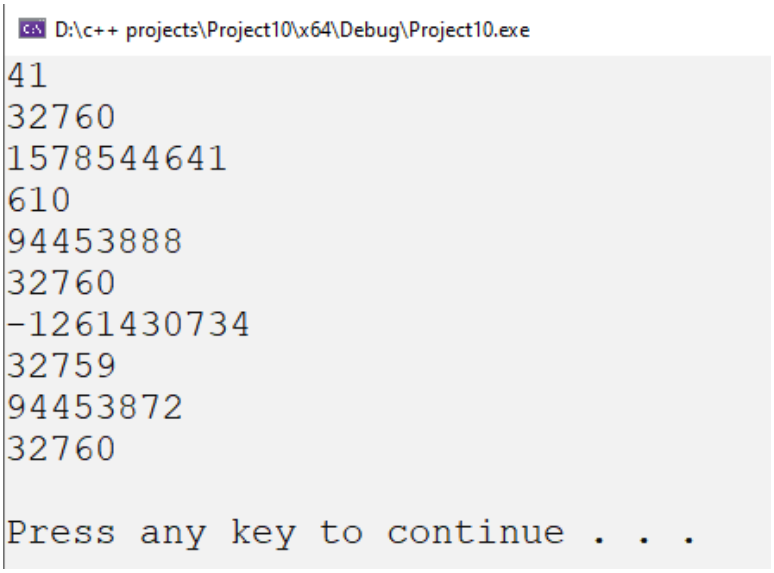
3. Недолік коду вище, можна віднести те, що слід проаналізувати весь код, щоб зрозуміти, що в якості аргумента функції передається вказівник саме на масив. Під вказівником може бути і одиничне значення, хоча синтаксично, код цілком правильний. Тому стандарт дозволяє інший механізм передачі масиву функції, і це відразу дає розуміння для програміста, що передається масив.

Замість `void mass(int *a)` прописати `void mass(int a[])`.

4. Повернення масивів з функцій доцільне, коли ці масиви створює сама функція. В якості аргумента можна передавати розмірність масиву.

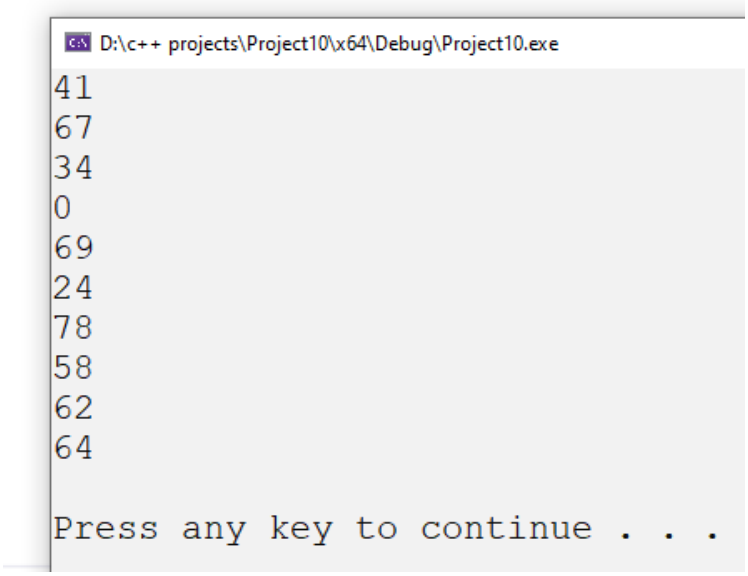
В коді виводиться сміття. Оскільки доступ до створеного масиву втрачається після виходу з функції.

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

int*  mass(int a)
{
	int A[20];
	for(int i=0; i<a;i++)
	{
		A[i] = rand() % 100;
	}
	return A;
}

int main()
{
	int *b=mass(10);

	for (int i = 0; i < 10; i++)
	{
		cout << b[i]<<endl;
	}

	cout << endl;
	system("pause");
	return 0;
}
```



5. Виправлення ситуації – динамічний масив. І не забути очистити пам’ять.

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

int*  mass(int a)
{
	int *A = new int[a];
	for(int i=0; i<a;i++)
	{
		A[i] = rand() % 100;
	}
	return A;
}

int main()
{
	int *b=mass(10);

	for (int i = 0; i < 10; i++)
	{
		cout << b[i]<<endl;
	}

	cout << endl;
	system("pause");
	delete[] b;
	return 0;
}
```



Символи і рядки

При виконуванні програм комп'ютер витрачає, за деякими оцінками, до 70 % часу на маніпулювання з текстовими рядками: копіює їх з одного місця пам'яті в інше, перевіряє наявність у рядках певних слів, поєднує чи відсікає рядки тощо. У C++ є два основні види рядків: С-рядки, які по суті є символьними масивами, і клас стандартної бібліотеки C++ `string`.

1 Символьний тип даних

Символами вважаються: великі й малі літери, цифри, знаки арифметичних дій ('+', '-', '*', '/', '='), пробіл, розділові знаки (`'.'`, `','`, `';'`, `':'`, `'!'`, `'?'`, `' -'`), службові символи, що відповідають клавішам <Enter>, <Esc>, <Tab> тощо. В C++ значення символьних констант записуються у одинарних лапках: `'3'`, `'f'`, `'+ '`, `'%'`.

Для кодування усіх символів використовується восьмирозрядна послідовність 0 і 1, тобто один байт. Наприклад: символ цифри `'9'` кодується послідовністю бітів 0011 1001, символ літери латиниці `'W'` - 0101 0111. За допомогою одного байта можна закодувати 256 різних комбінацій бітів, а отже, 256 різних символів.

Щоб не було розходжень у кодуванні символів, існує єдиний міжнародний стандарт - так звана таблиця ASCII-кодів (American Standard Code for Information Interchange - американський стандартний код для обміну інформацією, див. додаток А). Символи ASCII мають коди від 0 до 127, тобто значення першої половини можливих значень байта, хоча часто кодами ASCII називають всю таблицю з 256 символів. Перші 128 ASCII-кодів є єдині для всіх країн, а коди від 128 до 255 називають розширеною частиною таблиці ASCII, де залежно від країни розташовується національний алфавіт і символи псевдографіки.

У таблиці ASCII всі символи пронумеровано, тобто вони мають власний унікальний код. Так само як у кожній мові людського спілкування існує алфавіт (перелік усіх літер у чітко визначеному порядку), усі комп'ютерні символи теж є суворо упорядкованими. Символ `' '` (пробіл) має код 32, цифри мають коди від 48 для `'0'` до 57 для `'9'`, великі латинські літери - від 65 для `'A'` до 90 для `'Z'`, малі літери латиниці - від 97 для `'a'` до 122 для `'z'`. Кодування другої половини таблиці ASCII має різні варіанти. Найпоширенішими є DOS-кодування (866 кодова сторінка) і кодування 1251, яке є основним для Windows (див. додаток А). Звернімо увагу на різницю поміж цифрами і їхнім символьним зображенням. Наприклад, символ цифри `'4'` має ASCII-код 52 і не має безпосереднього відношення до числа 4.

Керувальні символи таблиці ASCII не мають символьного подання, тобто не мають візуального зображення, тому їх інколи називають недрукованими (non-printed), наприклад: <Esc>, <Enter>, <Tab> тощо. Ці символи розташовано в перших 32-х кодах таблиці ASCII-кодів. Звертатися до таких символів можна через їхній код чи за допомогою так званої ескейп-послідовності (escape). Ес- кейп-послідовність - це спеціальна комбінація символів, яка розпочинається зі зворотної скісної риси і записується в одинарних лапках (табл. 7.1), наприклад: `'\0'`, `'\n'`. Кожна з наведених у таблиці комбінацій символів вважається за один символ.

Таблиця 1 *Деякі поширені у застосуванні ескейп-послідовності*

Символьне подання	**Опис**
`\n`	символ переведення курсора на початок наступного рядка
`\r`	переведення каретки
`\t`	символ переведення курсора на наступну позицію табуляції (відповідає клавіші <Tab>),
`\b`	символ вилучення попереднього символу перед курсором (відповідає клавіші <BackSpace>),
`\a`	символ звукового сигналу системного динаміка
`\\`	символ \ (зворотна скісна риса)
`\?`	символ ? (знак запитання)
`\'`	символ ' (одинарні лапки)
`\”`	символ " (подвійні лапки)
`\0`	символ з кодом 0 є завершальним символом рядка символів
`\ 0x`Вісімкове число	символ, код якого зазначено у вісімковій системі числення
`\ 0x`Шістнадцяткове число	символ, код якого зазначено у шістнадцятковій системі числення

Тип символьних змінних у C++ називається **`char`**. Так, при оголошенні

```
char c, s, g;
```

в оперативній пам’яті для кожної з цих трьох змінних буде відведено по одному байту. Коли символьні змінні набувають певних значень, то комп’ютер зберігатиме в пам’яті не власне символи, а їхні коди. Наприклад, замість літери `'A'` зберігатиметься її код 65. Тому, якщо присвоїти символьній змінній певне число, то C++ сприйме його як код символу з таблиці ASCII- кодів. Це поширюється лише на цілі числа. Зважаючи на різні кодування розширеної частини ASCII-таблиці для уникнення помилок вважається за оптимальне при роботі з символами використовувати їхнє символьне подання замість кодів. Наприклад, при оголошенні `char c = 115;`
змінна `c` набуде значення символу `'s'`, який має код 115.

Зауважимо, що у C++, на відміну від більшості мов програмування, дані типу `char` змінюються у діапазоні -128 ... 127, причому додатні числа 0 ... 127
зайняті символами спільної частини ASCII-таблиці, а символи розширеної частини ASCII-таблиці у C++ відповідають від’ємним числам. Наприклад, літера кирилиці `'ч'` - має код -9, а кодом літери `'я'` є -1.

Окрім типу `char`, існує його беззнакова модифікація `unsigned char`. Дані типу `unsigned char` мають значення у діапазоні 0 ... 255. В ASCII-таблиці значення кодів літер кирилиці є більшими за 127, тому, якщо треба мати справу зі змінними, значеннями яких є літери кирилиці, їх слід оголошувати типом `unsigned char`:
`unsigned char c;`

Символи можна порівнювати. Більшим вважається той символ, у якого код є більший, тобто символ, розташований у таблиці ASCII-кодів пізніше.

Наприклад: `'a'<'h', 'A'<'a'`.

Оскільки символьний тип `char` вважається у С++ за цілий тип, змінні цього типу можна додавати й віднімати. Результатом додавання буде символ, код якого дорівнює сумі кодів символів-доданків, наприклад:

```
char c = 'A';
char c1          = c +    5;  //  c1    = 'F'
char c2          = c +   32;  //  c2    = 'a'
char c3          = c -   10;  //  c2    = '7'
```

У С++ існує ціла низка спеціальних функцій для роботи з символьними даними. Деякі з цих функцій наведено у табл. 7.2.

Таблиця 2 *Функції для роботи з символами*

Функція	Призначення
`tolower()`	повертає символ у нижньому регістрі
`toupper()`	повертає символ у верхньому регістрі
Належність символу до множини перевіряють такі функції:	
`isalnum()`	латинських літер та цифр (`'A' - 'Z'`, `'a' - 'z'`, `'0' - '9'`)
`isalpha ()`	латинських літер (`'A' - 'Z'`, `'a' - 'z'`)
`iscntrl ()`	керувальних символів (з кодами 0...31 та 127)
`isdigit ()`	цифр (`'0' - '9'`)
`isgraph ()`	видимих символів, тобто не є відповідним до клавіш <Esc>, <Tab> тощо
`islower ()`	латинських літер нижнього регістру (`'a' - 'z'`)
`isprint ()`	друкованих символів (`isgraph ()` + пробіл)
`isupper ()`	літер верхнього регістру (`'A' - 'Z'`)
`ispunct ()`	знаків пунктуації
`isspace ()`	символів-роздільників

Розглянемо деякі *поширені алгоритми опрацювання символьних змінних.*

1) Щоб визначити код символу, треба значення цього символу присвоїти цілій змінній. І, навпаки, щоб дізнатися, який символ відповідає певному числу, слід це число присвоїти символові. С++ сам виконає потрібні перетворення. Наприклад, після виконання команд `int x; char c = 'n';`
`x = c;`
змінна `x` набуде значення 110, яке відповідає коду символу `'n'` в ASCII- таблиці.

2) Визначити, чи є символ `c` **цифрою**, можна двома способами: перевірити його належність до символьного проміжку від `'0'` до `'9'` за допомогою умови:
`if(c>='0' && c<='9') . . .`
або застосувати функцію `isdigit ()` для перевірки належності символу до множини цифр:
`if(isdigit (c)) . . .`

3) Дізнатися, чи є символ `c` **великою латинською літерою**, можна теж двома способами: перевірити його належність до символьного проміжку від 'A' до 'Z' за допомогою умови:
`if(c>='A' && c<='Z')  . . .`
або застосувати функцію `isupper()` для перевірки належності символу до множини

великих латинських літер:
```
if(isupper (c)) . . .
```
4) Перевірку, чи є символ `c` **латинською літерою**, можна здійснити за допомогою умови
```
if(c>='A' && c<='Z'              || c>='a' && c<='z')   . . .
```
або за допомогою функції `isalpha ()`: `if(isalpha (c)) . . .`

5) Для перевірки, чи є символ `c` **малою латинською літерою**, слід записати умову
```
if(c>='a' && c<='z')             . . .
```
або використати функцію `islower()` для перевірки належності символу до множини малих латинських літер:
```
if(islower(c))             . . .
```
6) Для перетворення малої латинської літери `c` **на велику** (верхній регістр) можна використати функцію `toupper ()`:
```
c = toupper(c);
```
чи то відняти від значення літери різницю кодів між відповідними великими і малими літерами (вона становить 32):
```
char c = c - 32;
```
У аналогічний спосіб працює функція `tolower ()`, яка збільшує код великих літер латиниці на 32 й, отже, здобуває **нижній регістр** літер латиниці.

Примітка. Вищезазначені функції перевірки й перетворення регістру працюють лише з латинськими літерами. Для виконання дій 1 - 5 з літерами кирилиці слід використовувати перевірку належності символу до відповідного символьного проміжку, а для перетворення регістра - зменшення або збільшення коду символу на різницю кодів між великими і малими літерами (для більшості символів вона теж становить 32).

2 Рядки

Рядки у С являють собою послідовність (масив) символів із завершальним нуль-символом. Нуль-символ (нуль-термінатор) - це символ з кодом 0, який записується у вигляді керувальної послідовності '\0'. За розташуванням нуль-символу визначається фактична довжина рядка.

2.1 Масиви символів

Відмінною рисою символьного масиву є те, що в ньому насправді може бути менше символів, аніж зазначено при оголошенні. Окрім того, з цими масивами можна виконувати певні специфічні дії, які не можна здійснювати з числовими масивами (наприклад перевіряти наявність у масиві літери чи послідовності літер, копіювати масив як одне ціле, порівнювати масиви за алфавітом, дописувати один масив наприкінці іншого тощо).

Пам’ять під розміщення рядків, як і для будь-яких масивів, може виділятися як компілятором, так і динамічно - при виконуванні програми. Довжина динамічного рядка може задаватися змінною з визначеним заздалегідь значенням, а довжина статичного рядка має задаватися лише константою.

Рядок може бути оголошеним в один з нижче наведених способів:

1) `char *s;` *// Оголошення вказівника на перший символ рядка;*
// пам’ять під сам рядок не виділяється

2) `char ss[15];` *// Оголошення рядка ss з 15-ти символів;*

```
// пам'ять виділяється компілятором
```
3)
```
    const int n = 10;
char st[n];                    // Оголошення рядка st з n (тобто 10-ти) символів;
// пам'ять виділяється компілятором
```
4)
```
    int n = 10;
char *str = new char[n]; // Оголошення рядка str з n (тобто 10)
// символів; пам 'ять виділяється динамічно
```

При оголошенні рядок можна ініціалізувати рядковою константою, при цьому нуль-символ формується автоматично після останнього символу: `char str[10] = "Vasia";`

При цьому виділиться пам'ять під масив з 10-ти елементів та нуль-символ і перші 5 символів рядка записуються в перші 5 байт цієї пам'яті (`str[0] = 'V'`, `str[1] = 'a'`, `str[2] = 's'`, `str[3] = 'i'`, `str[4] = 'a'`), а в шостий елемент `str[5]` записується нуль-символ. Якщо рядок при оголошенні ініціалізується, його розмірність можна опускати (компілятор сам виділить потрібну кількість байтів):

```
char str[] = "Vasia"; // Виділено й заповнено 6 байтів
```

Рядки у лапках завжди неявно містять нуль-символ, тому при ініціалізації прописувати його немає потреби. Окрім того, різні наведені способи введення символьних масивів автоматично долучають нуль-символ у кінець масиву.

При оголошенні й ініціалізації масиву слід бути впевненим, що розмір масиву є достатній, щоб умістити всі символи рядка з нуль-символом. Річ у тім, що функції, які опрацьовують рядки, керуються позицією нуль-символу, а не розміром рядка. C++ не накладає жодних обмежень на довжину рядка.

Звернімо увагу на те, що рядкова константа (у подвійних лапках) і символьна константа (в одинарних лапках) не є взаємозамінними. Це константи різних типів. Символ у одинарних лапках, наприклад, `'s'` є *символьною* константою. Для зберігання такої константи компілятор C++ виділяє лише один байт пам'яті. Символ у подвійних лапках, наприклад, `"s"` є *рядковою* константою, що окрім символу `'s'` містить символ `'\0'`, який долучається компілятором. Більш того, `"s"` фактично являє собою адресу пам'яті, в якій зберігається рядок.

Як і числові масиви, символьні масиви опрацьовуються поелементно у циклі. Операція присвоювання одного рядка іншому є невизначена (оскільки рядок є масивом) і може виконуватися за допомогою циклу чи за допомогою функцій стандартної бібліотеки.

Для *введення* й *виведення* рядків у консолі використовуються **`cin`** і **`cout`**, наприклад:

```
const int n=10; char s[n];
cin>>s;
cout<<s;
```

Коли рядок виводиться за допомогою потоку `cout`, символи рядка виводяться по одному, допоки не зустрінеться завершальний символ `'\0'`.

При введенні рядків у консолі замість оператора >> більш оптимально використовувати метод **`getline`** `()`, оскільки потоковий оператор введення >> ігнорує пробіли. Окрім того, він може продовжувати введення елементів за межами масиву, якщо в пам'яті під рядок виділено менше місця, аніж вводиться символів.

Функція `getline ()` має два параметри: перший аргумент - рядок, який вводиться, а другий - кількість символів.
Наприклад:

```
char s[4];
cout<<"Введіть рядок: "<<endl;
cin.getline(s, 4);
```

Аналогічні дії можна зробити використовуючи функцію gets_s:

```
char s[4];
cout<<"Введіть рядок: "<<endl;
gets_s(s);
```

Рядок `s` у цьому фрагменті програми може прийняти лише три значущих символи і буде завершений нуль-термінатором (символом `'\0'`). Решту введених символів рядка буде проігноровано.

Поширеним засобом доступу до символів рядка є вказівники типу `char*`.
У прикладі

```
char *st = "Комп'ютерна програма";
```

компілятор записує всі символи рядка до масиву і присвоює змінній `st` адресу першого елемента масиву.

Рядок може вважатися за порожній у двох випадках: якщо вказівник на рядок має нульове значення `NULL` (немає взагалі жодного рядка) чи коли вказівник вказує на масив, який складається з одного нульового символу (не містить жодного значущого символу).

```
char *pc1 = 0;                  // pc1 не адресує жодного масиву символів,
const char *pc2 = "";           // pc2 адресує нульовий символ
```

Функції стандартної бібліотеки для роботи з рядками

C++ має багату колекцію функцій опрацювання рядків із завершальним нулем. Якщо в рядку відсутній нуль-термінатор, опрацювання рядка може тривати скільки завгодно, допоки в пам'яті не зустрінеться `'\0'`. У якості аргументів до функцій переважно передаються вказівники на рядки. Якщо при виконуванні функції здійснюється перенесення символів рядка з місця-джерела до місця-призначення, для рядка-призначення слід завчасно зарезервувати місце в пам'яті. Копіювання рядків з використанням просто вказівника, а не адреси початку завчасно оголошеного масиву - одна з найпоширеніших помилок програмування, навіть у досвідчених програмістів. При виділенні місця для рядка- призначення слід виділяти місце і для нуль-термінатора.

У табл. 7.3 наведено функції стандартної бібліотеки для роботи з С-рядками. Деякі з цих функцій, наприклад `strlen ()`, `strcpy ()`, опрацьовують рядки, оголошені в будь-який з чотирьох раніш розглянутих способів. Але більшість функцій потребує, щоб рядок був оголошений як вказівник типу `char*` (див. способи 1 і 4 на початку).

У консолі для використання цих функцій слід залучити до програми бібліотеку <cstring>. Перелік основних функцій цієї та інших стандартних бібліотек C++ наведено в таблиці

Таблиця 3 *Функції стандартної бібліотеки <cstring>*

Функція	**Призначення**	**Формат**
strlen()	повертає довжину рядка (без урахування символу завершення рядка)	size t **strlen** (char *s) ;
strcat()	долучає s2 до s1 і, як результат, повертає s1	char ***strcat**(char *s1, char *s2);
strncat()	долучає до рядка s1 перші n символів з рядка s2	char * **strncat** (char *s1, char *s2, size t n);
strlwr()	перетворює всі латинські літери до нижнього регістру	char ***strlwr**(char *s);
strupr()	перетворює всі латинські літери до верхнього регістру	char ***strupr**(char *s);
strcmp()	порівнює рядки і повертає нульове значення, якщо рядки є однакові (s1=s2), від'ємне (s1<s2) чи додатне (s1>s2). Порівняння відбувається посимвольно і в якості результату повертається різниця кодів перших неоднакових символів	int ***strcmp**(char *si, char *s2);
strncmp()	на відміну від попередньої функції, порівнює лише перші n символів рядка si з n символами рядка s2	int ***strncmp** (char *si, char *s2, size t n);
stricmp()	порівнює два рядки, не розрізнюючи прописні й малі літери латиниці	int **stricmp** (char*si, char *s2);
strnicmp()	порівнює лише перші n символів двох рядків, не розрізняючи великі й малі літери латиниці	int **strnicmp** (char *si, char *s2, size t maxlen);
strcpy()	копіює до si рядок s2, повертає si, при цьому попереднє значення si втрачається	char ***strcpy** (char *si, char *s2);
strncpy()	замінює перші n символів рядка s1 на перші n символів рядка s2	char ***strncpy** (char *si, char *s2, size t n);
strchr()	відшукує перше входження символу ch в рядок s і повертає вказівник на цей символ, тобто частину рядка s, розпочинаючи з символу ch і до кінця рядка. Якщо символу ch в рядку s немає, результат - NULL	char ***strchr**(char *s, int ch);

strrchr()	відшукує останнє входження символу в рядку і повертає частину рядка s, розпочинаючи з останнього входження символу ch і до кінця рядка. Якщо символу ch в рядку s немає, результат - NULL	char *strrchr (char *s, int c);
strstr()	відшукує підрядок s2 в рядку si, повертає частину рядка si, розпочинаючи з першого спільного символу для обох рядків і до кінця si	char *strstr(char *si, char *s2);
strspn()	повертає довжину початкового сегмента рядка si, символи якого є в рядку s2	size t strcspn (char *si, char *s2);
strcspn()	повертає індекс першого символу в рядку si, який є спільним для обох рядків (нумерація індексів символів розпочинається з нуля)	size t strcspn (char *si, char *s2);
strset()	замінює усі символи рядка s на символ c	char *strset(char *s, char c);
strnset()	замінює лише перші n символів рядка s на символ c	char *strnset (char *s, int ch, size t n);
strpbrk()	відшукує місце першого входження у рядок s1 будь-якого символу рядка s2 і повертає частину рядка s1, розпочинаючи з цього символу і до кінця рядка	char *strpbrk (char *s1, const char *s2);
strrev()	реверс рядка s	char *strrev (char *s);
strtok()	повертає частину (лексему) рядка s1 від поточної позиції вказівника до розділового символу, зазначеного у рядку s2; зазвичай використовується для перетворювання рядка на послідовність лексем	char *strtok(char *s1, const char *s2);

Розглянемо деякі з наведених у табл. 3 функцій детальніше на прикладах, для чого попередньо оголосимо вказівники на рядки s1, s2 та s3 і надамо їм початкові значення.

```
int k, n; char       *s1 = "На Дерибасівській",*s2="гарна погода",*s3;
              n = strlen(s1);      // Обчислюється довжина рядка si; n дорівнюватиме 17
       k = strlen(s2);      // Обчислюється довжина рядка s2; k=13
       k = strcmp(s1, s2);  // k=173; різниця між ASCII-кодами перших неоднакових
                                  // символів рядків s1 і s2 дорівнює 173, а оскільки 173>0,
                                        // можна стверджувати, що s1>s2
       strcpy(s3, s2);      // s3=" гарна погода" (рядку s3 присвоюється рядок s2)
       strncpy(s3, s1, 2);  // Копіюються два перші символи з рядка s1 до s3
       s3[2] = '\ 0';       // Третім символом після символів"На" задається
                            // нуль-символ щоби обрізати рядок
       strcat(s1, s2);      // s1 = "На Дерибасівській гарна погода"; до рядка s1
                                        // долучається рядок s2
```

```
s3 = strchr(s1, ’ ’); // Пошук пробілів у рядку s1; тепер s3=" Дерибасівській
                      //гарна погода" - це рядок від першого пробілу до кінця
                      // рядка s1, тобто без першого слова
s3=strrchr (s1, ’ ’);// Пошук останнього пробілу в рядку s1;
                      // тепер s3=" погода" - це останнє слово рядка s1
n=strchr (s1, ’ ’)-s1+1;// n=3 - індекс першого пробілу в рядку s1, обчислений
                      // як різниця вказівників
k = strcspn(s1, ” ”) + 1;// k=3 - індекс першого пробілу в рядку s1; оскільки
                     // нумерація індексів символів розпочинається з 0, слід додати 1
s3=strstr(s1,s2); // s3 = "гарна погода"; пошук рядка s2 у рядку s1
s3=strrev (s2);     // s3="адогоп анраг" - відбувається реверс рядка
s3 = strupr(”C++ ms”); // s3=”C+ + MS ” - перетворення усіх
                       // латинських літер рядка до верхнього регістру
s3 = strlwr(s3);    // s3="c+ + ms" – зворотне перетворення усіх
                                  // латинських літер рядка до нижнього регістру
s3 = strtok(s1, ” ”); // s3="На" - перше слово (лексема) рядка s1 до розділового
                      // знака, який зазначено символом (пробілом) у другому
  // аргументі функції. Почергове здобуття усіх лексем рядка можна організовувати у циклі
```

Отже, функції `strcpy ()` та `strncpy ()` призначено для копіювання рядка чи то його частини до іншого рядка. Функції `strchr()`, `strrchr()` та `strstr()` повертають вказівник на віднайдений символ чи підрядок.

Функція `strtok ()` використовується для перетворювання рядка на послідовність лексем. *Лексема* - це послідовність символів, відокремлених символами-розділювачами (зазвичай пробілами чи знаками пунктуації). Наприклад, у рядку тексту кожне слово може розглядатися як лексема, а пробіли, що відокремлюють слова одне від одного, можна розглядати як розділювачі. Для того щоб розбити рядки на лексеми, треба організувати кілька викликів функції `strtok ()` (за умови, що рядок вміщує понад одну лексему).

The ASCII code

American Standard Code for Information Interchange

ASCII control characters

DEC	HEX	Simbolo ASCII	
00	00h	NULL	(carácter nulo)
01	01h	SOH	(inicio encabezado)
02	02h	STX	(inicio texto)
03	03h	ETX	(fin de texto)
04	04h	EOT	(fin transmisión)
05	05h	ENQ	(enquiry)
06	06h	ACK	(acknowledgement)
07	07h	BEL	(timbre)
08	08h	BS	(retroceso)
09	09h	HT	(tab horizontal)
10	0Ah	LF	(salto de linea)
11	0Bh	VT	(tab vertical)
12	0Ch	FF	(form feed)
13	0Dh	CR	(retorno de carro)
14	0Eh	SO	(shift Out)
15	0Fh	SI	(shift In)
16	10h	DLE	(data link escape)
17	11h	DC1	(device control 1)
18	12h	DC2	(device control 2)
19	13h	DC3	(device control 3)
20	14h	DC4	(device control 4)
21	15h	NAK	(negative acknowle.)
22	16h	SYN	(synchronous idle)
23	17h	ETB	(end of trans. block)
24	18h	CAN	(cancel)
25	19h	EM	(end of medium)
26	1Ah	SUB	(substitute)
27	1Bh	ESC	(escape)
28	1Ch	FS	(file separator)
29	1Dh	GS	(group separator)
30	1Eh	RS	(record separator)
31	1Fh	US	(unit separator)
127	20h	DEL	(delete)

ASCII printable characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
32	20h	espacio	64	40h	@	96	60h	`
33	21h	!	65	41h	A	97	61h	a
34	22h	"	66	42h	B	98	62h	b
35	23h	#	67	43h	C	99	63h	c
36	24h	$	68	44h	D	100	64h	d
37	25h	%	69	45h	E	101	65h	e
38	26h	&	70	46h	F	102	66h	f
39	27h	'	71	47h	G	103	67h	g
40	28h	(	72	48h	H	104	68h	h
41	29h	)	73	49h	I	105	69h	i
42	2Ah	*	74	4Ah	J	106	6Ah	j
43	2Bh	+	75	4Bh	K	107	6Bh	k
44	2Ch	,	76	4Ch	L	108	6Ch	l
45	2Dh	-	77	4Dh	M	109	6Dh	m
46	2Eh	.	78	4Eh	N	110	6Eh	n
47	2Fh	/	79	4Fh	O	111	6Fh	o
48	30h	0	80	50h	P	112	70h	p
49	31h	1	81	51h	Q	113	71h	q
50	32h	2	82	52h	R	114	72h	r
51	33h	3	83	53h	S	115	73h	s
52	34h	4	84	54h	T	116	74h	t
53	35h	5	85	55h	U	117	75h	u
54	36h	6	86	56h	V	118	76h	v
55	37h	7	87	57h	W	119	77h	w
56	38h	8	88	58h	X	120	78h	x
57	39h	9	89	59h	Y	121	79h	y
58	3Ah	:	90	5Ah	Z	122	7Ah	z
59	3Bh	;	91	5Bh	[	123	7Bh	{
60	3Ch	<	92	5Ch	\	124	7Ch	|
61	3Dh	=	93	5Dh	]	125	7Dh	}
62	3Eh	>	94	5Eh	^	126	7Eh	~
63	3Fh	?	95	5Fh	_			

theASCIIcode.com.ar

Extended ASCII characters

DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo	DEC	HEX	Simbolo
128	80h	Ç	160	A0h	á	192	C0h	└	224	E0h	Ó
129	81h	ü	161	A1h	í	193	C1h	┴	225	E1h	ß
130	82h	é	162	A2h	ó	194	C2h	┬	226	E2h	Ô
131	83h	â	163	A3h	ú	195	C3h	├	227	E3h	Ò
132	84h	ä	164	A4h	ñ	196	C4h	─	228	E4h	õ
133	85h	à	165	A5h	Ñ	197	C5h	┼	229	E5h	Õ
134	86h	å	166	A6h	ª	198	C6h	ã	230	E6h	µ
135	87h	ç	167	A7h	º	199	C7h	Ã	231	E7h	þ
136	88h	ê	168	A8h	¿	200	C8h	╚	232	E8h	Þ
137	89h	ë	169	A9h	®	201	C9h	╔	233	E9h	Ú
138	8Ah	è	170	AAh	¬	202	CAh	╩	234	EAh	Û
139	8Bh	ï	171	ABh	½	203	CBh	╦	235	EBh	Ù
140	8Ch	î	172	ACh	¼	204	CCh	╠	236	ECh	ý
141	8Dh	ì	173	ADh	¡	205	CDh	═	237	EDh	Ý
142	8Eh	Ä	174	AEh	«	206	CEh	╬	238	EEh	¯
143	8Fh	Å	175	AFh	»	207	CFh	¤	239	EFh	´
144	90h	É	176	B0h	░	208	D0h	ð	240	F0h	
145	91h	æ	177	B1h	▒	209	D1h	Ð	241	F1h	±
146	92h	Æ	178	B2h	▓	210	D2h	Ê	242	F2h	‗
147	93h	ô	179	B3h	│	211	D3h	Ë	243	F3h	¾
148	94h	ò	180	B4h	┤	212	D4h	È	244	F4h	¶
149	95h	ò	181	B5h	Á	213	D5h	ı	245	F5h	§
150	96h	û	182	B6h	Â	214	D6h	Í	246	F6h	÷
151	97h	ù	183	B7h	À	215	D7h	Î	247	F7h	¸
152	98h	ÿ	184	B8h	©	216	D8h	Ï	248	F8h	°
153	99h	Ö	185	B9h	╣	217	D9h	┘	249	F9h	¨
154	9Ah	Ü	186	BAh	║	218	DAh	┌	250	FAh	·
155	9Bh	ø	187	BBh	╗	219	DBh	█	251	FBh	¹
156	9Ch	£	188	BCh	╝	220	DCh	▄	252	FCh	³
157	9Dh	Ø	189	BDh	¢	221	DDh	¦	253	FDh	²
158	9Eh	×	190	BEh	¥	222	DEh	Ì	254	FEh	■
159	9Fh	ƒ	191	BFh	┐	223	DFh	▀	255	FFh	

Структури (struct)

У попередніх розділах для зберігання даних розглянуто прості типи даних: числа, символи, рядки тощо. Але часто на практиці перед програмістом постає завдання запрограмувати той чи інший об`єкт, який характеризується водночас кількома параметрами, приміром, точка на площині задається парою дійсних чисел (*x*, *y*), а дані про людину можна задавати кількома параметрами: прізвище, ім`я, по-батькові – рядки, рік народження – число тощо. Для поєднання кількох параметрів в одному об`єкті в C++ існує спеціальний тип даних, який має назву структура. На відміну від масивів, які також є структурованим типом даних і містять елементи одного типу, структури дозволяють поєднувати дані різних типів.

Структура – це складений тип даних, змінні (поля) якого можуть містити різноманітну й різнотипну інформацію.

Опис структури має синтаксис

```
struct <ім`я_типу_структури>
      {
            <тип1> <поле1>;
            <тип2> <поле2>;
            . . .
                  <типN> <полеN>;
      } <список_змінних>;
```

Ім`я типу структури задається програмістом виходячи зі змісту даних, які зберігатимуться у структурі. Список змінних наприкінці оголошення структури може бути відсутнім, у цьому разі після фігурної дужки має бути крапка з комою.

Наведемо приклад оголошення структури з ім`ям sportsman, яка поєднає в собі інформацію про змагання спортсменів: прізвище і вік спортсмена, країна, кількість його очок, утворюючи відповідні поля:

```
struct sportsman
{
      char prizv[15]; char kraina[10]; // Прізвище та країна,
      int vik; float ochki; // вік та кількість очок.
      };
```

Як наслідок цього оголошення створено новий тип sportsman. Тепер у програмі цей тип може використовуватись нарівні зі стандартними типами для оголошення змінних. Оголошені змінні типу sportsman матимуть чотири поля: два рядки (prizv і kraina), ціле (vik) і дійсне (ochki) числа.

```
      sportsman Sp; /* Змінна Sp типу sportsman може зберігати інформацію про
одного спортсмена. */
      sportsman Mas[15]; /* Масив Mas типу sportsman може містити інформацію
про 15 спортсменів. */
```

При оголошуванні змінної типу структури пам`ять під усі поля структури виділяється послідовно для кожного поля. У наведеному прикладі структури sportsman

під змінну Sp послідовно буде виділено 15, 10, 4, 4 байти – усього 33 байти. Поля змінної Sp у пам`яті буде розміщено у такому порядку:

Слід зазначити, що сумарну пам`ять, яка виділяється під структуру, може бути збільшено компілятором на вимогу процесора для так званого вирівнювання адрес. Реальний обсяг пам`яті, яку займає структура, можна визначити за допомогою sizeof(). При оголошуванні структур їхні поля можна ініціалізовувати початковими значеннями. Наприклад, оголошення змінної Sp типу sporsman й ініціалізація її даними про 20-річного спортсмена з України за прізвищем Бойко, який набрав 75.3 очок, матиме вигляд:

```
sporsman Sp = { "Бойко", "Україна", 20, 75.3 };
```

Оголошення типу “точка на площині” POINT з полями-координатами – x та y і змінної spot цього типу – точки з координатами (20, 40) може бути таким:

```
struct POINT
      {
            int x, y;
      } spot = { 20, 40 }; // Точка має координати x = 20, y = 40
```

При ініціалізації масивів структури кожен елемент масиву слід записувати у фігурних дужках, наприклад при створенні двовимірного масиву compl розміром 2*x*3 типу структури complex з полями real та im ініціалізація початкових значень елементів може мати вигляд:

```
struct complex
      {
            float real, im;
      } compl[2][3] = { { { 1.4 },{ 5 },{ -3.1 } },{ { 2 },{ -5.2 },{ 0 } } };
```

Доступ до полів структури здійснюється за допомогою операції “крапка”:
<структурна змінна>.<ім'я поля>

Наприклад, для розглянутої змінної Sp типу sporsman надання значень її полям може мати вигляд:

```
      strcpy(Sp.prizv, "Бойко");
      strcpy(Sp.kraina, "Україна");
      Sp.vik = 20;
      Sp.ochki = 75.3;
```

Можна створювати тип структури за допомогою typedef, наприклад:

```
typedef struct
      {
            AnsiString priz, gruppa;
            int Bal_math, Bal_inform;
      } student; // Оголошення типу “студент”
```

```
student W; // і змінної W цього типу.
```

Тут означено тип структури за ім`ям student для зберігання даних про успішність студентів з полями: прізвище, група, екзаменаційні оцінки з математики та інформатики й оголошено змінну W цього типу.

Структури можуть розміщуватись у динамічній пам`яті за допомогою оператора new і оголошуватися як покажчики. У такому разі доступ до полів структури здійснюється за допомогою операції вибору – “стрілка” (->, на клавіатурі послідовно натискаються клавіші “мінус” і “більше”):

```
sportsman *p = new sportsman; // Виділення пам“яті,
p->ochki = 95.3; // присвоєння очок полю ochki
(*p).vik = 23; // і віку полю vik.
```

Зверніть увагу на те, що якщо p – покажчик на структуру, то (*p) – сама структура і доступ до її полів здійснюється за допомогою крапки.

Якщо змінні типу “структура” оголошуються у програмі лише один раз, то, зазвичай, їх записують наприкінці оголошення структури і при цьому ім`я структури можна явно не зазначати, наприклад:

```
struct
{
      char prizv[15];
      char kraina[10];
      int vik; float ochki;
}
Sp;
```

Тут оголошено змінну Sp як структуру з чотирма полями без створення типу “спортсмен”.

Поля структури можуть бути якого завгодно типу, у тому числі й масивом, покажчиком чи структурою, окрім типу тієї ж самої структури (але можуть бути покажчиком на неї). За приклад створимо дві структури для опрацювання інформації про співробітників у відділі кадрів: прізвище, ім`я, по-батькові співробітника, домашня адреса, домашній телефон, дата народження, дата вступу на роботу, зарплатня.

```
struct date
{
      int day, month, year; // День, місяць та рік
};
struct person
{
      char prizv[20], adresa[150]; // Прізвище, адреса,
      char phone[10]; date birth_date; // телефон, дата народження,
      date work_date; // дата вступу на роботу
      float salary; // та зарплатня.
};
```

У цьому прикладі оголошується новий тип – структура date для зберігання дати (день, місяць, рік). Окрім того, оголошується тип – структура person, яка використовує тип date для полів birth_date і work_date. Посилан-ня на поля вкладеної структури формується з імені структурної змінної, імені структурного поля й імені поля вкладеної структури. Те, що поля структур самі можуть бути структурами, надає можливість цілісно описувати доволі складні об`єкти реального світу. Оскільки структури, на відміну від масивів, зберігають дані різних типів, їх відносять до неоднорідних типів даних.

Оголосимо змінну створеного типу person:

```
person P;
```

Присвоювання прізвища у змінну P:

```
strcpy(P.prizv, "Шкуропат");
```

Присвоювання дати народження 5 березня 1987 року у змінну P:

```
P.birth_date.day=5;
P.birth_date.month=3;
P.birth_date.year=1987;
```

Розмістимо змінну man типу person у динамічній пам‟яті:

```
person *man = new person;
```

І присвоїмо ті ж самі значення:

```
strcpy(man->prizv, "Шкуропат");
(man->birth_date).day=5;
(man->birth_date).month=3;
(man->birth_date).year=1987;
```

Оголосимо масив з даними про 100 осіб:

```
person P[100];
```

Попередні присвоювання для четвертої людини:

```
strcpy(P[3].prizv, "Шкуропат");
P[3].birth_date.day=5;
P[3].birth_date.month=3;
P[3].birth_date.year=1987;
```

Уведемо значення всіх полів для всіх співробітників до масиву:

```
for (i = 0; i<100; i++)
     {
          cout << "Введіть призвіще: "
               gets(P[i].prizv);
          cout << "Введіть адресу: "
               gets(P[i].adresa);
          cout << "Введіть телефон: "
               gets(P[i].phone);
          cout << "Введіть дату народження: " << endl;
          cout << "День: "; cin >> P[i].birth_date.day;
          cout << "Місяць: "; cin >> P[i].birth_date.month;
          cout << "Рік: " cin >> P[i].birth_date.year;
          cout << "Введіть дату вступу до роботи: "endl;
          cout << "День: "; cin >> P[i].work_date.day;
          cout << "Місяць: "; cin >> P[i].work_date.month;
          cout << "Рік: " cin >> P[i].work_date.year;
     }
```

Структурні змінні одного типу можна переприсвоювати одну одній, при цьому виконується присвоювання усіх полів, наприклад:

```
person man=P[0];
```

Як вже було зазначено вище, поля структури можуть бути якого завгодно типу, у тому числі масивами. Наприклад, оголошення структури “погода за місяць” може мати вигляд

```
struct pogoda
	{
		int month; // Номер місяця,
		int temp[31];// масив температур за 31 день.
	};
	pogoda M; // Оголошення змінної M типу pogoda (погода за один місяць).
	pogoda G[12]; // і масиву G типу pogoda (погода за рік).
```

Визначимо значення температури повітря за 14 липня:

```
M.month = 7;
M.temp[13] = 27;
```

Присвоїмо випадкові значення температури за весь лютий:

```
M.month = 2;
for(int i=0; i<28; i++) M.temp[i] = random(5)-10;
```

Виведення температури повітря восени (за 3 місяці):

```
for (int i = 9; i <= 11; i++)
	{
		G[i - 1].month = i;
		for (int j = 0; j<31; j++)
			if (j == 30 && (i == 9 || i == 11))
				continue; // Пропустити 31.09 та 31.11
			else
				cout << G[i - 1].temp[j] << " ";
		cout << endl;
	}
```

У C++ структура при описуванні, окрім звичних полів, може містити елементи-функції. Наприклад, можна дописати до структури pogoda функцію month_name, яка визначатиме рядок – назву місяця за номером.

```
struct pogoda
	{
		int month, temp[31]; // Номер місяця і масив температур за 31
день.
		char *month_name()
		{
			switch (month)
			{
			case 1: return "Січень";
			case 2: return "Лютий";
			case 3: return "Березень";
			case 4: return "Квітень";
			case 5: return "Травень";
			case 6: return "Червень";
```

```
                case 7: return "Липень";
                case 8: return "Серпень";
                case 9: return "Вересень";
                case 10: return "Жовтень";
                case 11: return "Листопад";
                case 12: return "Грудень";
                default: return "Помилковий номер";
                }
        }
};
```

У програмі цю функцію можна викликати в такий спосіб:

```
pogoda M;
        . . .
                cin >> M.month;
        cout << M.month_name();
```

Після виконання цих операторів на екран буде виведено рядок, утворений за допомогою функції month_name().

Наведемо ще один приклад елемента-функції. Зорганізуємо структуру student з функцією Show(), яка будуватиме рядок типу AnsiString, поєдную-чи усі поля цієї структури. Надалі цей рядок можна буде використовувати для виведення записів на екран:

```
struct student
        {
                String prizv, grupa;
                int Bal_math, Bal_inform;
                String Show()
                {
                        return prizv + " " + grupa + " " + Bal_math+ " " +
                                Bal_inform;
                }
};
```

До функцій структури передаються як звичайні змінні.

Об’єднання (union)

*Об`єднання* – формат даних, який може містити різні типи даних, але лише один тип водночас. Можна сказати, що об`єднання є окремим випадком структури, всі поля якої розташовуються за однією й тією самою адресою. Формат опису об`єднання є такий самий як і у структури, лише замість ключового слова struct використовується слово union. Але, тоді як структура може містити, скажімо, елементи типу і int, і short, і double, об`єднання може містити чи то int, чи short, чи double:

```
union prim
{
        int x;
        short y;
        double z;
```

```
};
```

Обсяг пам`яті, яку займає об`єднання дорівнює найбільшому з розмірів його полів. Об`єднання застосовують для економії пам`яті у тих випадках, коли відомо, що понад одного поля водночас не потрібно. Часто об`єднання використовують як поле структури. При цьому до структури зручно включити додаткове поле, яке визначає, який саме елемент об`єднання використовується даного моменту:

```
struct
{
      int type;
      union id
      {
            long id_num; char id_ch[20];
      };
} price;
. . .
      if (price.type == 1) cin >> price.id.id_num;
      else cin >> price.id.id_ch;
```

Тут структура price містить ціле поле type та поле id, яке може набувати лише одне зі значень чи то цілого числа id_num, чи то рядка id_ch залежно від значення поля type. У наведеному прикладі ім`я об`єднання можна не зазначати, а звертатися безпосередньо до його полів. У цьому разі об`єднання є анонімним, його елементи стають змінними, розташованими за однією адресою.

```
struct
      {
            int type;
            union
            {
                  long id_num;
                  char id_ch[20];
            };
      } price;
      . . .
            if (price.type == 1) cin >> price.id_num;
            else cin >> price.id_ch;
```

Перерахування (enum)

При написанні програм часто виникає потреба у визначенні наперед відомої кількості іменованих констант, які мають мати різні значення (при цьому конкретні значення можуть бути неважливими). Для цього зручно користуватися типом перерахування enum, всі можливі значення якого задаються списком цілочисельних констант:

enum [<*ім`я_типу*>] {<*список_констант*>};

Ім`я типу задається в тому разі, якщо у програмі потрібно визначати змінні цього типу. Змінним перелічувального типу можна присвоювати кожне значення зі

списку констант, зазначених при оголошуванні типу. Імена перелічуваних констант мають бути унікальними. Перераховні константи подаються й опрацьовуються як цілі числа і можуть ініціалізуватися у звичайний спосіб. Окрім того вони можуть мати однакові значення. За відсутності ініціалізації перша константа обнулюється, а кожній наступній присвоюється значення на одиницю більше, аніж попередній.

```
enum week { sat = 0, sun = 0, mon, tue, wed, thu, fri } rob_den;
```

Тут описано перерахування week з відповідною множиною значень і оголошено змінну rob_den типу week. Перераховні константи sat та sun мають значення 0, mon – значення 1, tue – 2, wed – 3, thu – 4, fri – 5.

До перераховних змінних можна застосовувати арифметичні операції й операції відношення, наприклад:

```
enum days { sun, mon, tue, wed, thu, fri, sat };
      days day1 = mon, day2 = thu;
      int diff = day2 - day1;
      cout << "Різниця у днях: " << diff << endl;
      if (day1 < day2)
            cout << "day1 настане раніш, аніж day2 \n";
```

Арифметичні операції над змінними перелічуваного типів зводяться до операцій над цілими числами. Проте, не зважаючи на те, що компіляторові відомо про цілочисельну форму подання перераховних значень, варто використовувати цей факт надто обережно. Якщо спробувати виконати присвоювання

```
day1 = 5;
```

компілятор видасть повідомлення (хоча компіляція відбудеться без помилок). Рекомендовано без нагальної потреби не використовувати цілочисельну інтерпретацію перераховних значень.

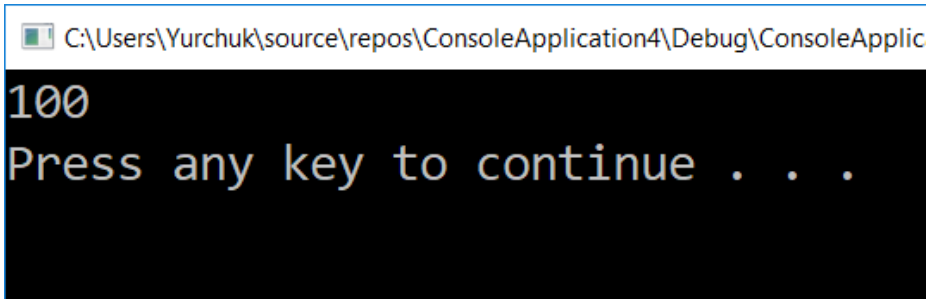
Значення перелічуваним константам можна встановлювати явно, ініціалізуючи їх при оголошенні, причому значення мають бути лише цілими числами, наприклад:

```
enum { first, second = 100, third };
```

У цьому разі first за замовчуванням дорівнює 0, а наступні неініціалізовані константи перерахування перевищують своїх попередників на 1. Приміром, third матиме значення 101.

Істотним недоліком типів перерахування є те, що вони не розпізнаються засобами введення-виведення C++. При виведенні такої змінної виводиться не її формальне значення, а її внутрішнє подання, тобто ціле число.

```
      enum xx { first, second = 100, third };
      xx a;
      a = second;
      cout << a;
```


C:\Users\Yurchuk\source\repos\ConsoleApplication4\Debug\ConsoleApplic
100
Press any key to continue . . .

Lec_ Файли.

Не дивлячись на те що система введення-виведення мови С++ у цілому являє собою єдиний механізм, система файлового введення-виведення має свої особливості. Частково це пояснюється тим, що на практиці найчастіше використовуються файли на жорсткому диску, можливості яких значно відрізняються від усіх інших обладнань. Однак слід мати на увазі, що файлове введення- виведення є лише частиною загальної системи введення- виведення, в лекції розглядатимуться потоки, які можуть бути пов'язані з іншою апаратурою.

Заголовок <fstream> і класи файлів

Для реалізації файлового введення- виведення в програму слід включити заголовок <*fstream*>. У ньому визначені деякі класи, зокрема , *ifstream*, *ofstream* і *fstream*. Ці класи є похідними від класів *istream*, *ostream* і *iostream* відповідно. Слід пам'ятати, що класи *istream*, *ostream* і *iostream*, у свою чергу, є похідними від класу *ios*. Файлова система використовує також клас *filebuf*, що надає низькорівневі засоби керування файловим потоком. Звичайно клас *filebuf* безпосередньо не застосовується, однак він є складовою частиною інших класів.

Відкриття й закриття файлу

У мові С++ відкриття файлу означає його зв'язування з потоком. Отже, спочатку необхідно одержати потік. Існують три види потоків: уведення, виведення й вводу-виведення. Для того щоб створити потік уведення, необхідно оголосити потік, що представляє собою об'єкт класу ifstream. Для генерації потоку виведення необхідно оголосити потік, що представляє собою об'єкт класу ofstream. Потоки, що здійснюють уведення й виведення, оголошуються як об'єкти класу fstream. Наприклад, у наступному фрагменті створюється потік уведення, потік виведення й потік вводу-виведення.

```
ifstream in;      //   Уведення
      ofstream out;   //   Виведення
      fstream  io;        //   Уведення й  виведення
```

Створений потік можна зв'язати з файлом за допомогою функції open(). Ця функція є членом кожного із трьох потокових класів. Її прототипи в кожному класі показані нижче.

```
ifstream::open(const  char* filename, ios::openmode mode = ios::in);
ifstream::open(const  char* filename, ios::openmode mode = ios::out | ios::trunc);
ifstream::open(const  char* filename, ios::openmode mode = ios::in | ios::out);
```

Тут параметр filename задає ім'я файлу. Він може містити шлях до цього файлу. Значення параметра mode визначає спосіб відкриття файлу. Цей параметр може ухвалювати наступні значення, описані у функції openmode.

```
ios::app
ios::ate
ios::binary
ios::in
ios::out
ios::trunc
```

Дані значення можна комбінувати за допомогою логічної операції "АБО"

Якщо задається значення **ios::app**, усі результати дописуються в кінець файлу, відкритого для виведення.

Якщо зазначене значення **ios::ate**, при відкритті виконується пошук кінця файлу. Незважаючи на це, запис проводиться в будь-яке місце файлу.

Якщо задається значення **ios: :in**, файл відкривається для введення, а якщо **ios: :out** - для виведення.

Значення **ios::binary** дозволяє відкрити файл у бінарному режимі. За замовчуванням усі файли відкриваються в текстовому режимі. У цьому випадку проводиться перетворення деяких символів, наприклад, эскейп- послідовність "повернення каретки" і "прогін паперу" перетвориться в символ переходу на новий рядок. Однак, якщо файл відкритий у бінарному режимі, перетворення символів не проводиться. Слід пам'ятати, що будь-який файл можна відкрити як у текстовому, так і в бінарному режимі. Єдина відмінність між ними полягає в тому, проводиться перетворення символів чи ні.

Значення **ios: :trunc** сигналізує, що попередній уміст існуючого файлу з тим же іменем буде знищено, а довжина файлу зменшена до нуля. При відкритті потоку виведення за допомогою класу ofstream уміст будь-якого існуючого файлу із зазначеним іменем стирається.

У наступному фрагменті відкривається текстовий файл для виведення.

```
ofstream out;
out.open("test", ios::out);
```

Однак функція **open()** рідко застосовується для відкриття файлів, тому що для кожного типу потоку параметр mode має значення, задані за замовчуванням. Прототипи функції **open** демонструють, що значення параметра **mode**, задане за замовчуванням, у класі **ifstream** рівно **ios:: in**, у класі **ofstream - ios: :out | ios: :trunc**, а в класі **fstream - ios: :in | ios: :out**. Із цієї причини попередній виклик функції open звичайно записують так:

```
out.open("test");   //   Текстовий файл для  виведення
```

Залежно від компілятора параметр mode для функції f stream: :open() може мати значення за замовчуванням, відмінне від in \ out. Отже, іноді його необхідно задавати явно.

Якщо потік застосовується в логічних виразах, у випадку відмови функції **open()** йому привласнюється значення **false**. Таким чином, перед використанням файлу слід переконатися, що він успішно відкритий. Для цього можна скористатися наступними операторами.

```
if(!mystream) {
      cout << "Неможливо  відкрити  файл.\n"; //  Обробка помилки }
```

Однак частіше всього функцію **open()** не застосовують, оскільки класи **ifstream**, **ofstream** і **fstream** містять конструктори, що автоматично відкривають файл. Параметри цих конструкторів ухвалюють ті ж значення за замовчуванням, що й функція **open()**. Із цієї причини файли звичайно відкривають у такий спосіб.

```
ifstream mystream("myfile");   //   Відкриття  файлу для  введення
```

Якщо з якої-небудь причини файл відкрити не вдалося, пов'язаному з ним потоку привласнюється значення false. Отже, якщо якийсь конструктор викликає функцію **open()**, слід переконатися, що файл дійсно відкритий, перевіривши значення потоку.
Щоб перевірити, чи відкритий файл, можна викликати функцію **is_open()**, що є членом класу **fstream**, **ifstream** і **ofstream** Вона має наступний прототип.

```
bool   is_open();
```

Якщо потік пов'язаний з відкритим файлом, ця функція повертає значення **true**, а якщо ні, то вона повертає значення **false**. Наприклад нище фрагмент, перевіряє, чи відкритий файл, пов'язаний з потоком **mystream**.

```
if(!mystream.is_open()) {
            cout << "Файл не відкритий.\n";
      }
```

Щоб закрити файл, слід викликати функцію **close()**. Наприклад, щоб закрити файл, пов'язаний з потоком **mystream**, можна застосувати наступний оператор.

```
mystream.close();
```

Функція **close ()** не має параметрів і не повертає ніяких значень.

Читання й запис текстових файлів

Читання й запис текстових файлів здійснюються дуже легко. Для цього досить застосувати оператори "<<" і ">>", як це звичайно робиться для консольного введення- виведення, тільки замість потоків cin і cout необхідно підставити потік, звязаний з файлом. Наприклад наступна програма створює короткий файл, що містить назву предмета і його вартість.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
      ofstream out("INVNTRY"); // Текстовий файл для виведення.
      if(!out) {
            cout << "Неможливо відкрити файл INVENTORY.\n";
```

```
            return 1;
      }
      out << "Радіоприймачі " << 39.95 << endl;
      out << "Тостери " << 19.95 << endl;
      out << "Міксери " << 24.80 << endl;
      out.close();
      return 0;
}
```

Програма, наведена нижче, зчитує файл, створений попередньої програмою, і виводить його вміст на екран.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
      ifstream in("INVNTRY"); // Уведення
      if(in) {
            cout << "Неможливо відкрити файл INVENTORY.\n";
            return 1;
      }
      char item[20]; float cost;
      in >> item >> cost;
      cout << item << " " << cost << "\n";
      in >> item >> cost;
      cout << item << " " << cost << "\n";
      in >> item >> cost;
      cout << item << " " << cost << "\n";
      in.close(); return 0;
}
```

Наступна програма зчитує рядки, уведені із клавіатури, і записує їх на диск. Програма зупиняється, якщо користувач увів знак оклику.

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

      ofstream out("my.txt"); // Текстовий файл для виведення.
      if(!out) {
            cout << "Неможливо відкрити файл для виведення.\n";
            return 1;
      }
      char str[80];
      cout << "Запис рядків на жорсткий диск. Для припинення роботи введіть знак оклику.\n";
      do {
            cout << ": ";
            cin << str;
            out << str << endl;
      } while(*str != '!');
      out.close();
      return  0;
}
```

Зчитуючи файли за допомогою оператора ">>", майте на увазі, що деякі символи при введенні трансформуються. Наприклад, роздільники ігноруються. Щоб запобігти перетворенню символів при читанні, файл слід відкрити в бінарному режимі й застосувати функції, описані в наступному розділі.
Якщо при введенні досягається кінець файлу, потоку, пов'язаному з файлом, привласнюється значення false. (Ця ситуація ілюструється в наступному розділі.)

Бесформатне і бінарне введення- виведення

Отже, читання й запис форматних текстових файлів не викликає ніяких труднощів, хоча це не найефективніший спосіб роботи з файлами. Крім того, іноді виникає необхідність зберігати бесформатні дані, а не текст. Розглянемо функції, призначені для роботи з такими даними.

Виконуючи з файлом бінарні операції, слід переконатися, що він відкритий у режимі ios::binary. Бесформатні дані можуть зберігатися й у текстовому файлі, але в цьому випадку при читанні деякі символи будуть перетворені. Бінарні файли застосовуються саме для того, щоб цього уникнути.

Порівняння символів і байтів.

При вивченні бесформатного введення-виведення необхідно враховувати наступне. Багато років введення-виведення у мовах C и C++ було байтовим (byte oriented). Це відбувалося тому, що символ (char) є еквівалентом байта, і потоки введення-виведення були символьними. Однак з появою розширених символів (wchar_t) і пов'язаних з ними потоків систему введення-виведення мови C++ не можна назвати байтовою. Тепер її слід називати символьною (character oriented). Зрозуміло, потоки звичайних символів (char) залишаються байтовими, особливо при обробці нетекстових даних. Однак еквівалентність понять "символ" і "байт" більше не гарантується.

Функції put() і get()

Один зі способів зчитування й запису бесформатных файлів заснованих на застосуванні функцій **put()** і **get()**. Ці функції оперують символами. Точніше кажучи, функція **get()** зчитує символ, а функція **put()** - записує його. Зрозуміло, якщо файл відкритий у бінарному режимі, то при зчитуванні символу (а не розширеного символу), ці функції зчитують і записують байти.

Функція **get()** має кілька форм, однак найчастіше використовується її наступна версія. "Клас:ostream: функція-член:put”

```
istream& get(char& ca)
ostream& put(char ch)
```

Функція **get()** зчитує окремий символ з потоку й записує його в змінну **ch**. Крім того, функція **get()** повертає посилання на потік. Функція **put()** записує змінну **ch** у потік і повертає посилання на потік.

Наступна програма відображає на екрані вміст будь-якого файлу (як текстового, так і бінарного). Для зчитування данных вона використовує функцію **get()**.

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

	char ch;

	ifstream in("myfile.txt", ios::in | ios::binary);
	if(!in) {
		cout << "Неможливо відкрити файл.";
		return 1;
	}
	while(in) {  // Якщо досягнуться кінець файлу,
	// значення об'єкта in рівно false.
		in.get(ch);
		if(in) cout << ch;
	}
	return 0;
}
```

По досягненню кінця файлу потік, пов'язаний із цим файлом, набуває значення **false**. Отже, рано або пізно об'єкт **in** прийме значення **false**, і виконання циклу **while** припиниться.

Наведений вище цикл можна записати коротше.

```
while(in.get(ch)) cout << ch;
```

Цей фрагмент є правильним, оскільки функція **get()** повертає посилання на потік **in**, який прийме значення **false** при виявленні кінця файлу.

У наступній програмі для запису в файл CHARS усіх символів від 0 до 255 застосовується функція **put()**. Як відомо, символи ASCII займають лише половину з можливих значень типу **char**. Інші символи, як правило, називаються розширеними (**extended character set**). До них відносяться букви національних алфавітів і математичні знаки. (Деякі системи не підтримують розширені символи.)

```
#include <iostream>
#include <fstream>
```

```
using namespace std;
int main() {
	int i;
	ofstream out("chars", ios::out | ios::binary);
		if(!out) {
		cout << "неможливо відкрити файл.\n";
		return 1;
	}
	// Записати символи на диск.
	for (i = 0; i < 256; i++) out.put((char)i);
	out.close(); return 0;
}
```

За допомогою цієї програми можна перевірити, чи підтримує ваш комп'ютер розширені символи.

Функції read() і write()

Блоки бінарних даних можна зчитувати за допомогою функцій **read()** і **write()**. Їхні прототипи виглядають у такий спосіб.

```
	istream& read(char* buf, streamsize num);
	istream& write(const char* buf, streamsize num);
```

Функція **read** зчитує **num** символів з потоку й записує їх у буфер, на який посилається покажчик **buf**. Функція **write** записує **num** символів у потік, зчитуючи їх з буфера, на який посилається покажчик **buf**. Тип **streamsize** визначений у бібліотеці як різновид типу **int**. Він дозволяє зберігати максимальна кількість символів, які можуть перетворюватися при виконанні операцій введення-виведення.
Наступна програма записує структуру на диск, а потім зчитує назад.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct status {
	char name[80];
	double balance;
	unsigned long account_num;
};
int main() {
	struct status ace, acc;
	strcpy(ace.name, "Ральф Трантор");
	ace.balance = 1123.23;
	ace.account_num = 34235678;
	// Записуємо дані
	ofstream outbal("balance", ios::out | ios::binary);
	if(!outbal) {
		cout << "Неможливо відкрити файл.\n";
		return 1;
	}
	outbal.write((char*)&ace, sizeof(struct status));
	outbal.close();
	// Зчитуємо дані знову
	ifstream inbal("balance", ios::in | ios::binary);
	if(!inbal) {
		cout << "Неможливо відкрити файл.\n";
		return 1;
	}
	inbal.read((char*)&acc, sizeof(struct status));
	cout << acc.name << endl;
	cout << "Рахунок # " << ace.account_num;
	cout.precision(2);
	cout.setf(ios::fixed);
	cout << endl << "Баланс: $" << ace.balance;
	inbal.close(); return 0;
}
```

Як бачимо, для зчитування або запису цілої структури досить одного виклику функції **read()** або **write()**. Окреме поле структури неможливо вважати або записати окремо. Крім того, цей приклад показує, що буфером може служити об'єкт будь-якого типу.

Якщо буфер не є символьним масивом, при виклику функцій read() й write() необхідно виконувати переведення типів. Оскільки в мові C++ виконується строга перевірка типів, покажчик одного типу не може автоматично перетворюватися в покажчик іншого типу.

Якщо кінець файлу виявиться перш, ніж будуть злічені **num** символів, функція **read()** просто припинить роботу, а в буфері буде записана максимально можлива кількість символів. Кількість зчитаних символів можна визначити за допомогою функції **gcount()**. Її прототип виглядає в такий спосіб.

```
streamsize  gcount();
```

Дана функція повертає кількість символів, зчитаних при виконанні останньої операції бінарного введення. Розглянемо ще один приклад використання функцій **read()**, **write()** і **gcount()**.

```
#include <iostream>
#include <fstream>
#include <cstring>
using namespace std;
struct status {
	char name[80];
	double balance;
	unsigned long account_num;
};
int main() {
	double fnum[4] = { 99.75, -34.4, 1776.0, 200.1 };
	int i;
	ofstream out("numbers", ios::out | ios::bwnary);
	if(!out) {
		cout << "Неможливо відкрити файл.";
		return 1;
	}
	out.write((char*)&fnum, sizeof fnum);
	out.close();
	for (i = 0; i < 4; i++) // Очищення масиву.
		fnum[i] = 0.0;
	ifstream in("numbers", ios::in | ios::binary);
	in.read((char*)&fnum, sizeof fnum);
	// Визначаємо кількість лічених символів.
	cout << "Зчитано " << in.gcount() <<" байтів.\n";
	for (i = 0; i < 4; i++) // Показати значення, зчитані з файлу.
		cout << fnum[i] << " ";
			in.close();
	return 0;
}
```

Ця програма записує масив десяткових чисел на жорсткий диск, а потім зчитує їх назад. Після виклику функції **read()** застосовується функція **gcount()**, що повертає кількість лічених символів.

Додаткові функції get()

Крім попередніх форм функція **get()** має ще кілька перевантажених різновидів. Розглянемо три основні прототипи.

```
istream& get(char* buf, streamsize num);
istream& get(char* buf, streamsize num, char delim);
int get();
```

Перша форма функції get зчитує символи з масиву, на який посилається покажчик buf поки не будуть зчитані num-1 символів, виявлений символ переходу на наступний рядок або досягнеться кінець файлу. Функція **get()** записує нульовий символ у кінець масиву, на який посилається покажчик **buf** Символ переходу на новий рядок не зчитується. Він залишається в потоці, поки не буде виконана наступна операція введення.

Друга форма функції **get** зчитує символи в масив, на який посилається покажчик **buf**, поки не будуть зчитані num-1 символів, виявлений символ delim або досягнутий кінець файлу. Функція **get()** записує нульовий символ у кінець масиву, на який посилається покажчик **buf**. Символ **delim** з потоку не зчитується. Він залишається в потоці, поки не буде виконана наступна операція введення.

Третя форма функції **get** витягає з потоку наступний символ. Якщо виявлений кінець файлу, вона

повертає константу **EOF**. Ця форма функції **get()** схожа на функцію getc () у мові С.

Функція getline()

Крім функції **get()** у мові С++ існує функція **getline()**. Вона є членом кожного потокового класу. Її прототипи виглядають у такий спосіб.

```
istream& getline(char* buf, streamsize num);
istream& getline(char* buf, streamsize num, char delim);
```

Перша форма функції **getline** зчитує символи з масиву, на який посилається покажчик **buf** поки не будуть зчитані num-1 символів, виявлений символ переходу на наступний рядок або досягнутий кінець файлу. Функція getline() записує нульовий символ у кінець масиву, на який посилається покажчик **buf** Символ переходу на новий рядок видаляється з потоку, але не записується в буфер.
Друга форма функції **getline** зчитує символи в масив, на який посилається покажчик **buf**, поки не будуть зчитані num-1 символів, виявлений символ **delim** або досягнутий кінець файлу. Функція **getline()** записує нульовий символ у кінець масиву, на який посилається покажчик **buf.** Символ **delim** видаляється з потоку, але не записується в буфер.
Як бачимо, дві версії функції **getline()** дуже нагадують версії **get (buf, num)** і **get(buf, num, delim)**. Різниця полягає в тому, що на відміну від функції **get()** функція **getline()** видаляє роздільник з потоку.
Розглянемо програму, що демонструє роботу функції **getline()**. Вона зчитує вміст текстового файлу й виводить його на екран.

```
// Програма зчитує й виводить на екран рядки, зчитані з текстового файлу.
#include <iostream>
#include <fstream>
using namespace std;
int main() {

	ifstream in("myfile"); // Уведення
	if(!in) {
		cout << "Неможливо відкрити файл\n";
			return 1;
	}
	char str[255];
	while(in) {
		in.getline(str, 255);  // За замовчуванням delim == '\n'.
		if(in) cout << str << endl;
	}
	in.close();
	return 0;
}
```

Розпізнавання кінця файлу

Розпізнати кінець файлу можна за допомогою функції **eof()**, що має прототип

```
bool eof();
```

Якщо досягнуто кінець файлу, вона повертає значення true, а якщо ні, то вона повертає значення false.

Функція ignore()

За допомогою функції Ignore () можна вважати й проігнорувати символ із вхідного потоку.
| istream &ignore(streamsize num=l, int_type delim=EOF);
Вона зчитує й відкидає символи, поки не будуть пропущені пит символів (за замовчуванням параметр пит рівний 1) або не зустрінеться символ delim, який за замовчуванням дорівнює константі EOF. Виявлений роздільник не віддаляється з потоку введення. Тип int_type визначений як різновид типу int.
Наступна програма зчитує файл TEST. Вона ігнорує символи, поки не зустрінеться пробіл або не будуть лічено 10 символів. Потім вона виводить на екран іншу частину файлу.

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
	ifstream in("test");
```

```
        if(!in) {
        cout << "Неможливо відкрити файл.\n";
              return 1;
}
/* Ігноруються 10 символів, поки не зустрінеться пробіл. */
        in.ignore(10, ' ');
        char c;
        while(in) {
        in.get(c);
        if(in) cout << c;
        }
              in.close();
              return 0;
}
```

Функції peekq і putback()

Можна отримати наступний символ з потоку, не витягаючи його звідти. Для цього призначена функція **peekq** (), що має прототип, наведений нижче.

```
int_type peekq();
```

Ця функція повертає наступний символ з потоку введення або ознака кінця файлу. Тип int_type визначений як різновид типу int
Символ, зчитаний з потоку останнім, можна повернути назад за допомогою функції putback(). Її прототип має такий вигляд.

```
istream& putback(char c);
```

Тут параметр с означає символ, зчитаний останнім.

Функція flush()

При виведенні дані не відразу передаються фізичному обладнанню, пов'язаному з потоком. Замість цього вони накопичуються у внутрішньому буфері, поки він не заповниться. Однак існує спосіб примусово записати інформацію з буфера на диск, не чекаючи його заповнення. Для цього призначена функція flush(). Її прототип має такий вигляд.

```
ostream& flush();
```

Функцію flush слід викликати, коли програма виконується в несприятливих умовах (наприклад, якщо часто відбуваються збої харчування).
Закриття файлу або припинення роботи програми також очищає всі буфери.

Довільний доступ

Довільний доступ до файлу забезпечується функціями seekg() і seekp(). Їхні прототипи мають такий вигляд.

```
istream& seekg(off_type offset, seekdir origin);
ostream& seekp(off_type offset, seekdir origin);
```

Тип off_type є різновидом цілого типу. Він визначений у класі ios і дозволяє зберігати максимальні значення, які може ухвалювати параметр offset. Тип seekdir являє собою перерахування, певне в класі ios. У ньому втримуються різновиди пошуку, виконуваного функціями seekg і seekp.
Система введення- виведення мови С++ управляє двома покажчиками, пов'язаними з файлами. Перший покажчик, що визначає позицію, у якій виконується читання файлу, називається курсором читання (get pointer). Інший покажчик, що визначає позицію, у якій виконується запис у файл, називається курсором запису (put pointer). Щораз при виконанні введення й виведення відповідний курсор файлу переміщається на одну позицію вперед. Однак функції seekg() і seekpo дозволяють виконувати довільні переміщення по файлу.
Функція seekg() переміщає пов'язаний з нею курсор запису на offset символів, відраховуючи від позиції origin. Позиція origin задається трьома можливими значеннями.

```
ios::beg       // Початок файлу
ios::cur     //   Поточне положення
ios::end     //   Кінець файлу
```

Функція seekp() переміщає пов'язаний з нею курсор запису на offset символів, відраховуючи від позиції origin. Позиція origin задається трьома можливими значеннями зазначеними вище.
Як правило, довільний доступ при введенні- виведення здійснюється тільки до файлу, відкритого в бінарному режимі. Перетворення символів при зчитуванні текстових файлів може порушити правильний порядок проходження байтів у файлі.
Наступна програма демонструє роботу функції seekp(). Вона дозволяє змінювати зазначений символ у файлі.. Зверніть увагу на те, що файл відкритий для операцій вводу-виведення.

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

            cout << "Застосування: CHANGE < ім'я файлу> <старий символ> <новий символ>\n";

                  fstream out("myfile.in", ios::in | ios::out | ios::binary);
                  if(!out) {
                  cout  <<   "Неможливо  відкрити  файл.";
                        return  1;
            }
                  int numb_char; char new_char;
                  cin >> numb_char >> new_char;

            out.seekp(numb_char, ios::beg);
            out.put(new_char); out.close();
            return 0;
      }
```

Наступна програма використовує функцію seekg(). Вона виводить на екран уміст файлу, починаючи з позиції, зазначеної в командному рядку.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
      int main(int argc, char* argv[]) {
      char ch;
      if(argc != 3) {
            cout << "Застосування: SHOW < ім'я файлу> <початкова позиція>\n";
                  return 1;
      }
      ifstream in(argv[1], ios::in | ios::binary);
      if(!in) {
            cout << "Неможливо відкрити файл.";
                  return 1;
      }
      in.seekg(atoii(argv[2]), ios::beg);
      while(in.get(ch)) cout << ch;
            return 0;
}
```

Нижче показане, як за допомогою функцій seekp() і seekg() переставити у зворотному порядку перші пит символів у файлі.

```
#include <iostream>
#include <fstream>
#include <cstdlib>
using namespace std;
int main(int argc, char* argv[]) {
      if(argc != 3) {
```

```
		cout << "Застосування: Reverse < ім'я файлу> <num>\n"; return 1; }
			fstream inout(argv[1], ios::in | ios::out | ios::binary);
		if(!inout) {
			cout << "Неможливо відкрити файл.\n";
			return 1;
		}
		long e, i, j;
		char c1, c2;
		e = atol(argv[2]);
		for (i = 0, j = e; i < j; i++, j--) {
			inout.seekg(i, ios::beg); inout.get(c1); inout.seekg(j, ios::beg);
inout.get(c2);
			inout.seekp(i, ios::beg); inout.put(c2); inout.seekp(j, ios::beg);
inout.put(c1);
		}
		inout.close(); return 0;
	}
```

Визначення поточної позиції

Поточну позицію курсорів читання й записи можна визначити за допомогою функцій tellg() і tellp(). Їхні прототипи мають такий вигляд

```
pos_type   tellg();
pos_type  tellp();
```

Тип pos_type визначений у класі ios і дозволяє зберігати максимальне значення, яке може повернути функція. Значення, що вертаються функціями tellp() і tellg(), можна використовувати в якості аргументів функції seekg() і seekp() відповідно.

```
istream& seekg(pos_type pos);
ostream& seekg(pos_type pos);
```

Ці функції дозволяють зберегти поточне положення файлового курсору, виконати певні файлові операції, а потім відновити колишнє положення курсору.

Статус уведення- виведення

Система введення- виведення мови C++ зберігає інформацію про результат каждбй операції вводу-виведення. Поточний стан системи введення- виведення зберігається в об'єкті класу iostate, який є перерахуванням, певним у класі ios. Крім цього, клас ios містить наступні члени.

```
/Ім'я                          Значення
ios::goodbit           //  Набір байтів, що описують нормальний стан.
ios::eofbit           //    1  якщо виявлений кінець файлу, 0 а якщо ні.
ios::failbit          //   1, якщо виявлена(можливо) поправна помилка, 0 а якщо ні.
ios::badbit         //   1, якщо виявлена непоправна помилка, 0 а якщо ні.
```

Існують два способи одержати інформацію про статус вводу-виведення. По-перше, можна викликати функцію rdstate(). Вона має наступний прототип.

```
iostate  rdstate();
```

Ця функція повертає поточний стан прапорів помилок. Як випливає из вищесказаного, якщо ніяких помилок не виявлене, функція rdstate () повертає значення goodbit. А якщо ні, то встановлюється прапор помилки.

Застосування функції rdstate() ілюструється наступною програмою.

```
#include  <iostream>
#include  <fstream>
using namespace  std;
void checkstatus(ifstream& in);
int main(int argc, char* argv[]) {
	if(argc != 2) {
		cout << "Застосування: Display < ім'я файлу>\n"; return 1; }
	ifstream in(argv[1]);
			if(!in) {
```

```
            cout << "Неможливо відкрити файл.\n";
            return 1;
      }
      char c;
      while(in.get(c)) {
      if(in) cout << c;
      checkstatus(in); }
      checkstatus(in);  // Перевірка заключного стану
      in.close();
      return 0; }
      void checkstatus(ifstream & in) {
            ios::iostate i;
            i = in.rdstate();
            if(i & ios::eofbit)
                  cout << "Виявлений  кінець файлу\n";
            else   if(i & ios::failbit)
                  cout << "Виявлена поправна  помилка\n";
            else  if(i & ios::badbit)
                  cout   <<   "Виявлена  непоправна  ошибка\n"; }
```

Ця програма завжди виявляє одну "помилку". Після завершення циклу while останній виклик функції checkstatus (), як і очікувалося, виявляє кінець файлу. Ця функція може виявитися корисної в будь-якій програмі.
Другий спосіб виявлення помилки заснований на застосуванні наступних функцій.

```
      bool  bad();
      bool  eof();
      bool   fail();
      bool  good();
```

Функція bad() повертає значення true, якщо встановлений прапор badbit. Функція eof () повертає значення true, якщо встановлений прапор failbit. Функція good() повертає значення true, якщо ніяких помилок не виявлене. А якщо ні, то функція повертає значення false.
Прапори, відповідні до виявлених помилок, можна скинути. Для цього слід викликати clear (), прототип якої має такий вигляд.

```
void clear(iostate flags = ios::goodbit);
```

Якщо параметр flags є об'єктом goodbit (за замовчуванням), усі прапори помилок скидаються. А якщо ні, то параметр flags слід задати довільно.

ПІДСУМКИ

Відкриття файлу

```
#include <fstream>
#include <iostream>

int main() {
    std::ifstream inputFile("input.txt"); // для читання
    std::ofstream outputFile("output.txt"); // для запису

    if (!inputFile.is_open() || !outputFile.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1; // повертаємо код помилки
    }

    // ваш код для роботи з файлами

    inputFile.close(); // закриваємо файл після використання
    outputFile.close();

    return 0; // програма завершилася успішно
}
```

Читання з файлу:

```
#include <fstream>
#include <iostream>
```

```
#include <string>

int main() {
    std::ifstream inputFile("input.txt");

    if (!inputFile.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1;
    }

    std::string line;
    while (std::getline(inputFile, line)) {
        std::cout << line << std::endl;
    }

    inputFile.close();

    return 0;
}
```

Запис у файл:

```
#include <fstream>
#include <iostream>

int main() {
    std::ofstream outputFile("output.txt");

    if (!outputFile.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1;
    }

    outputFile << "Hello, World!" << std::endl;

    outputFile.close();

    return 0;
}
```

Позиціювання покажчика файлу:

```
#include <fstream>
#include <iostream>

int main() {
    std::fstream file("example.txt", std::ios::in | std::ios::out);

    if (!file.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1;
    }

    file.seekg(5, std::ios::beg); // Переміщення покажчика для читання на 5 байт від початку
файлу
    file.seekp(10, std::ios::beg); // Переміщення покажчика для запису на 10 байт від початку
файлу

    // ваш код

    file.close();

    return 0;
}
```

Перевірка кінця файлу:

```
#include <fstream>
#include <iostream>
#include <string>

int main() {
    std::ifstream file("example.txt");
```

```
    if (!file.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1;
    }

    std::string content;
    while (!file.eof()) {
        std::getline(file, content);
        std::cout << content << std::endl;
    }

    file.close();

    return 0;
}
```

Перейменування файлу:

```
#include <iostream>

int main() {
    const char* oldName = "old_name.txt";
    const char* newName = "new_name.txt";

    if (std::rename(oldName, newName) != 0) {
        std::cerr << "Error renaming file." << std::endl;
        return 1;
    }

    return 0;
}
```

Перевірка існування файлу:

```
#include <iostream>
#include <fstream>

int main() {
    const char* fileName = "example.txt";

    std::ifstream file(fileName);
    if (file.good()) {
        std::cout << "File exists." << std::endl;
    }
    else {
        std::cout << "File does not exist." << std::endl;
    }

    return 0;
}
```

Видалення файлу:

```
#include <iostream>

int main() {
    const char* fileName = "example.txt";

    if (std::remove(fileName) != 0) {
        std::cerr << "Error deleting file." << std::endl;
        return 1;
    }
    else {
        std::cout << "File successfully deleted." << std::endl;
    }

    return 0;
}
```

Отримання розміру файлу:

```
#include <iostream>
#include <fstream>

int main() {
    const char* fileName = "example.txt";

    std::ifstream file(fileName, std::ios::binary | std::ios::ate);
```

```
    if (!file.is_open()) {
        std::cerr << "Unable to open file." << std::endl;
        return 1;
    }

    std::streamsize fileSize = file.tellg();
    std::cout << "File size is: " << fileSize << " bytes." << std::endl;

    file.close();

    return 0;
}
```

Перенаправлення потоків:

```
#include <iostream>
#include <fstream>

int main() {
    std::ofstream outFile("output.txt");
    std::streambuf* coutbuf = std::cout.rdbuf(); // зберігаємо стандартний буфер cout

    std::cout.rdbuf(outFile.rdbuf()); // перенаправляємо вивід у файл

    std::cout << "This goes to the file." << std::endl;

    std::cout.rdbuf(coutbuf); // повертаємо стандартний буфер

    std::cout << "This goes to the console." << std::endl;

    return 0;
}
```

Зчитування та запис у бінарному режимі:

```
#include <iostream>
#include <fstream>

struct Data {
    int value;
    double price;
};

int main() {
    const char* fileName = "binary_data.bin";

    // Запис у бінарний файл
    std::ofstream binaryOutFile(fileName, std::ios::binary);
    if (!binaryOutFile.is_open()) {
        std::cerr << "Error opening binary file for writing." << std::endl;
        return 1;
    }

    Data data1 = { 42, 3.14 };
    binaryOutFile.write(reinterpret_cast<char*>(&data1), sizeof(Data));
    binaryOutFile.close();

    // Читання з бінарного файлу
    std::ifstream binaryInFile(fileName, std::ios::binary);
    if (!binaryInFile.is_open()) {
        std::cerr << "Error opening binary file for reading." << std::endl;
        return 1;
    }

    Data data2;
    binaryInFile.read(reinterpret_cast<char*>(&data2), sizeof(Data));
    binaryInFile.close();

    std::cout << "Read data: " << data2.value << ", " << data2.price << std::endl;

    return 0;
}
```

Використання буферу для зчитування та запису:

```cpp
#include <iostream>
#include <fstream>
#include <vector>

int main() {
    const char* fileName = "buffered_data.txt";

    // Запис у файл з використанням буфера
    std::ofstream outFile(fileName);
    if (!outFile.is_open()) {
        std::cerr << "Error opening file for writing." << std::endl;
        return 1;
    }

    std::vector<std::string> lines = { "Line 1", "Line 2", "Line 3" };

    for (const auto& line : lines) {
        outFile << line << std::endl;
    }

    outFile.close();

    // Читання з файлу з використанням буфера
    std::ifstream inFile(fileName);
    if (!inFile.is_open()) {
        std::cerr << "Error opening file for reading." << std::endl;
        return 1;
    }

    std::vector<std::string> readLines;
    std::string buffer;
    while (std::getline(inFile, buffer)) {
        readLines.push_back(buffer);
    }

    inFile.close();

    for (const auto& line : readLines) {
        std::cout << line << std::endl;
    }

    return 0;
}
```

Виняткові ситуації

Одним з найбільш яскравих утілень принципу об'єктно-орієнтованого програмування є механізм обробки виняткових ситуацій у мові C++. У ході виконання програми можуть виявитися різні помилки. Вони можуть бути позв'язані з неправильним програмуванням (наприклад, вихід індексу масиву за межі припустимого чи переповнення пам'яті), а іноді їхня причина не залежить від програміста (наприклад, розрив зв'язку при мережевому з'єднанні). У кожній з цих ситуацій реакція програми непередбачена. Іноді вона завершує виконання, і лише після закінчення деякого інтервалу часу починають проявлятися наслідки помилки, а частіше програма негайно припиняє роботу, піддаючи ризику дані, що знаходяться в пам'яті чи у файлі. Якщо не передбачити акуратне завершення роботи, використовуючи обробку виняткових ситуацій, результати можуть виявитися неприємними.

В подальшому ми будемо називати винятковою ситуацією будь-яку подію, що вимагає особливої обробки. При цьому зовсім неважливо, чи є ця подія фатальною чи простою помилкою. Перевірка умов, що описують виняткову ситуацію, і реакція на її виникнення називається обробкою виняткової ситуації. Ця задача покладається на оброблювача виняткової ситуації.

Механізм обробки виняткових ситуацій

Обробка виняткових ситуацій у мові C++ є об'єктно-орієнтованою. Це значить, що виняткова ситуація є об'єктом, що генерується при виникненні незвичайних умов, передбачених програмістом, і передається оброблювачу, що її перехоплює. Об'єктом, що описує природу виняткової ситуації, може бути будь-яка сутність — літерал, рядок, об'єкт класу, число і т.д. Не слід думати, що виняткова ситуація обов'язково повинна бути об'єктом якого-небудь класу.

Обробка виняткових ситуацій

В основі обробки виняткових ситуацій у мові C++ лежать три ключових слова: try, catch і throw. Якщо програміст підозрює, що визначений фрагмент програми може спровокувати помилку, він повинний занурити цю частину коду в блок try. Необхідно мати на увазі, що зміст помилки (за винятком стандартних ситуацій) визначає сам програміст. Це значить, що програміст може задати будь-яку умову, що приведе до створення виняткової ситуації. Після цього необхідно вказати, у яких умовах варто генерувати виняткову ситуацію. Для цієї мети призначене ключове слово throw. І нарешті, виняткову ситуацію потрібно перехопити й обробити в блоці catch. Ось як виглядає ця конструкція.

```
try {
        // Тіло блоку try
        if (умова) throw виняткова_ситуація
}
catch (тип1 аргумент) {
        // Тіло блоку catch
                                        }
        catch (тип2 аргумент) {
                // Тіло блоку catch
                                                }
                        . . .
                catch (тип N аргумент) {
                        // Тіло блоку catch
                                                        }
```

Розмір блоку try не обмежений. У нього можна занурити як один оператор, так і цілу програму. Один блок try можна зв'язати з довільною кількістю блоків catch. Оскільки кожен блок catch відповідає окремому типу виняткової ситуації, програма сама визначить, який з них виконати. У цьому випадку інші блоки catch не виконуються. Кожен блок catch має аргумент, що приймає визначене значення. Цей аргумент може бути об'єктом будь-якого типу. Якщо

програма виконана правильно й у блоці try не виникло жодної виняткової ситуації, усі блоки catch будуть зігноровані. Якщо в програмі виникла подія, яку програміст вважає небажаною, оператор throw генерує виняткову ситуацію. Для цього оператор throw повинний знаходитися усередині блоку try або усередині функції, викликуваної усередині блоку try.

Генерація винятків.

Для використання оператора throw застосовується ключове слово throw, а за ним вказується значення будь-якого типу даних, яке ви хочете задіяти, щоб сигналізувати про помилку. Як правило, цим значенням є код помилки, опис проблеми або спеціальний клас-виняток. Наприклад:

```
throw - 1; // генерація винятку типу int
throw ENUM_INVALID_INDEX; // генерація винятку типу enum
throw "Can not take square root of negative number"; // генерація винятку типу const char*
(рядок C-style)
throw dX; // генерація винятку типу double (змінна типу double, яка була визначена раніше)
throw MyException("Fatal Error"); // генерація винятку з використанням об'єкту класу
MyException
```

Кожен з цих рядків сигналізує про те, що виникла якась помилка, яку потрібно обробити.

Пошук винятків

Генерація винятків — це лише одна частина процесу обробки винятків. Повернемося до нашої аналогії з баскетболом: як тільки просвистів арбітр, що відбувається далі? Гравці зупиняються, і гра тимчасово припиняється. Звичайний хід гри порушений.

У мові C++ ми використовуємо ключове слово try для визначення блоку стейтментів (так званого «блоку try»). Блок try діє як спостерігач в пошуках винятків, які були викинуті будь-яким з операторів в цьому ж блоці try, наприклад:

```
try
{
	// Тут ми пишемо стейтменти, які генеруватимуть наступний виняток
	throw - 1; // типовий стейтмент throw
}
```

Обробка винятків

Поки арбітр не оголосить штрафний кидок, і поки цей штрафний кидок не буде виконано, гра не відновиться. Іншими словами, штрафний кидок повинен бути “оброблений” до відновлення гри.

Фактично, обробка винятків — це робота блоку(ів) catch. Ключове слово catch використовується для визначення блоку коду (так званого «блоку catch»), який обробляє винятки певного типу даних.

Ось приклад блоку catch, який обробляє (ловить) винятки типу int:

```
catch (int a)
{
	// Обробляємо виняток типу int
	cerr << "We caught an int exception with value" << a << '\n';
}
```

Блоки try і catch працюють разом. Блок try виявляє будь-які винятки, які були викинуті в ньому, і направляє їх до відповідного блоку catch для обробки. Блок try повинен мати, принаймні, один блок catch, який знаходиться відразу ж за ним, але також може мати і кілька блоків catch, розміщених послідовно (один за одним).

Як тільки виняток було спіймано блоком try і направлено в блок catch для обробки, він вважається обробленим (після виконання коду блоку catch), і програма відновлює своє виконання.

Параметри catch працюють так само, як і параметри функції, причому параметри одного блоку catch можуть бути доступні і в іншому блоці catch (який знаходиться за ним). Винятки фундаментальних типів даних можуть бути спіймані по значенню (параметром блоку

catch є значення), але винятки нефундаментальних типів даних повинні бути спіймані по константному посиланню (параметром блоку catch є константне посилання), щоб уникнути непотрібного копіювання.

Як і у випадку з функціями, якщо параметр не використовується в блоці catch, то ім'я змінної можна не вказувати:

```
catch (double) // примітка: Ми не вказуємо ім'я змінної, тому що в цьому немає необхідності (ми її ніде в блоці не використовуємо)
{
	// Обробляємо виняток типу double тут
	cerr << "We caught an exception of type double" << '\n';
}
```

Це запобіжить виведенню попереджень від компілятора про невикористані змінні.

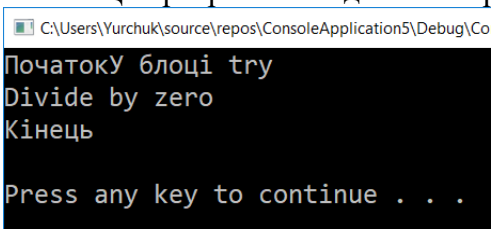
Якщо в програмі виникла виняткова ситуація, для якої не передбачені перехоплення й обробка, викликається стандартна функція terminate(), що, у свою чергу, викликає функцію abort(). Утім, іноді виняткова ситуація не є небезпечна. У цьому випадку можна виправити помилку (наприклад, привласнити нульовому знаменнику ненульове значення) і продовжити виконання програми.

Розглянемо найпростіший приклад.

Обробка виняткової ситуації

```
int n = 10, m = 0;
	cout << "Початок";
	try {
		cout << "У блоці try\n";
		if (m == 0) throw "Divide by zero \n"; else n = n / m;
		cout <<"Подальша частина блоку не виконується!";
	}
	catch (const char* s) {
		cout << s;
	}
	cout << "Кінець\n";
```

Ця програма виводить на екран наступні рядки.



Простежимо за потоком керування при виконанні цієї програми. Спочатку з'являються і ініціалізуються дві цілочисельні змінні (одна з них дорівнює нулю). Потім виводиться повідомлення про початок виконання програми, і потік керування входить у блок try. Після виводу рядка повідомлення про вхід у блок try, потік керування переходить до перевірки рівності m==0. Оскільки ця рівність є істиною, генерується виняткова ситуація (у даному випадку — константний рядок). Керування негайно передається блоку catch, аргументом якого є константний символьний вказівник, ігноруючи всі інші оператори в блоці try. У цій програмі блок catch не робить жодних спроб виправити помилку. Замість цього він просто видає повідомлення — рядок, отриманий як аргумент — і передає керування оператору, що слідує за блоком. На закінчення виводиться на екран рядок Кінець, і програма завершує свою роботу.

Тип виняткової ситуації повинен збігатися з типом аргументу розділу catch. Поглянемо, що відбудеться, якщо цією умовою зневажити.

Порушення угоди про тип виняткової ситуації

```
	int n = 10, m = 0;
	cout << "Початок";
	try {
```

```
		cout << "У блоці try\n";
		if (m == 0) throw "Divide by zero\n"; else n = n / m;
		cout <<"Подальша частина блоку не виконується!";
	}
	catch (const char s) // Помилка! Необхідно const char* s! {
		cout << s;
	}
	cout << "Кінець\n";
```

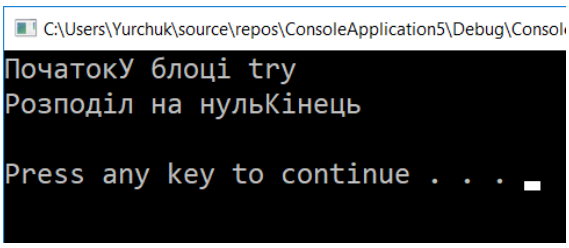
У цій програмі ми зробили цілком “природну” помилку — забули поставити зірочку в оголошенні аргументу. Тепер блок catch очікує виняткову ситуацію, що представляє собою константний символ, а не вказівник. Ця помилка приводить до аварійного завершення роботи програми.

Покажемо, що відбудеться, якщо виняткова ситуація генерується усередині функції, яка викликається в блоці try.

Виняткова ситуація**,** згенерована усередині функції

```
int Denominator(int);
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);


	int n = 10, m;
	cout << "Початок";
	try {
		cout << "У блоці try\n";
		m = Denominator(0);
		cout << "Подальша частина блоку не виконується!";
	}
	catch (const char *s) // Помилка! Необхідно const char* s!
	{
		cout << s;
	}
	cout << "Кінець\n";

	cout << endl;
	system("pause");
	return 0;
}
int Denominator(int i) {
	if (i == 0)throw "Розподіл на нуль";
	return i;
}
```

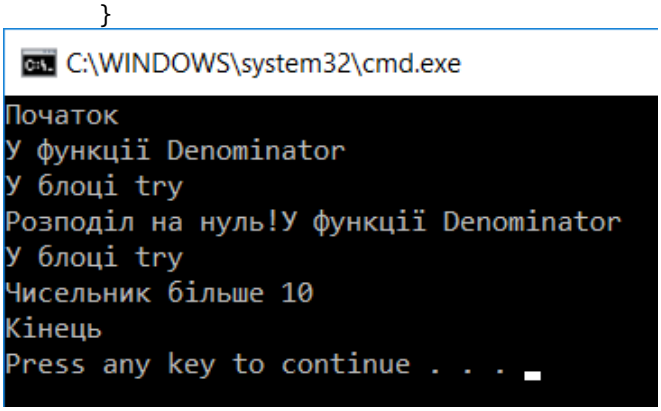
У цій програмі виняткова ситуація генерується у функції Denominator(), яка викликається в блоці try. Завдяки цьому результати роботи програми цілком збігаються з попередніми.



Якщо блок try знаходиться усередині функції, обробка виняткової ситуації виконується при кожному виклику.

Розміщення блоку try усередині функції

```
int Denominator(int);
int main()
{
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);


	int n = 10, m;
	cout << "Початок\n";
	m = Denominator(0);
	n = Denominator(11);
	cout << "Кінець\n";
	return 0;
}
int Denominator(int i) {
	cout << "У функції Denominator\n";
	try
	{
		cout << "У блоці try\n";
		if (i == 0) throw("Розподіл на нуль!");
		if (i>10) throw 10;
		cout << "Подальша частина блоку не виконується!";
	}

		catch (const char* s)
		{
			cout << s;
		}
		catch (int n)
		{
			cout << "Чисельник більше " << n << endl;
			return i;
		}

	}
```



У цій програмі передбачене перехоплення двох виняткових ситуацій. Перша з них має тип const char* і генерується, коли знаменник дорівнює нулю, а друга — тип int і генерується, коли чисельник перевищує 10. Як бачимо, ці виняткові ситуації перевіряються і перехоплюються при кожнім виклику функції Denominator().

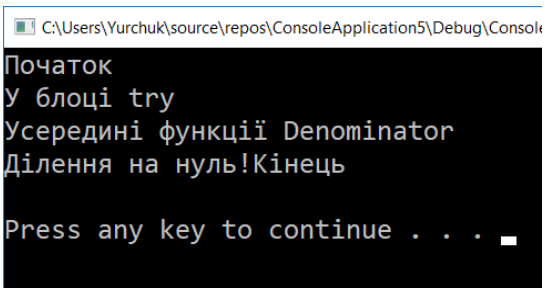
Розглянемо тепер приклад, у якому функція Denominator() лише генерує виняткові ситуації, а їх обробка здійснюється у функції main().

Окрема обробка виняткових ситуацій

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

using namespace std;
int Denominator(int);
int main() {
	SetConsoleCP(1251);
```

```
	SetConsoleOutputCP(1251);

	int n = 10, m;
	cout << "Початок\n";
	try {
		cout << "У блоці try\n";
		m = Denominator(0);
		n = Denominator(11);
		cout << "Подальша частина блоку не виконується!";
	}
	catch (const char* s) {

		cout << s;
	}

	catch (int n) {
		cout << "Чисельник більше 10\n" << n;
	}
	cout << "Кінець\n";

	cout << endl;
	system("pause");
	return 0;
}
int Denominator(int i) {
	cout << "Усередині функції Denominator\n";
	if (i == 0) throw("Ділення на нуль!");
	if (i>10) throw 10;
	cout << "Кінець функції Denominator\n";
	return i;
}
```

Результат демонструє декілька важливих особливостей, властивим функціям, що спричиняють, але не обробляють виняткову ситуацію.



По-перше, блоки try і catch нерозривні. Не можна помістити блок try у функцію, залишивши блок catch у функції main(). Необхідно або обробити виняткову ситуацію усередині функції, як це зроблено в одному із попередніх прикладів, або перенести обробку в модуль виклику. У першому випадку функція, завершивши обробку, повертає визначене її специфікацією значення, а в другому — виняткову ситуацію. Таким чином, можна обійти обмеження мови C++, відповідно до якого функція може повертати лише одне значення, тип якого визначений заздалегідь. По-друге, механізм обробки виняткових ситуацій дозволяє створювати альтернативні значення, що повертаються. По-третє, функції можуть генерувати декілька виняткових ситуацій. Згенерувавши одну з них, вони негайно припиняють своє виконання і повертають виняткову ситуацію в модуль виклику. Необхідно враховувати, що присвоювання m=Denominator(0) чи n=Denominator(11) у цьому випадку не виконуються.

Приклад розкручування стеку

Коли функція викликає іншу функцію, а та в свою чергу ще іншу функцію, то такі функції поміщаються в стек, відповідно вверху стеку знаходиться функція, яка обробляється в даний момент часу, при завершенні роботи вона видаляється із стеку, на її місце підсувається попередня функція, і так далі. Такий процес називається розкручування стеку.

Тут у нас вже більший стек. Хоча все здається занадто складним, але насправді це не

так:

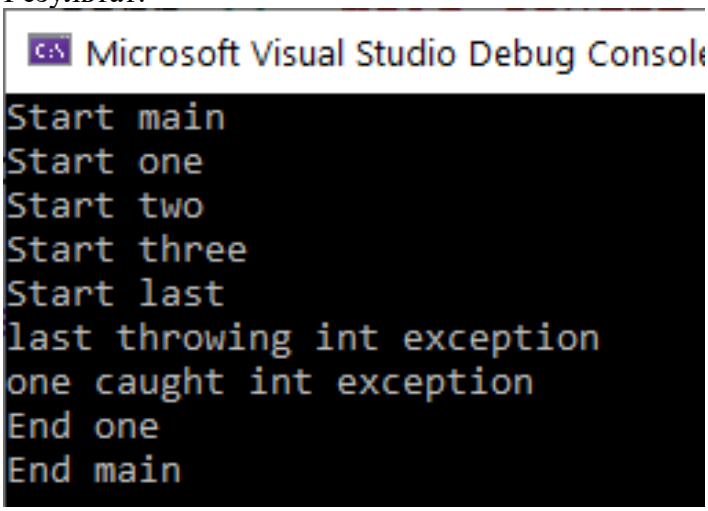
main() викликає one();
one() викликає two();
two() викликає three();
three() викликає last();
last() викидає виняток.

Дивимося:

```
#include <iostream>
using namespace std;
void last() // викликається функцією three()
{
    cout << "Start last\n";
   cout << "last throwing int exception\n";
    throw - 1;
   cout << "End last\n";

}

void three() // викликається функцією two()
{
   cout << "Start three\n";
    last();
    cout << "End three\n";
}

void two() // викликається функцією one()
{
   cout << "Start two\n";
    try
    {
        three();
    }
    catch (double)
    {
       cerr << "two caught double exception\n";
    }
    cout << "End two\n";
}

void one() // викликається функцією main()
{
   cout << "Start one\n";
    try
    {
        two();
    }
    catch (int)
    {
        cerr << "one caught int exception\n";
    }
    catch (double)
    {
        cerr << "one caught double exception\n";
    }
   cout << "End one\n";
}

int main()
{
   cout << "Start main\n";
    try
    {
        one();
    }
    catch (int)
    {
```

```
        cerr << "main caught int exception\n";
    }
    cout << "End main\n";

    return 0;
}
```

Погляньте на цю програму ще раз. Чи можете ви зрозуміти, що виведеться на екран? Результат:



Розглянемо хід виконання програми детально. Думаю не потрібно пояснювати виведення рядків Start. Функція last() виводить last throwing int exception, а потім викидає виняток типу int. Ось де починається найцікавіше.

Оскільки функція last() не обробляє винятки самостійно, то стек починає розкручуватися. Функція last() негайно завершує своє виконання, і точка виконання повертається назад в caller (в функцію three()).

Функція three() не обробляє жодних винятків, тому стек розкручується далі, виконання функції three() припиняється, і точка виконання повертається в two().

Функція two() має блок try, в якому знаходиться виклик three(), тому компілятор намагається знайти обробник винятків типу int, але, так як його не знаходить, точка виконання повертається назад в one(). Зверніть увагу, компілятор не виконує неявну конвертацію, щоб зіставити виняток типу int з обробником типу double.

Функція one() також має блок try з викликом two() всередині, тому компілятор дивиться, чи є відповідний обробник catch. Є — функція one() обробляє виняток і виводить one caught int exception.

Оскільки виняток було оброблено, то точка виконання переміщається в кінець блоку catch всередині one(). Це означає, що one() виводить End one, а потім завершує своє виконання, як зазвичай.

Точка виконання повертається назад в main(). Хоча main() має обробник винятків типу int, але наш виняток вже був оброблений функцією one(), тому блок catch всередині main() не виконується. Функція main() виводить End main, а потім завершує своє виконання.

З цієї програми можна зробити кілька цікавих висновків:

- По-перше, безпосередній caller, який викликає функцію, в якій викидається виняток, не зобов'язаний обробляти цей виняток, якщо він цього не хоче. У прикладі, наведеному вище, функція three() не обробляє виняток, який генерується функцією last(). Вона делегує цю відповідальність на іншого caller-а зі стеку.
- По-друге, якщо блок try не має обробника catch відповідного типу, то розкручування стеку відбувається так, наче цього блоку try не було взагалі. У прикладі, наведеному вище, функція two() не обробляє виняток, тому що у неї немає відповідного обробника catch.
- По-третє, коли виняток оброблено, виконання коду продовжується як зазвичай, починаючи з кінця блоку catch (в якому цей виняток був оброблений). У прикладі, наведеному вище, функція one() обробила виняток, а потім продовжила своє виконання

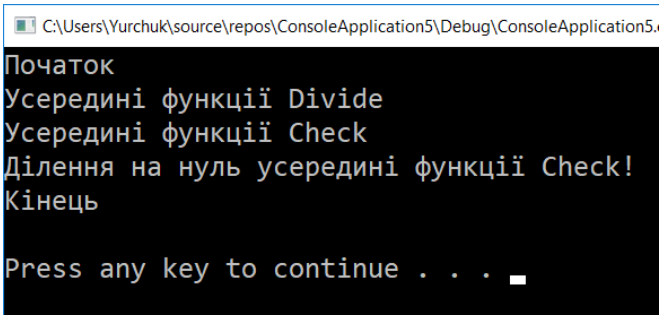
виведенням рядка End one. На той час, коли точка виконання повертається назад в функцію main(), виняток вже був згенерований і оброблений. Функція main() виконується так, наче цього винятку не було взагалі!

Розкручування стеку є дуже корисним механізмом, тому що дозволяє функціям не обробляти винятки, якщо вони цього не хочуть. Операція розкручування стеку виконується до тих пір, поки не буде виявлений відповідний блок catch! Таким чином, ми можемо самі вирішувати, де саме слід обробляти винятки.

Представимо тепер ланцюжок викликів функцій. Ланцюгове генерування виняткових ситуацій**:** перший варіант

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

using namespace std;
int Check(int);
int Divide(int, int);
int main() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);

	int n = 10, m = 0, l;
	cout << "Початок\n"; l = Divide(n, m); cout << "Кінець\n";
	cout << endl;
	system("pause");
	return 0;
}
int Check(int i) {
	cout << "Усередині функції Check\n";
	if (i == 0) throw("Ділення на нуль усередині функції Check!");
	cout << "Кінець функції Check\n";
	return i;
}
int Divide(int n, int m)
{
	cout << "Усередині функції Divide\n";
	try { m = Check(m); }
	catch (const char* s) {
		cout << s;
		return 1;
	}
	cout << "Кінець функції Divide\n";
	return n / m;
}
```

Простежимо за передачею виняткової ситуації.



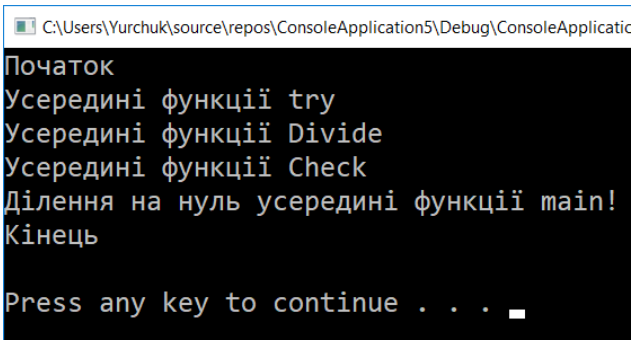
При виклику функції Divide() перевіряється знаменник m. Для цього викликається функція Check(). Якщо знаменник дорівнює нулю, усередині цієї функції генерується виняткова ситуація, що має тип const char*. Обробка цієї виняткової ситуації усередині функції Check() не передбачена, тому вона передається нагору по ланцюжку викликів — функції Divide(). Потім керування передається функції main(), і виконання програми завершується.

Перенесемо обробку виняткової ситуації у функцію main().

Ланцюгове генерування виняткових ситуацій: другий варіант

```
#include "windows.h"
#include <iostream>
using namespace std;
int Check(int);
int Divide(int, int);
int main() {
	SetConsoleCP(1251);
	SetConsoleOutputCP(1251);
	int n = 10, m = 0, l;
	cout << "Початок\n";
	try
	{
		cout << "Усередині функції try\n"; l = Divide(n, m);
	}
	catch (const char* s) {
		cout<<s<<endl;
	}

cout<<"Кінець\n";
cout << endl;
system("pause");
return 0;
}
int Check(int i) {
	cout << "Усередині функції Check\n";
	if (i == 0) throw("Ділення на нуль усередині функції main!"); cout << "Кінець функції
Check\n";
	return i;
}
int Divide(int n, int m) {

	cout<<"Усередині функції Divide\n"; m = Check(m);
	cout<<"Кінець функції Divide\n";
	return n / m;
}
```

Результат роботи цієї функції такий.



Оскільки усередині функцій Check() і Divide() обробка виняткової ситуації не передбачена, вона передається в головний модуль, про що свідчить представлений нижче рядок.

Неспіймані винятки

У наступному прикладі функція mySqrt() викидає виняток і припускає, що його хтось обробить. Але що станеться, якщо цього ніхто не зробить?

Ось наша програма обчислення квадратного кореня числа без блоку try в функції main():

```
#include <iostream>
#include <cmath> // для sqrt()
using namespace std;
// Окрема функція обчислення квадратного кореня числа
double mySqrt(double a)
```

```
{
    // Якщо користувач ввів від'ємне число,
    if (a < 0.0)
        throw "Can not take sqrt of negative number"; // то генерується виняток типу const
char*

    return sqrt(a);
}

int main()
{
   cout << "Enter a number: ";
    double a;
    cin >> a;

    // Тут немає ніякого обробника винятків!
   cout << "The sqrt of " << a << " is " << mySqrt(a) << '\n';

    return 0;
}
```

Тепер припустимо, що користувач ввів -5, і mySqrt(-5) згенерувало виняток. Функція mySqrt() не обробляє свої винятки самостійно, тому стек починає розкручуватися, і точка виконання повертається назад в функцію main(). Але, оскільки в main() також немає обробника винятків, виконання main() і всієї програми припиняється.

Коли main() завершує своє виконання з необробленим винятком, то операційна система зазвичай повідомляє нас про те, що відбулася помилка необробленого винятку. Як вона це зробить — залежить від кожної операційної системи окремо:

або виведе повідомлення про помилку;
або відкриє діалогове вікно з помилкою;
або просто збій.

Це те, що ми не повинні допускати, як програмісти!

Обробники всіх типів винятків

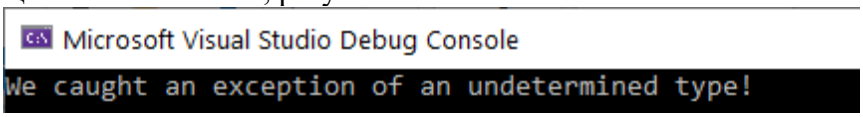
А тепер загадка: “Функції можуть генерувати винятки будь-якого типу даних, і, якщо виняток не спіймано, це призведе до розкручування стеку і потенційному завершенню виконання програми. Оскільки ми можемо викликати функції, не знаючи їх реалізації (і, відповідно, які винятки вони можуть генерувати), то як ми можемо цьому запобігти?”.

На щастя, мова C++ надає нам механізм виявлення/обробки всіх типів винятків — обробник catch-all. Обробник catch-all працює так само, як і звичайний блок catch, за винятком того, що замість обробки винятків певного типу даних, він використовує еліпсис (...) в якості типу даних.

А як ми вже знаємо, еліпсиси можуть використовуватися для передачі аргументів будь-якого типу даних в функцію. У цьому контексті вони представляють собою винятки будь-якого типу даних. Ось простий приклад:

```
#include <iostream>
using namespace std;
int main()
{
	try
	{
		throw 7; // генерується виняток типу int
	}
	catch (double a)
	{
		cout << "We caught an exception of type double: " << a << '\n';
	}
	catch (...) // обробник catch-all
	{
		cout << "We caught an exception of an undetermined type!\n";
	}
}
```

Оскільки для типу int не існує спеціального обробника catch, то обробник catch-all ловить цей виняток. Отже, результат:



Обробник catch-all повинен знаходитися останнім в ланцюжку блоків catch. Це робиться для того, щоб винятки спочатку могли бути спіймані обробниками catch, адаптованими до конкретних типів даних (якщо вони взагалі існують). У Visual Studio це контролюється, щодо інших компіляторів не впевнений, чи є таке обмеження.
Часто блок обробника catch-all залишають порожнім:

```
catch (...) {} // ігноруюься будь-які непередбачувані винятки
```

Цей обробник ловить будь-які непередбачені винятки і запобігає розкручуванню стеку (і потенційному завершенню виконання програми), але тут він не виконує ніякої обробки винятків.

Використання обробника catch-all в функції main()

Розглянемо наступну програму:

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

    try
    {
        runGame();
    }
    catch (...)
    {
        cerr << "Abnormal termination\n";
    }

    saveState(); // зберігаємо поточний стан гравця
    return 1;
}
```

В цьому випадку, якщо функція runGame() або будь-яка інша з функцій, які викликаються в runGame(), викине виняток, який не буде спіймано функціями в стеці вище, то, в кінцевому підсумку, він потрапить в обробник catch-all. Це запобіжить завершенню виконання функції main() і дасть нам можливість вивести повідомлення із зазначенням помилки на наш розсуд, а потім зберегти стан користувача до виходу з програми. Це може бути корисно для виявлення і усунення непередбачених проблем.

Специфікації винятків

Специфікації винятків — це механізм оголошення функцій із зазначенням того, чи генеруватиме функція винятки (і які саме) чи ні. Це може бути корисно при визначенні необхідності розміщення виклику функції в блоці try.

Існують три типи специфікації винятків, кожен з яких використовує так званий синтаксис throw (…).

По-перше, ми можемо використовувати порожній оператор throw для позначення того, що функція не генерує ніяких винятків, які виходять за її межі:

```
int doSomething() throw(); // не генеруються винятки
```

Зверніть увагу, функція doSomething() все ще може генерувати винятки, тільки обробляти вона повинна їх самостійно. Будь-яка функція, оголошена з використанням throw() (як у вищенаведеному прикладі), повинна негайно припинити виконання програми, якщо вона спробує згенерувати виняток, який призведе до розкручування стеку. Іншими словами, ми повідомляємо, що всі винятки функції doSomething(), функція doSomething() оброблятиме самостійно.

По-друге, ми можемо використовувати оператор throw із зазначенням типу винятку, який може генерувати ця функція:

```
int doSomething() throw(double); // можуть генеруватися винятки типу double
```

Нарешті, ми можемо використовувати еліпсис з оператором throw для позначення того, що функція може генерувати різні типи винятків:

```
int doSomething() throw(...); // можуть генеруватися будь-які винятки
```

Через погану (неповну) реалізацію і сумісність з компіляторами, і з огляду на той факт, що специфікації винятків більше нагадують заяви про наміри, ніж гарантії чого-небудь, і те, що вони погано сумісні з шаблонами функцій, і те, що більшість програмістів C++ не знають про їх існування, призводить до того, що використовувати специфікації винятків не рекомендується.