Лекція №1-2

Алгоритмізація і програмувння

КОМП'ЮТЕР ЯК ОСНОВНИЙ ІНСТРУМЕНТ ПРОГРАМУВАННЯ

Інформація в перекладі з латинської мови означає роз'яснення, виклад чого-небудь або відомості про що-небудь.

Історичний розвиток

- Потреба у точних і швидких розрахунках (арифметичні вирази, логарифмічні та тригонометричні функції).
- XIX ст. — науково-технічна революція → необхідність швидкодіючих лічильних машин.
- Початок XX ст. — розвиток електроніки → створення обчислювальних пристроїв на новій основі.
- **1945 р.** — Джон фон Нейман сформулював принципи побудови ЕОМ.
- **1946 р.** — створена перша електронна обчислювальна машина.
- Сучасні ЕОМ будуються на основі ідей фон Неймана.

АРХІТЕКТУРА ФОН НЕЙМАНА

Основні принципи (1945 р.)

- **Програмно-керований характер обчислень** — програма зберігається в пам’яті.
- **Послідовне виконання команд** (одна за одною).
- **Подання даних і команд у двійковій системі**.

Основні компоненти ЕОМ

- **Арифметико-логічний пристрій (АЛП)** — виконує обчислення й логічні операції.
- **Пристрій керування (ПК)** — організує виконання команд програми.
- **Пам’ять** — зберігає дані та програми.
- **Пристрої введення/виведення** — забезпечують взаємодію з користувачем.

📌 Важливо: усі сучасні комп’ютери побудовані на принципах цієї архітектури.

КОМП'ЮТЕР ЯК ОСНОВНИЙ ІНСТРУМЕНТ ПРОГРАМУВАННЯ

ОСНОВНІ ВИЗНАЧЕННЯ

За своїм призначенням комп'ютер – це універсальний прилад для роботи з інформацією.

Персональний комп'ютер (ПК) – це комп'ютер, призначений для обслуговування одного робочого місця. За своїми характеристиками він може відрізнятися від великих ЕОМ, але функціонально здатний виконувати аналогічні операції. За способом експлуатації розрізняють настільні (desktop), портативні (laptop і notebook) і кишенькові (palmtop) моделі ПК.

Сукупність апаратних засобів комп'ютера називають його апаратної конфігурацією.

Апаратне забезпечення (hardware) — фізичні компоненти комп’ютера, що:

- реєструють дані,
- зберігають інформацію,
- транспортують дані,
- перетворюють дані (за формою і змістом),
- представляють інформацію у формі, зручній для людини.

📌 Виконує всю фізичну роботу з даними.

ОСНОВНІ ВИЗНАЧЕННЯ

Програмне забезпечення.

Сукупність програм, що зберігаються на комп'ютері, утворює його програмне забезпечення. Сукупність програм, підготовлених до роботи, називають встановленим програмним забезпеченням. Сукупність програм, що працюють в той чи інший момент часу, називають програмною конфігурацією.

Процес створення і редагування програм називається програмуванням.

ПРИСТРОЇ КОМП'ЮТЕРА

Будь-який комп'ютер складається з чотирьох частин:

- пристрої введення інформації
- пристрою обробки інформації
- пристрої зберігання
- пристрої виведення інформації.

Конструктивно ці частини можуть бути об'єднані в одному корпусі розміром з книгу або ж кожна частина може складатися з декількох досить громіздких пристроїв.

Базова апаратна конфігурація ПК. Базовою апаратною конфігурацією персонального комп'ютера називають мінімальний комплект апаратних засобів, достатній для початку роботи з комп'ютером. З плином часу поняття базової конфігурації поступово змінюється.

Найчастіше персональний комп'ютер складається з наступних пристроїв:

- системний блок
- монітор
- клавіатура
- миша

Додатково можуть підключаться інші пристрої введення і виведення інформації, наприклад звукові колонки, принтер, сканер ...

ПРИСТРОЇ КОМП'ЮТЕРА

ПАМ'ЯТЬ ПК

Основні функції

- Збереження програм і даних.
- Забезпечення швидкого доступу до інформації.

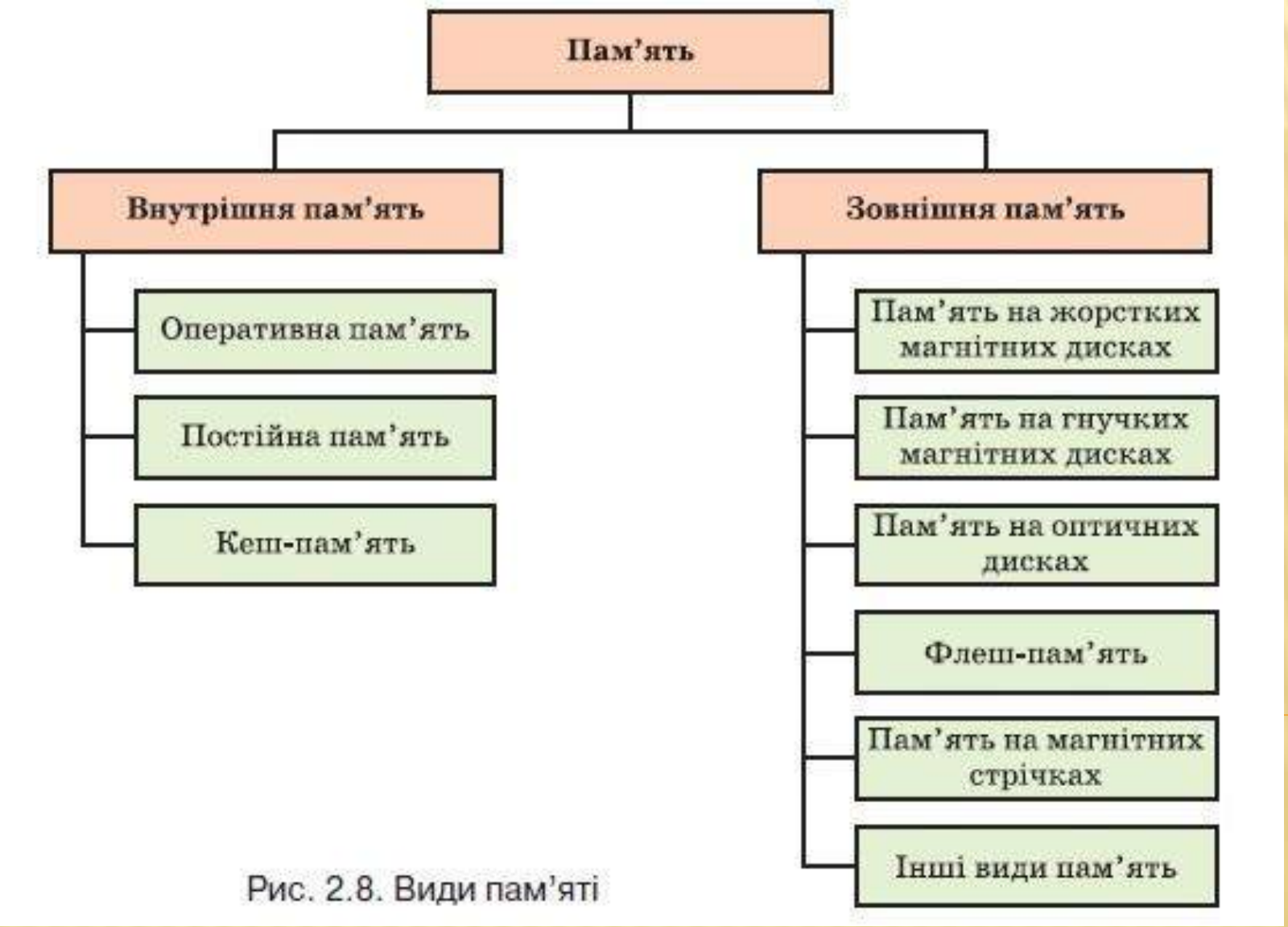


Рис. 2.8. Види пам'яті

ВИДИ ПАМ'ЯТІ ПК

За призначенням

- **Внутрішня пам'ять** – зберігає невеликі обсяги інформації під час обробки.
- **Зовнішня пам'ять** – довготривале зберігання великих обсягів даних, незалежно від живлення.

За залежністю від живлення

- **Енергозалежна** – стирається при вимкненні комп'ютера.
- **Енергонезалежна** – зберігає дані після вимкнення.

Приклади

Енергонезалежна внутрішня пам'ять

- ПЗУ (ROM) – записана виробником; містить базові програми керування (BIOS/UEFI).

Енергозалежна внутрішня пам'ять

- ОЗУ (RAM) – зберігає дані, програми, результати обчислень.
- Відеопам'ять – для зберігання зображень на екрані.
- Кеш-пам'ять – прискорює обмін між процесором і RAM (внутрішня – у процесорі, зовнішня – на материнській платі).

📌 Швидкість доступу: **Кеш > RAM > Зовнішня пам'ять**, але обсяг зростає у зворотному напрямку.

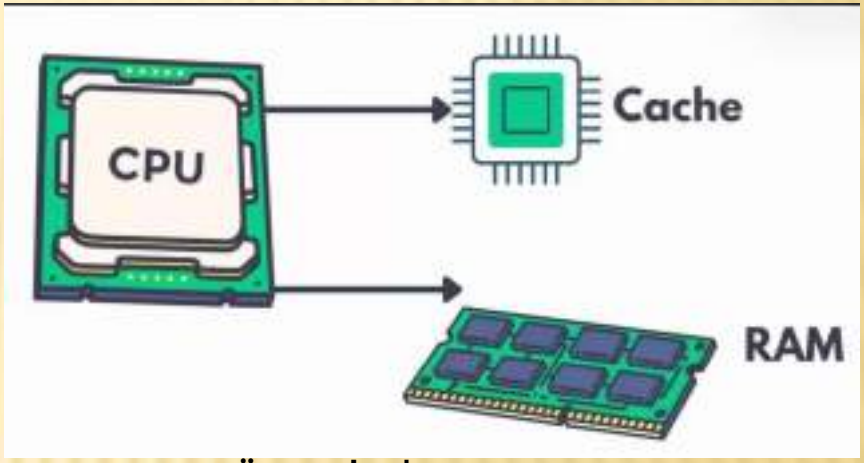
КЕШ-ПАМЯТЬ

Основні характеристики

- **Кеш** – проміжний буфер з дуже швидким доступом.
- Містить дані, які з найбільшою ймовірністю будуть потрібні процесору.
- Доступ до кешу значно швидший, ніж до RAM або зовнішніх носіїв.
- Обсяг обмежений у порівнянні з іншими видами пам’яті.

Розташування кешу

- Усередині центрального процесора (CPU).
- Може бути:
- інтегрований в ядро,
- спільний для кількох ядер.



Призначення

- Зменшує кількість звернень процесора до оперативної пам’яті.
- Прискорює виконання операцій і підвищує загальну продуктивність ПК.

📌 Види кешу: L1 (найшвидший, малий обсяг), L2 (більший, трохи повільніший), L3 (спільний для ядер, ще більший).

РЕГІСТРИ ПРОЦЕСОРА

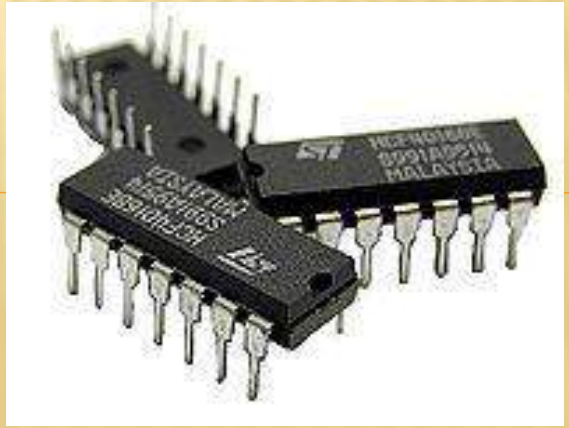
Що таке регістр?

- **Регістр** – спеціалізована осередок пам’яті всередині процесора.
- Виконує функції короткочасного зберігання та перетворення даних або команд.
- На фізичному рівні – це сукупність тригерів, кожен з яких зберігає 1 біт інформації.

Типові операції з регістрами

- Прийом слова в регістр (установка стану).
- Передача слова з регістру.
- Зсув слова вліво або вправо (зсувні регістри).
- Перетворення послідовного коду в паралельний і навпаки.
- Скидання – установка у початковий стан.

📌 Регістри – найшвидший рівень пам’яті, безпосередньо доступний для процесора.



ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ

Основні поняття

- **Програмне забезпечення (ПЗ)** – сукупність програм, що зберігаються на комп'ютері.
- **Встановлене ПЗ** – програми, підготовлені до роботи.
- **Програмна конфігурація** – програми, що працюють у певний момент часу.

Категорії програм:

Системні програми

- Виконують допоміжні функції:
  - створення резервних копій,
  - перевірка працездатності пристроїв,
  - управління ресурсами.

Прикладні програми

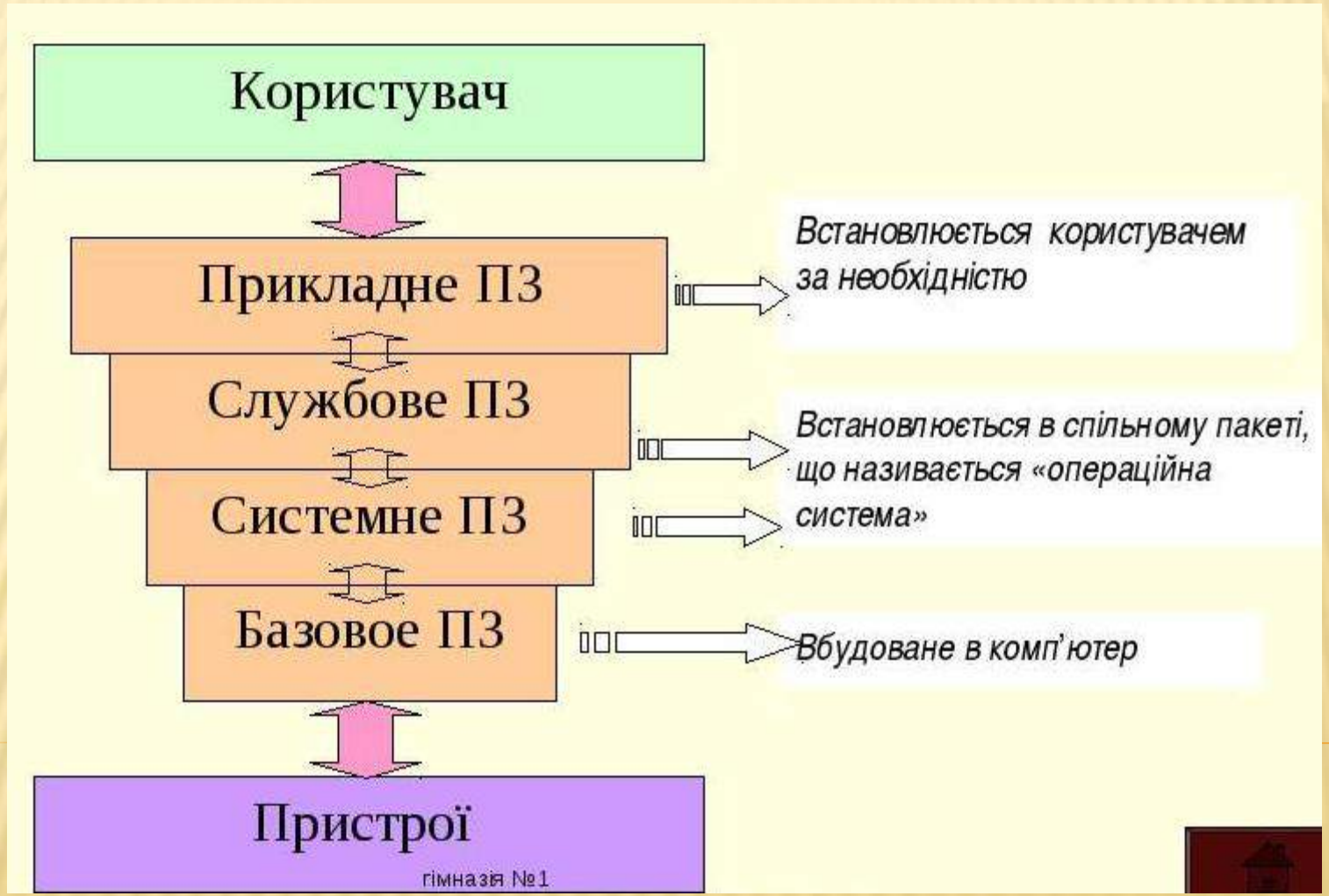
- Виконують робочі завдання користувача:
  - редагування текстів,
  - малювання графіки,
  - обробка баз даних, таблиць, масивів.

Інструментальні системи (системи програмування)

- Дають можливість створювати нові програми.

📌 Програмне забезпечення разом із апаратним утворює **комп'ютерну систему**.

ПРОГРАМНЕ ЗАБЕЗПЕЧЕННЯ



МОВИ ПРОГРАМУВАННЯ.

Мова програмування – формальна знакова система, призначена для опису алгоритмів в формі, яка зручна для виконавця (наприклад, ЕОМ, тобто комп'ютера).

Для машин першого покоління програми створювалися в машинному коді (або на машинному мовою). Це означає, що кожна операція, яку могла виконати ЕОМ, мала код (свій номер). Програміст, формуючи директиву для комп'ютера, повинен був вказати код операції та місця розташування інформації, над якою треба було виконати дану операцію. Наприклад, директива додавання двох чисел могла виглядати так:

001 0345 0363 0211.

У представленому коді набори чисел означають наступне:

001 – операція додавання,

0345 – адреса першого числа,

0363 – адреса другого числа,

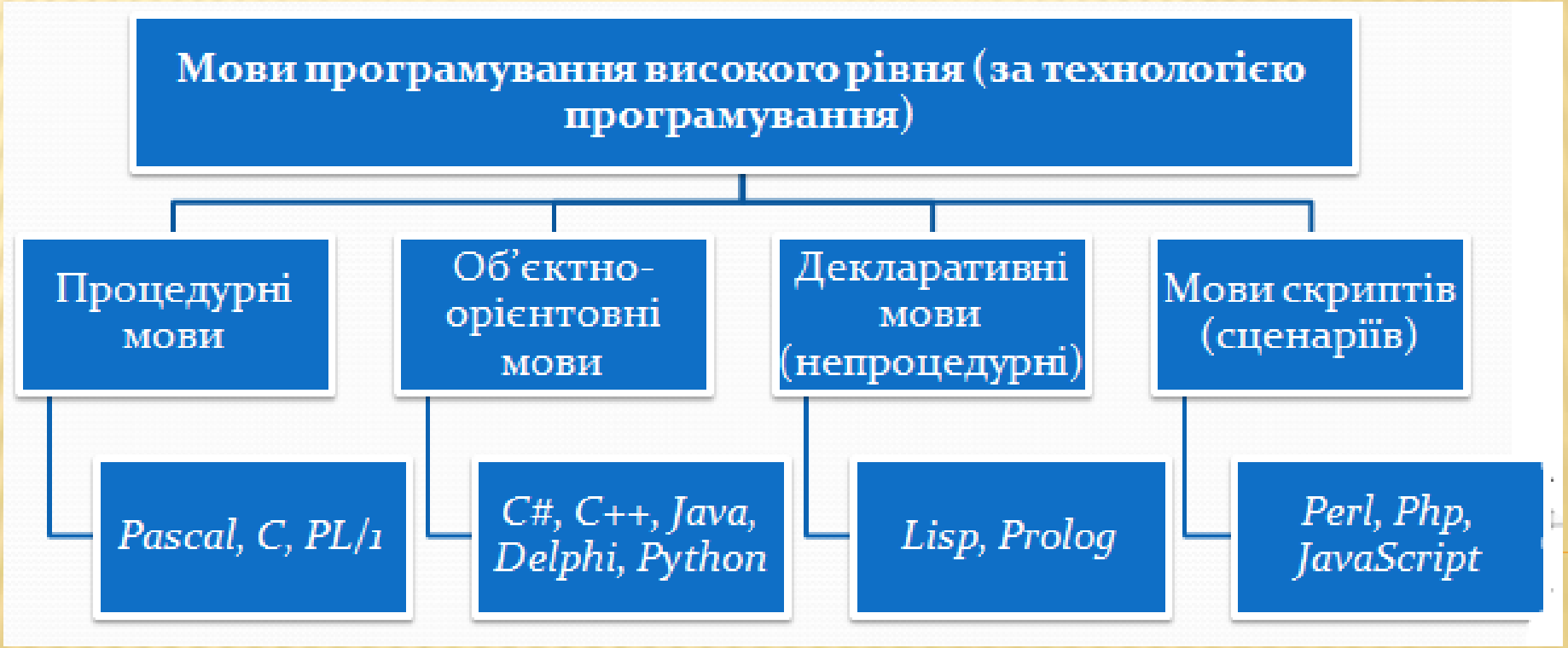
0211 – адреса для запису результату.

МОВИ ПРОГРАМУВАННЯ

В загальному випадку мови програмування поділяють на:

- мови низького рівня;

- мови високого рівня.



МОВИ ПРОГРАМУВАННЯ НИЗЬКОГО РІВНЯ

До групи мов ***низького рівня*** входять машинні мови і мови символічного кодування: Автокод, Асемблер. Оператори цієї мови – це ті ж машинні команди, але записані мнемонічними кодами, а в якості операндів використовуються не конкретні адреси, а символічні імена. Всі мови низького рівня орієнтовані на певний тип комп'ютера, отже є машинно-залежними.

Машинний код		Асемблер	
0005	B4 09	7	mov AH, 09h
0007	BA0000r	8	mov DX, offset msg
00A	CD 21	9	int 21 h

МОВА ПРОГРАМУВАННЯ C++

Загальна характеристика

C++ – універсальна мова програмування високого рівня.
Підтримує кілька парадигм:

- **процедурну**,
- **об'єктно-орієнтовану**.

Базується на мові **C**.

Історія

- Розроблена **Б'ярном Страуструпом** у **1979 р.** (AT&T Bell Laboratories, США).
- Спочатку називалася **«С з класами»**.
- У **1983 р.** перейменована на **C++**.
- Стандартизована ISO/IEC 14882:2003.

Популярність

- У 1990-х рр. стала однією з найуживаніших мов програмування загального призначення.
- Використовується для системного, прикладного та науково-технічного програмування.

МОВА ПРОГРАМУВАННЯ С++. ОСОБЛИВОСТІ

При створенні С++ прагнули зберегти сумісність з мовою С. Більшість програм на С справно працюватимуть і з компілятором С++. С++ має синтаксис, заснований на синтаксисі С.

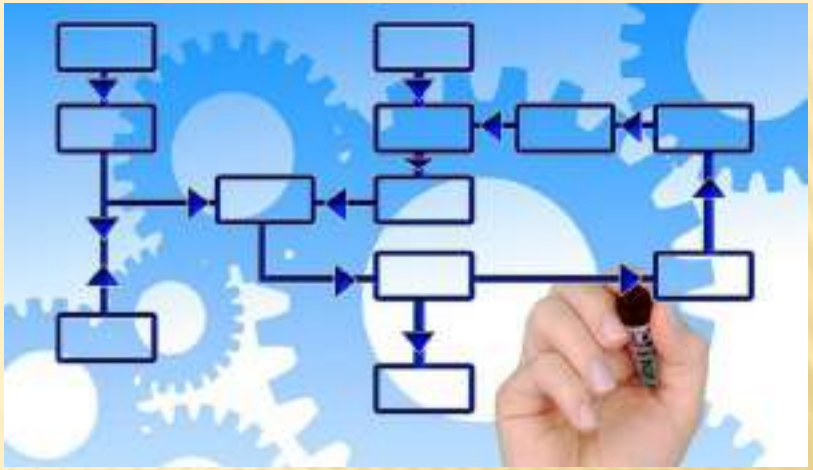
Нововведеннями С++ порівняно з С є:

1. підтримка об'єктно-орієнтованого програмування через класи;
2. підтримка узагальненого програмування через шаблони;
3. доповнення до стандартної бібліотеки;
4. додаткові типи даних;
5. обробка винятків;
6. простори імен;
7. вбудовані функції;
8. перевантаження операторів;
9. перевантаження імен функцій;
10. посилання і оператори управління вільно розподіленою пам'яттю.

У 1998 році ратифіковано міжнародний стандарт мови С++: ISO/IEC 14882 «Standard for the C++ Programming Language». Поточна версія цього стандарту — ISO/IEC 14882:2003.

Лекція №1-2

Алгоритмізація і програмувння



Лекція №3

Поняття алгоритму та правила його складання

Робота з IDE Microsoft Visual Studio

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

Поняття алгоритму так само фундаментально для інформатики, як і поняття інформації. Існує багато різних визначень алгоритму, так як це поняття досить широке і використовується в різних областях науки, техніки і повсякденному житті.

Алгори́тм (латинізов. *Algorithmi* за араб. ім'ям перського математика аль-Хорезмі) — набір інструкцій, які описують порядок дій виконавця, щоб досягти результату розв'язання задачі за скінченну кількість дій; система правил виконання дискретного процесу, яка досягає поставленої мети за скінченний час.

Виконавцем алгоритму може бути як людина (кулінарні рецепти, різні інструкції, алгоритми математичних обчислень), так і технічний пристрій. Різні машини (комп'ютери, промислові роботи, сучасна побутова техніка) є формальними виконавцями алгоритмів.

10 цікавих фактів про слово "Алгоритм":

1. Походження терміна: Слово "алгоритм" походить від імені узбецького математика Аль-Хорезмі, який жив у 9 столітті. Його латинізоване ім'я стало основою для терміна "алгоритм".

2. Історичний внесок: Аль-Хорезмі написав книгу, яка описувала процедури арифметичних обчислень за допомогою індуїстських цифр (які ми сьогодні називаємо арабськими). Це стало основою для розробки систематизованих методів розрахунків.

3. Широке застосування: Алгоритми використовуються не лише в програмуванні, але й у математиці, природничих науках, економіці, медицині, соціальних науках і навіть у повсякденному житті, наприклад, у рецептах або інструкціях.

4. Алгоритми в повсякденному житті: Навіть прості дії, як-от приготування їжі за рецептом, можуть бути описані як алгоритми. Це послідовність кроків, які потрібно виконати для досягнення певного результату.

5. Фундамент інформатики: У комп'ютерних науках алгоритми є основою будь-якого програмного забезпечення — від простих калькуляторів до складних систем штучного інтелекту.

6. Математична формалізація: Алгоритми можуть бути формально описані мовами програмування, блок-схемами або математичними виразами. Вони повинні бути детермінованими і мати чітко визначений порядок дій.

7. Важливість ефективності: Ефективність алгоритмів оцінюється за ***швидкістю*** (часом виконання) і ***використанням ресурсів*** (пам’ять, процесорні цикли). Два алгоритми для тієї ж задачі можуть мати суттєво різну продуктивність.

8. Різновиди алгоритмів: Існує безліч типів алгоритмів, таких як сортувальні алгоритми, алгоритми пошуку, алгоритми шифрування, алгоритми машинного навчання та багато інших.

9. Вплив на сучасні технології: Алгоритми лежать в основі технологій, які ми використовуємо щодня, включаючи пошукові системи, соціальні мережі, онлайн-банкінг і навіть навігацію.

10. Етичні питання: Використання алгоритмів у сучасному світі піднімає важливі етичні питання, особливо коли йдеться про алгоритми, що приймають рішення в таких сферах, як кредитування, найм працівників або рекомендаційні системи. Постає питання прозорості і справедливості цих алгоритмів.

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

Алгоритм можна записувати різними способами (словесний опис, графічне опис – блок схема, програма на одній з мов програмування і т.д.).

Програма – це алгоритм, записаний на мові програмування.

Для створення алгоритму (програми) необхідно знати:

1. -повний набір вихідних даних завдання (початковий стан об'єкта);
2. - мету створення алгоритму (кінцевий стан об'єкта);
3. - систему команд виконавця (тобто набір команд, які виконавець розуміє і може виконати).

Отриманий алгоритм (програма) повинен мати наступний набір властивостей:

- дискретність (алгоритм розбитий на окремі кроки – команди);
- однозначність (кожна команда визначає єдино можлива дія виконавця);
- зрозумілість (всі команди алгоритму входять в систему команд виконавця);
- результативність (виконавець повинен вирішити задачу за кінцеве число кроків).

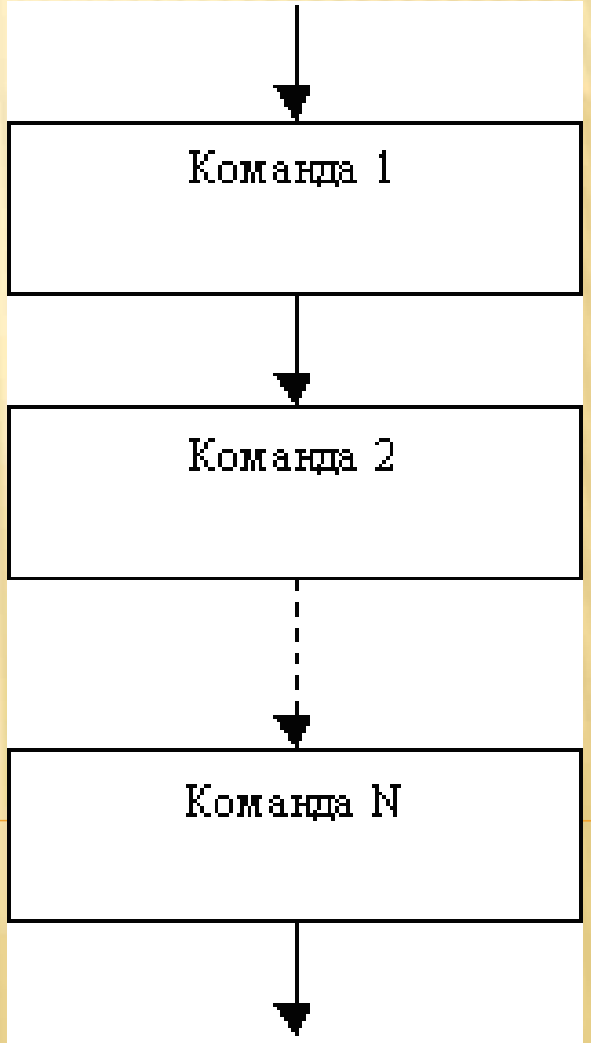
Велика частина алгоритмів має також властивість масовості або універсальності (за допомогою одного і того ж алгоритму можна вирішувати безліч однотипних завдань).

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

При складанні алгоритму керуються теоремою яка стверджує, що будь який алгоритм можна представити З типами процесів:

- Послідовним (лінійним);
- Розгалуженим;
- Циклічним

Найпростіша в написанні та виконанні перша з цих структур - лінійна. До неї відносяться алгоритми, що складаються лише з простих команд. **Простими** з точки зору комп'ютера є ті команди, що виконуються виконавцем безумовно, тобто після першої команди виконується друга, потім третя і т.д.



ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

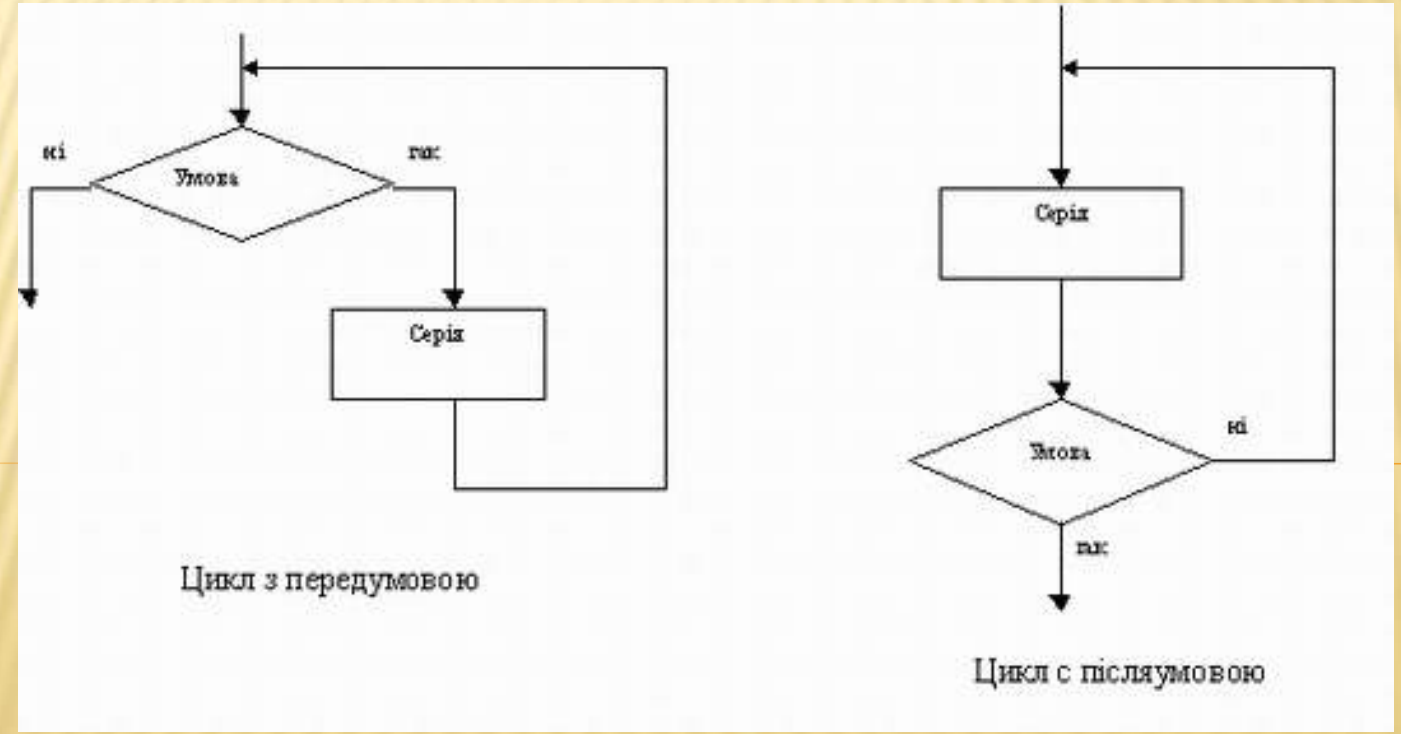
Тому набагато частіше зустрічається другий тип алгоритму - **розгалужений.** Цей алгоритм обов'язково містить в собі хоча б одну умову (як правило, їх набагато більше) і виконується він в залежності від цієї умови.Мовою блок-схем розгалужений алгоритм подається наступним чином:

В інформатиці використовують прості та складені умови. Складені умови містять кілька простих умов і об'єднуються між собою словами "або" або "та". Перше з цих слів ("або") використовується у тих випадках, коли необхідно виконання хоча б однієї з умов, тобто хоча б одна з умов являється істиною. Друге слово ("та"), навпаки, використовується лише в тих випадках, коли тільки одночасне виконання всіх умов призводить до результату.

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

Однак, навіть маючи в своєму арсеналі команду розгалуження, важко реалізувати алгоритми, що потребують багаторазового повторення деякої послідовності однакових дій. В цих випадках нас виручає циклічний алгоритм. Дуже часто зустрічаються алгоритми з повторами, причому чітко визначаються два типи повторів. :

1. ***цикл з передумовою*** - коли спочатку перевіряєnmся умова, а потім виконується деяка послідовність дій
2. ***цикл с післяумовою*** - спочатку виконується хоч один раз необхідна послідовність дій, а потім перевіряється, чи не досягнуто бажаного результату



БЛОК-СХЕМИ АЛГОРИТМІВ. ОСНОВНІ ЕЛЕМЕНТИ СХЕМ АЛГОРИТМУ

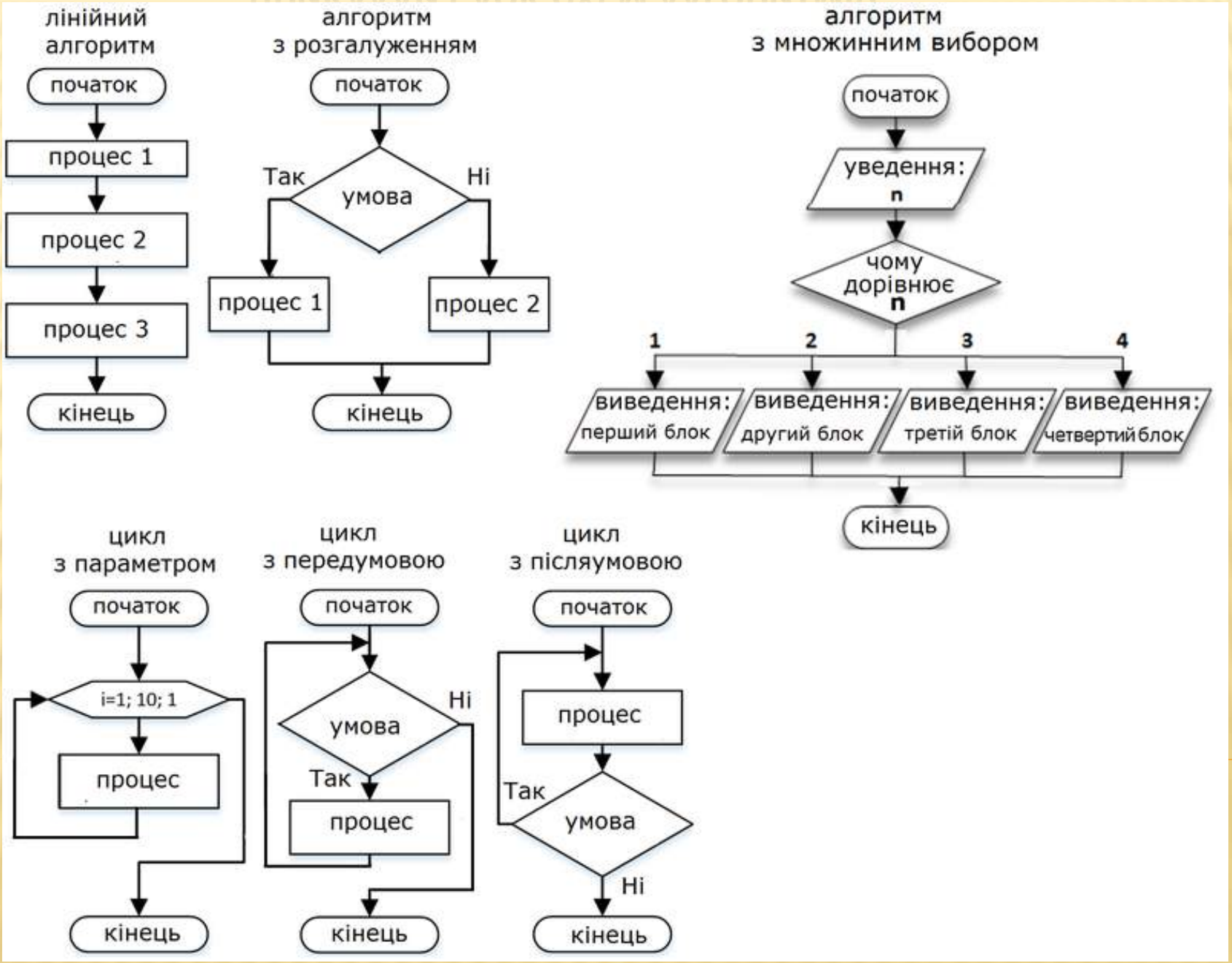
Блок схема – це графічне представлення алгоритму за допомогою стандартних позначень. Блок схеми складаються відповідно до державних стандартів і алгоритмів. На схемах алгоритмів їх дії зображуються у вигляді окремих блоків, які з'єднуються між собою лініями зв'язку в порядку виконання дій. На лініях зв'язку можуть ставитися стрілки, причому, якщо напрямок зв'язку зліва направо або зверху вниз, то стрілки не ставляться. Блоки нумеруються. Усередині блоку дається інформація про виконувані дії.

Найменування	Позначення	Функція
Початок(кінець)		Елемент відображає вхід у зовнішнє середовище або вихід з нього (найчастіше застосування - початок і кінець програми). Всередині фігури записується відповідна дія.
Процес		Елемент відображає одну або кількох операцій, обробку даних будь-якого виду (зміна значення даних, форми подання, розташування). Всередині фігури записують безпосередньо самі операції.
Умова		Елемент відображає обробку умови, рішення або функцію перемикального типу з одним входом і двома або більше альтернативними виходами, з яких тільки один може бути обраний після обчислення умов, визначених всередині цього елементу. Вхід в елемент позначається лінією, що входить зазвичай у верхню вершину елементу. Якщо виходів два чи три то зазвичай кожен вихід позначається лінією, що виходить з решти вершин (бічних і нижньої). Якщо виходів більше трьох, то їх слід показувати однією лінією, що виходить з вершини (частіше нижньої) елемента, яка потім розгалужується. Відповідні результати обчислень можуть записуватися поруч з лініями, що відображають ці шляхи.

БЛОК-СХЕМИ АЛГОРИТМІВ. ОСНОВНІ ЕЛЕМЕНТИ СХЕМ АЛГОРИТМУ

Функція (процедура)		Елемент відображає виконання процесу, що складається з однієї або кількох операцій, що визначені в іншому місці програми (у підпрограмі, модулі). Всередині символу записується назва процесу і передані в нього дані.
ввід/вивід		Елемент відображає перетворення у форму, придатну для обробки (введення) або відображення результатів обробки (виведення). Цей символ не визначає носія даних (для вказівки типу носія даних використовуються специфічні символи).
Цикл з параметром		Елемент відображає заголовок циклу з параметром. У ньому через крапку з комою вказуються ім'я змінної (параметра) з початковим значенням, граничне значення параметра (або умова виконання циклу), крок зміни параметра.
Межа циклу		Елемент складається з двох частин - відповідно, початок і кінець циклу - операції, що виконуються всередині циклу, розміщуються між ними. Умови циклу і збільшення записуються всередині символу початку або кінця циклу - в залежності від типу організації циклу. Часто для зображення на блок-схемі циклу замість цього символу використовують символ рішення, вказуючи в ньому умову, а одну з ліній виходу замикають вище в блок-схемі (перед операціями циклу).
З'єднувач		Елемент відображає вихід в частину схеми і вхід з іншої частини цієї схеми. Використовується для обриву лінії та продовження її в іншому місці (приклад: поділ блок-схеми, що не поміщається на листі). Відповідні сполучні символи повинні мати одне (при тому унікальне) позначення.
Коментар		Елемент використовується для детальнішої інформації про кроки, процесу або групи процесів. Опис поміщається з боку квадратної дужки і охоплюється нею по всій висоті. Пунктирна лінія йде до описуваного елементу, або групи елементів (при цьому група виділяється замкнутою пунктирною лінією). Також символ коментаря слід використовувати в тих випадках, коли обсяг тексту в будь-якому іншому символі (наприклад, символ процесу, символ даних та ін) перевищує його обсяг.

ПРИКЛАДИ БЛОК-СХЕМ АЛГОРИТМІВ



ПРИКЛАД 1

Приклад 1. Обчислити площу трикутника(s), якщо відомі довжина кожної сторони(a, b, c) .

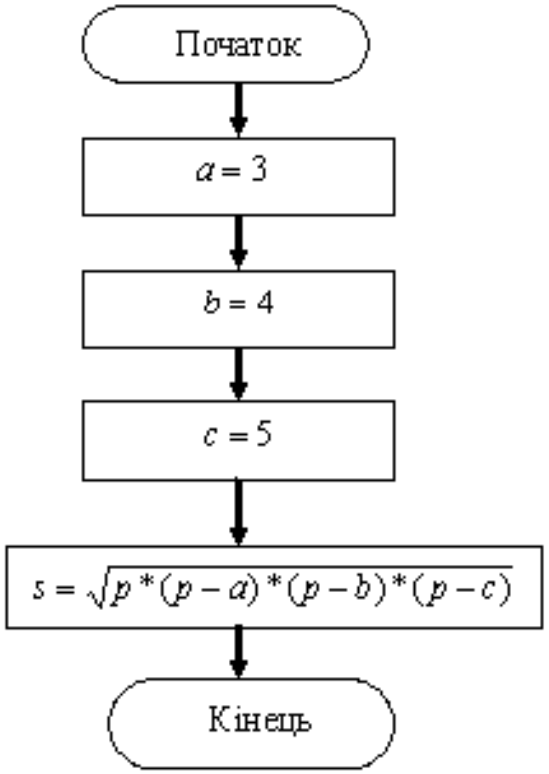


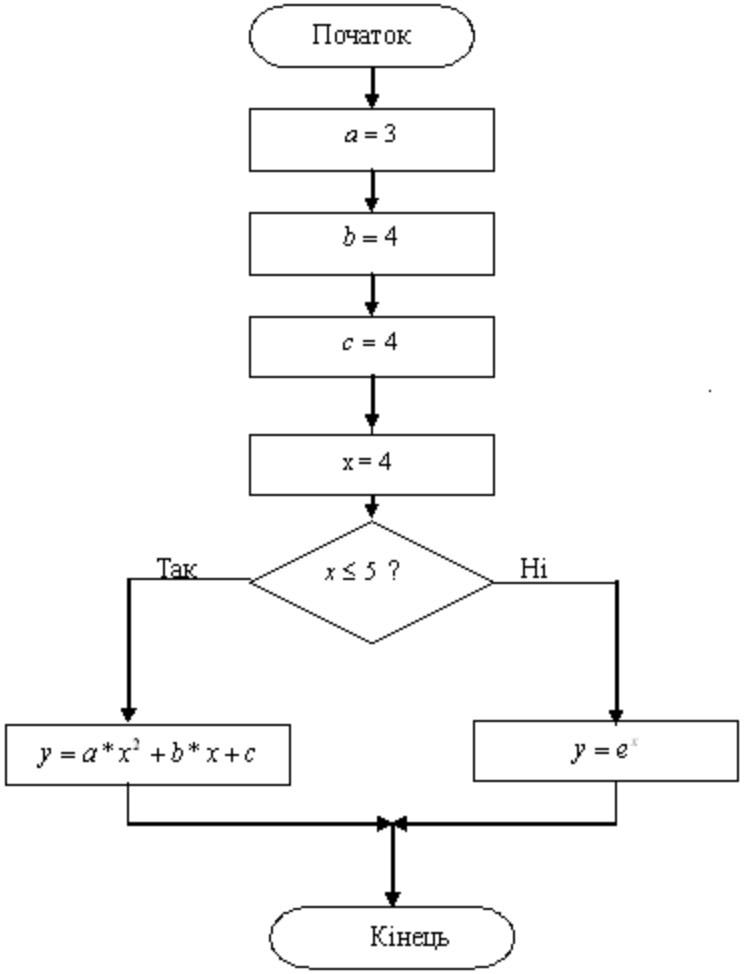
Рисунок 2 - Схема алгоритму прикладу 1

ПРИКЛАД 2

Приклвд 2. Обчислити функцію із заданими значеннями параметрів a, b, c і змінної x.

$$y = \begin{cases} ax^2 + bx + c, & x \le 5, \\ e^x, & x>5; \end{cases}$$

де значення a, b, c, x задається.



Приклад 3. Обчислити значення функції $y = ax^2 + \sin(x)$, якщо x - масив довжиною 7, значення елементів якого вводяться. Вивод здійснюється значень масиву та значення функції(y).

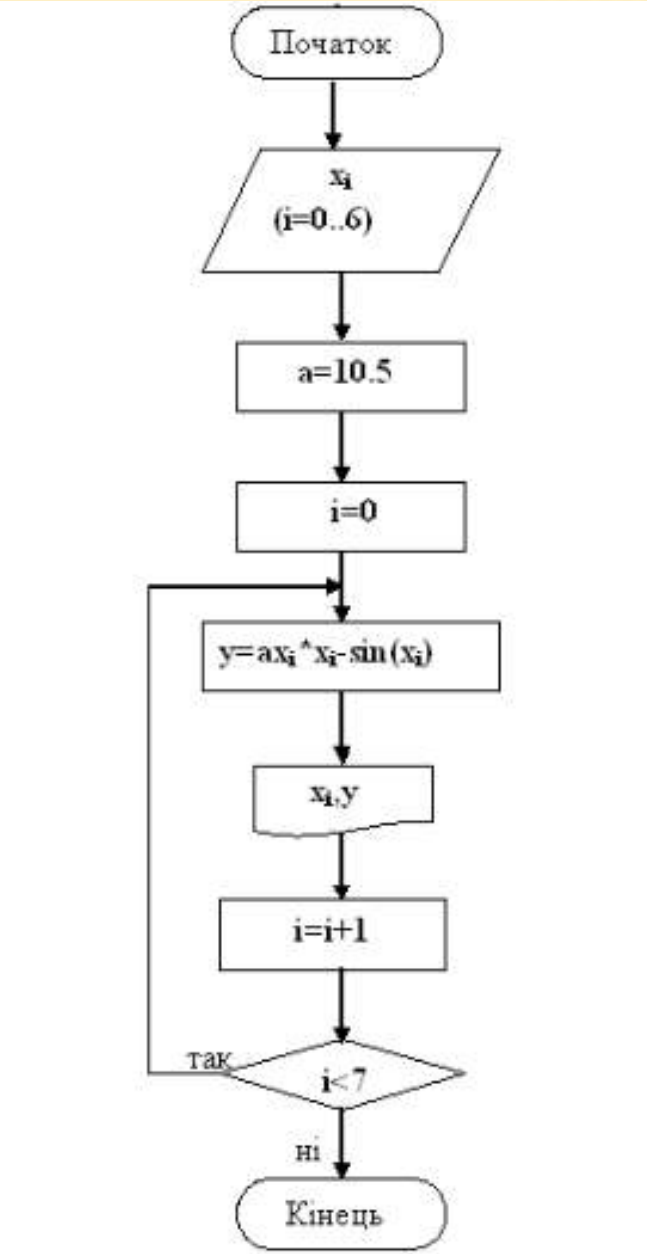


Рисунок 4 - Схема алгоритму прикладу 3

РОБОТА З IDE MICROSOFT VISUAL STUDIO

Що таке IDE?

IDE (Integrated Development Environment) — це **інтегроване середовище розробки**, яке об'єднує в собі всі необхідні інструменти для створення програмного забезпечення. Метою IDE є підвищення продуктивності розробників шляхом надання зручного та ефективного інструментарію в одному місці.

Основні компоненти IDE:

Редактор коду: Зручний текстовий редактор з підтримкою синтаксичного підсвічування, автодоповнення та перевірки синтаксису.

Компілятор/Інтерпретатор: Інструменти для перетворення вихідного коду в виконуваний файл або для його виконання в режимі інтерпретації.

Дебагер (налагоджувач): Засіб для відслідковування та виправлення помилок у коді.

Інструменти тестування: Засоби для написання та запуску тестів, забезпечення якості коду.

Графічний інтерфейс користувача (GUI) розробник: Інструменти для створення графічних інтерфейсів шляхом перетягування елементів.

.

РОБОТА З IDE MICROSOFT VISUAL STUDIO

Що таке Microsoft Visual Studio?

Microsoft Visual Studio — це **потужне інтегроване середовище розробки**, створене компанією Microsoft. Воно призначене для розробки різноманітних додатків, включаючи настільні програми, веб-додатки, мобільні додатки, ігри та хмарні сервіси.

Ключові особливості Microsoft Visual Studio:

- **Підтримка багатьох мов програмування**: C#, C++, Python, JavaScript, TypeScript, F#, Visual Basic та інші.
- **Розробка для різних платформ**: Windows, Android, iOS, Linux, macOS, а також хмарні платформи на базі Microsoft Azure.
- **IntelliSense**: Розумне автодоповнення коду, що спрощує та прискорює процес написання коду.
- **Потужні засоби налагодження та діагностики**: Включаючи можливості віддаленого налагодження, профілювання продуктивності та аналізу пам'яті.
- **Інтеграція з Azure**: Просте розгортання та керування хмарними сервісами.
- **Visual Studio Marketplace**: Доступ до тисяч розширень та плагінів для розширення функціональності IDE.
- **Командна робота**: Інтеграція з Git, GitHub, Azure DevOps для спільної розробки та управління проектами.

РОБОТА З IDE MICROSOFT VISUAL STUDIO

Переваги Microsoft Visual Studio

Універсальність та масштабованість: Visual Studio підходить для проектів будь-якої складності — від невеликих додатків до масштабних корпоративних систем.

Потужні інструменти розробки:

- **IntelliSense**: Допомагає уникнути синтаксичних помилок та швидко знаходити необхідні класи, методи та змінні.
- **Refactoring**: Інструменти для швидкого та безпечного реорганізування коду.

Засоби налагодження та тестування:

- **Гнучкі точки зупинки**: Можливість встановлювати умови та дії для точок зупинки.
- **Live Unit Testing**: Автоматичний запуск юніт-тестів у реальному часі під час внесення змін у код.

Інтеграція з хмарними сервісами:

- **Azure SDK**: Вбудовані інструменти для розробки та розгортання хмарних додатків.
- **Легка публікація**: Можливість розгортання додатків безпосередньо з IDE.

Розширюваність:

- **Visual Studio Extensions**: Підтримка великої кількості розширень для додавання нових функцій.
- **Підтримка різних фреймворків**: .NET, .NET Core, ASP.NET, Unity, Xamarin та інших.

Зручний інтерфейс користувача:

- **Налаштовувані панелі та вікна**: Можливість адаптувати робочий простір під власні потреби.
- **Темні та світлі теми**: Для комфортної роботи при різному освітленні.

Командна розробка та DevOps:

- **Інтеграція з Azure DevOps та GitHub**: Управління завданнями, збірками та релізами.
- **Керування версіями**: Зручний інтерфейс для роботи з гілками, комітами та злиттями.

Безкоштовна версія Community:

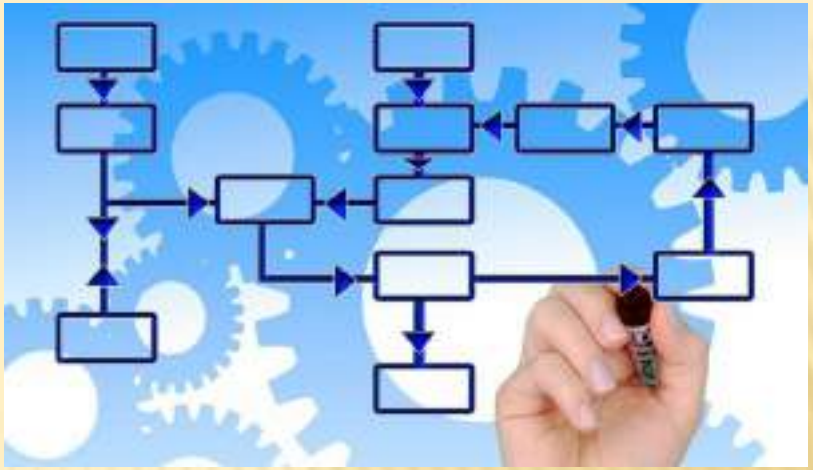
- **Доступність**: Повнофункціональна версія для індивідуальних розробників, відкритих проектів та навчальних закладів.

Підтримка та документація:

- **Офіційна документація**: Детальні керівництва та API-документація.
- **Спільнота розробників**: Форумі, блоги, відеоуроки та інші ресурси для навчання та вирішення проблем.

Постійні оновлення та інновації:

- **Сучасні технології**: Підтримка останніх версій мов програмування та фреймворків.
- **Покращення продуктивності**: Регулярні оновлення для підвищення швидкості та стабільності роботи IDE.



Лекція №3

Поняття алгоритму та правила його складання

Робота з IDE Microsoft Visual Studio

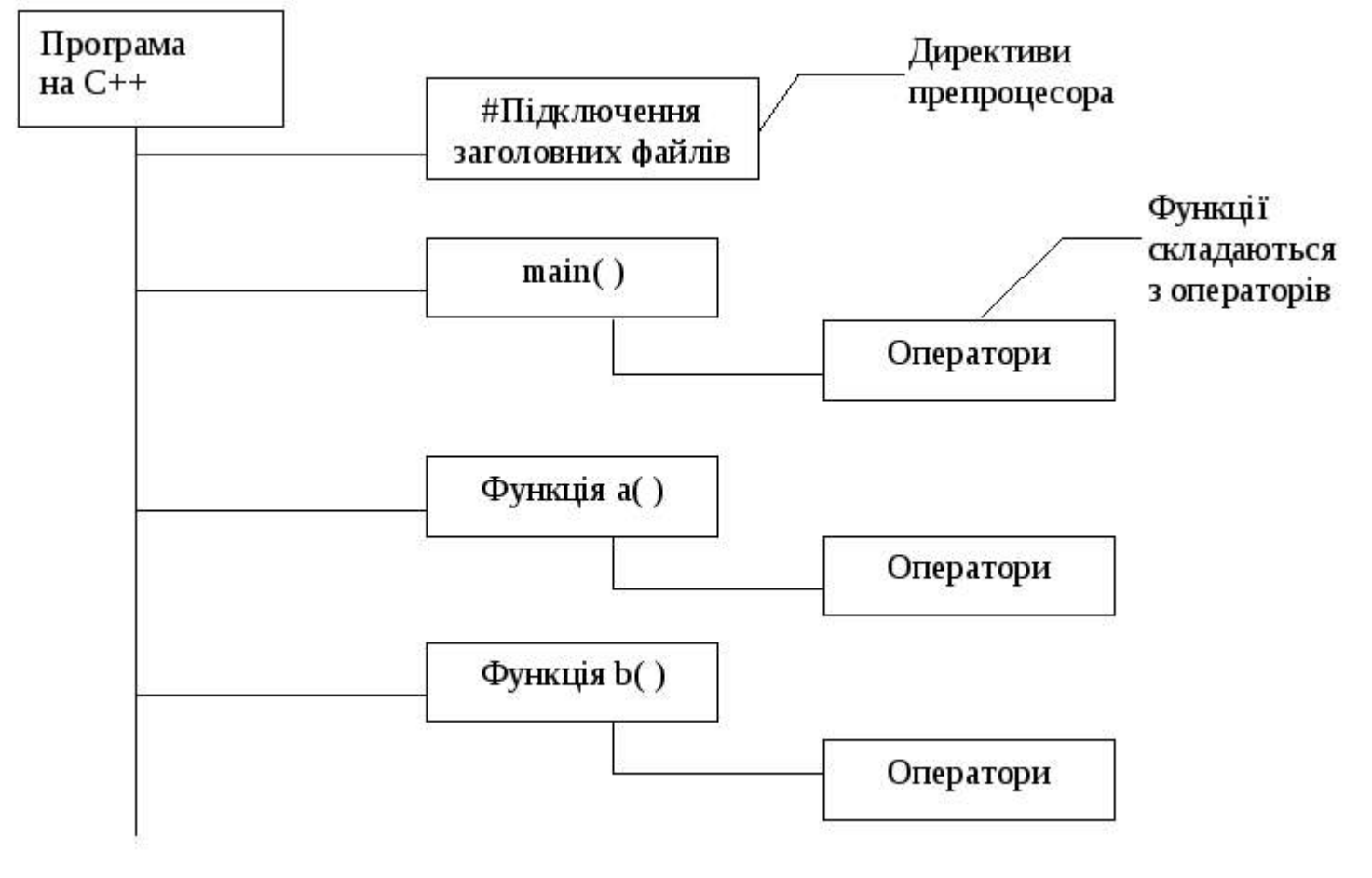
Лекція №4

Структура програми на мові С++

ЗАГАЛЬНА СТРУКТУРУ ПРОГРАМИ С++

Загальну структуру програми на мові С, С++ можна представити наступним чином:

- Директиви препроцесора: **#include, #define**
- Глобальна область: константи**, typedef/enum/struct/class**, простори імен
- Точка входу: **int main(...)**
- Визначення допоміжних функцій



ЗАГАЛЬНА СТРУКТУРА ПРОГРАМИ

```c
// Препроцесорні команди
#include <stdio.h>

// Опис глобальних змінних (необов'язково)
int global_variable = 10;

// Тіло програми
int main() {
    // Основний код програми
    printf("Значення глобальної змінної: %d\n", global_variable);
    return 0;
}
```

ЗАГАЛЬНА СТРУКТУРА ПРОГРАМИ

```cpp
#include <iostream>  // Підключення бібліотеки для вводу/виводу

using namespace std;  // Використання простору імен std

// Глобальні змінні (якщо потрібно)

void myFunction() {  // Оголошення функції
    // Код функції
}

int main() {  // Основна функція програми
    // Ініціалізація змінних
    int a = 5;
    int b = 10;

    // Виклик функцій
    myFunction();

    // Виконання операцій
    cout << "Сума: " << a + b << endl;  // Вивід результату на екран

    return 0;  // Повернення 0 означає успішне завершення програми
}
```

ЗАГАЛЬНА СТРУКТУРА ПРОГРАМИ. ПРЕПРОЦЕСОРНІ КОМАНДИ

```
#include <stdio.h> // Приклад: включення стандартної бібліотеки для вводу-вивод
#define MAX_SIZE 100 // Приклад: визначення макросу
```

Препроцесорні команди: Цей розділ містить директиви препроцесора, які виконуються перед фактичним компілюванням програми. Наприклад, включення стандартних бібліотек чи визначення макросів.

ЗАГАЛЬНА СТРУКТУРА ПРОГРАМИ. ПРЕПРОЦЕСОРНІ КОМАНДИ

```
int global_variable = 10; // Приклад: оголошення глобальної змінної
```

Опис глобальних змінних (необов'язково): У цьому розділі оголошуються глобальні змінні, які будуть доступні в усій програмі.

```
int main() {
    // Основний код програми
    printf("Hello, World!\n");
    return 0;
}
```

Тіло програми: Цей розділ містить основний код програми, який виконується у відповідності до логіки програми.

Така загальна структура дозволяє вам організувати програму та надає вам можливість визначити та використовувати функції, типи даних, змінні тощо відповідно до ваших потреб.

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

```c
#include <stdio.h>

// Оголошення функції користувача
int multiply(int a, int b) {
    return a * b;
}

void greet() {
    printf("Привіт, світе!\n");
}

// Головна функція
int main() {
    // Виклик функцій
    greet();

    int x = 5;
    int y = 3;
    int result = multiply(x, y);

    printf("%d * %d = %d\n", x, y, result);

    return 0;
}
```

Основною структурною одиницею програми на мові С, що складає тіло програми є ***функція*** – це самостійна одиниця програми, яка спроектована для реалізації конкретної підзадачі. Функція є підпрограмою, яка може міститися в основній програмі, а може бути створена окремо (в бібліотеці). Кожна функція виконує в програмі певні дії. Всі функції можна розбити на:

- стандартні функції, що входять до складу мови;
- функції користувача (підпрограми, які виконують певні дії і складаються користувачем за необхідність);
- головна функція, яка є точкою входу в програму і є обов'язковою складовою будь якої програми.

СТРУКТУРА ФУНКЦІЇ

Кожна ***функція має наступну структуру***:

- заголовок;
- опис зовнішніх змінних (необов’язково)
- тіло функції.

Заголовок (прототип): Включає в себе ім'я функції, тип повернення, та список параметрів, які приймає функція.

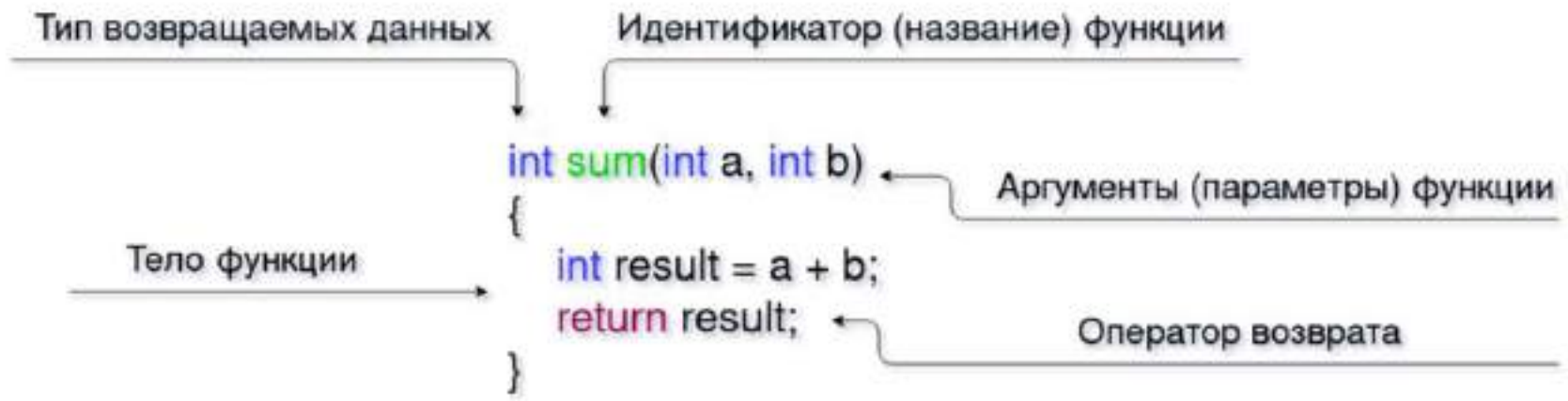
```
int multiply(int a, int b);
```

Опис зовнішніх (локальних) змінних (необов'язково): В цьому розділі оголошуються змінні, які будуть використовуватися в середині функції. Ці змінні є видимими тільки в межах цієї функції.

```
int multiply(int a, int b) {
    int result; // Локальна змінна, видима лише всередині функції
    result = a * b;
    return result;
}
```

Тіло функції: В цьому розділі розміщений сам код функції. Він виконує вказані дії.

СТРУКТУРА ФУНКЦІЇ



Заголовок функції має наступну структуру:

тип ім'я(список параметрів),

- тип – це тип значення, який функція повертає;
- ім'я – це ідентифікатор за яким до функції звертаються;
- список параметрів – перелік (через кому «,») змінних, які передаються у функцію. Список параметрів є не обов'язковим, тоді дужки є пустими.

Тіло функція представляє собою сукупність операторів та команд, які беруться у фігурні дужки (представляють собою складений оператор).

Завершується функція оператором ***return***, який повертає результат виконання функції.

Головна функція завжди має фіксоване ім'я – ***main()***.

Кожна змінна або функція (кожне ім'я), що з'являється в програмі повинно бути описане. Тому часто на початку описують всі функції, а в кінці ставлять головну функцію. Частіше, головну функцію ставлять на початку програми, а за нею описують функції (вважається, що так зручніше), але тоді перед головною функцією необхідно описати прототипи функцій.

ПОНЯТТЯ АЛГОРИТМУ ТА ПРАВИЛА ЙОГО СКЛАДАННЯ

Ім'я бібліотек, що підключаються, пишеться всередині знаків більше, менше. Заголовки і ім'я підключаємих бібліотек – це синоніми.

Синтаксис підключення заголовних файлів: #include <ім'я заголовку>

Більш старі заголовки підключаються так (цей стиль підключення бібліотек успадкований у мови програмування С):

#include <ім'я заголовку файла.h>

Різниця полягає в тому, що після імені ставиться розширення .h.

```
#include <stdio.h>
```

```
#include "my_header.h"
```

Різниця між старим та сучасним підключенням:

Старий стиль (#include <ім'я файла.h>): Цей стиль підключення бібліотек був успадкований з мови С. В ньому після імені файлу слідує розширення .h. Наприклад: #include <stdio.h>.

Сучасний стиль (#include <ім'я файла> або #include "ім'я файла"): Цей стиль дозволяє підключати файли без вказування розширення .h. Наприклад: #include <iostream> або #include "my_header.h". Це стало можливим завдяки розвитку стандартів мови та компіляторів.

ДИРЕКТИВА USING

Директива using і простір імен std:

Директива using в C++ використовується для відкриття доступу до певного простору імен, щоб використовувати його ідентифікатори без необхідності повторювати префікс з назвою простору імен. Це робить код більш читабельним та зменшує ймовірність конфлікту імен.

Наприклад, якщо у вас є простір імен std, який містить багато корисних функцій та класів, ви можете скористатися директивою using для того, щоб уникнути вказання std:: перед кожним ідентифікатором з цього простору імен.

```cpp
#include <iostream>

using namespace std; // Відкриття доступу до простору імен std

int main() {
    int x = 5;
    cout << "Значення x: " << x << endl; // Без необхідності вказувати std::
    return 0;
}
```

```cpp
#include <iostream>

int main() {
    int x = 5;
    std::cout << "Значення x: " << x << std::endl;
    return 0;
}
```

СТРУКТУРА ПРОГРАМ

Звернемо увагу на те, що кожна інструкція в мові С ++ закінчується крапкою з комою. Існують такі винятки:
- директиви препроцесора, що починаються з символу #;
- складені оператори, які обрамлені фігурними дужками.
- заголовок функції.

Для пояснення призначення структурних одиниць програми застосовуються коментарі.
Коментар це зрозуміла для програміста анотація в коді комп'ютерної програми. Існує дві нотації опису коментарів:
1) /* початок коментарю коментар може бути записаний
в декількох рядках*/ кінець коментарю
2) // коментар записаний в одному рядку.

```
#include <iostream>

int main() {
    int x = 5;  // Оголошення змінної та присвоєння їй значення 5
    double y = 3.14;  // Оголошення змінної та присвоєння їй значення 3.14

    // Вивід повідомлення на екран
    std::cout << "Значення x: " << x << ", Значення y: " << y << std::endl;

    /*
       Це багаторядковий коментар.
       Він може займати кілька рядків.
    */
    for (int i = 0; i < 10; i++) {  // Складений оператор
        std::cout << i << " ";
    }

    return 0;
```

АЛФАВІТ С, С++

Під елементами мови розуміють його базові конструкції, які використовуються при написанні програм. До них відносяться: алфавіт, правила запису констант і ідентифікаторів, основні типи даних і дії над ними.

Алфавіт Алфавітом називають сукупність символів, що використовуються в мові. В С ++ вони утворюють букви, цифри і спеціальні символи. Як букви використовуються прописні букви латинського алфавіту від *A* до *Z* та маленьку літери від *a* до *z*, а також знак підкреслювання _. Для десяткових цифр використовуються арабські цифри від 0 до 9. Спеціальні символи в мові С ++ застосовуються для різних цілей: від організації тексту програми до визначення вказівок компілятору.

- великі (**A-Z**) і малі (**a—z**) літери латинського алфавіту та символ підкреслення (_);
- арабські цифри від **0** до **9**;
- знаки арифметичних дій **+, -, *, /, %, ++, --**;
- знаки побітових операцій **<<, >>, &, |, ~, ^**;
- знаки відношень **<, <=, ==, !=, >, >=**;
- знаки логічних операцій **&&, ||, !**;
- розділові знаки **, ; : пропуск**;
- спеціальні знаки **., =, ->, ?, \, $, #, ', "**;
- символи дужок **(,), [,], {, }**.

Ключові слова

Ключові слова - це зарезервовані ідентифікатори, які мають спеціальне значення для компілятора.

auto	continue	float	interrupt	short	unsigned
asm	default	for	long	signed	void
break	do	far	near	sizeof	volatile
case	double	goto	pascal	static	while
cdecl	else	huge	switch	struct	
char	enum	if	register	typedef	
const	extern	int	return	union	

АЛФАВІТ С, С++

Константи:

Константи - це значення, які залишаються незмінними протягом виконання програми. У багатьох мовах програмування, константи оголошуються з використанням ключового слова const.

Змінні:

Змінні - це дані, які можуть змінювати свої значення в процесі виконання програми. Щоб оголосити змінну в більшості мов програмування, використовується відповідний синтаксис (наприклад, int x;).

Операнд:

Операнд - це дані чи значення, над якими виконується певна операція. У виразі a + b, a і b є операндами для операції додавання (+).

Оператор:

Оператор - це дія, яка виконується над одним чи більше операндами. Наприклад, вираз a = b + c; містить оператор присвоювання (=) та операнди b і c.

АЛФАВІТ С, С++

У цьому прикладі:

const int MAX_VALUE = 100; - оголошення константи.

int x = 5; і int y = 3; - оголошення змінних.

int sum = x + y; - використання операндів та оператора (+).

std::cout << ... ; - використання потоку виведення для виведення інформації на екран.

```
#include <iostream>

int main() {
    const int MAX_VALUE = 100;  // Константа
    int x = 5;  // Змінна
    int y = 3;  // Змінна
    int sum = x + y;  // Операнди та операція

    std::cout << "Сума " << x << " і " << y << " дорівнює " << sum << std::en

    return 0;
}
```

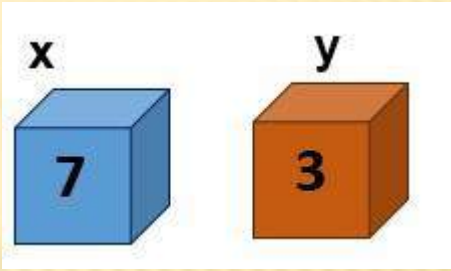
Лекція №4

Структура програми на мові С++

Структура мови C++. Змінні, оператори, модулі

Лекція №5

ЗМІННІ



Змінна — іменована область пам’яті для зберігання даних.

- ✔ Кожна змінна має:
  - ◇ **Ім’я (ідентифікатор)**
  - ◇ **Тип даних** (int, float, char, …)
- ✔ Усі змінні в C++ потрібно **оголошувати явно**.
- ✔ Синтаксис: тип_даних ім’я_змінної;
- ✔ Можна задати **початкове значення** при оголошенні.
- ✔ Рекомендація: назва змінної має відображати її призначення.


```
int age = 25;        // ✓ ціле число
float price = 99.5; // ✓ число з плаваючою крапкою
char grade = 'A';    // ✓ символ
```

У пам'яті комп'ютера змінні зберігаються у вигляді послідовностей байтів. Кожна змінна має свою **адресу** — унікальне місце в пам'яті, де вона розташована. Ця адреса вказує на осередок пам'яті, в якому зберігається її значення.

ЗМІННІ

Розташування змінних може залежати від їхнього типу та контексту використання:

1. Глобальні змінні зберігаються в окремій області пам'яті, яка називається *Data Segment*.

2. Локальні змінні (створені в межах функцій) зберігаються в *стеку* (Stack), який використовується для тимчасового зберігання даних функцій.

3. Динамічно виділені змінні (через new, malloc тощо) зберігаються в *кучі* (Heap), яка призначена для керування пам'яттю під час виконання програми.

Кожен тип змінної займає різну кількість пам'яті (наприклад, 4 байти для int або 8 байтів для double), а оператори доступу до них використовують відповідні адреси.

БАЗОВІ ТИПИ ДАНИХ

Type	Size	Description	Range
char	1 byte	small integer	-128 to 127
unsigned char	1 byte	unsigned small integer	0 to 255
short int	2 bytes	Short Integer	-32768 to 32767
unsigned short int	2 bytes	unsigned Short Integer	0 to 65535
int	4 bytes	Integer	-2147483648 to 2147483647
unsigned int	4 bytes	unsigned Integer	0 to 4294967295
bool	1 byte	Boolean value	true or false
float	4 bytes	Floating point number	$\pm 3.4 \cdot 10^{\pm 38}$ (7 digits)
double	8 bytes	Double floating point	$\pm 1.7 \cdot 10^{\pm 308}$ (15 digits)

- **int** — цілі числа (наприклад, -10, 0, 25)
- **float** — числа з плаваючою крапкою одинарної точності (наприклад, 3.14)
- **double** — числа з плаваючою крапкою подвійної точності (наприклад, 2.7182818)
- **char** — окремий символ ('A', 'b', '1')
- **bool** — логічний тип (true / false)

БАЗОВІ ТИПИ ДАНИХ

```
// Оголошення змінних без значень
int age;
float weight;
double pi;
char grade;
bool isReady;

// Присвоєння значень після оголошення
age = 30;           // ✓ цілі числа
weight = 65.5;      // ✓ дійсне число
pi = 3.14159;       // ✓ подвійна точність
grade = 'A';        // ✓ символ
isReady = true;     // ✓ логічне значення
```

```cpp
int num; // Цілочисельний тип (може бути зі знаком)
float f_num; // Тип даних для чисел з плаваючою точкою менше (мінімально) ні
double d_num; // Тип даних для чисел з плаваючою точкою більше (максимально)
bool is_true; // Тип для логічних значень (true або false)
char letter; // Тип для представлення одного символу в ASCII коду
string msg; // Рядковий тип даних
const int const_val; // Константи - значення, яке не можна змінити
int* ptr; // Вказівник - змінна, яка містить адресу в пам'яті
int (*func_ptr)(int, int); // Вказівник на функцію - змінна, яка містить ад
int nums[5]; // Масив - зберігає кілька значень одного типу

// Присвоєння значень
num = 10;
f_num = 3.14;
d_num = 2.71828;
is_true = true;
letter = 'A';
msg = "Hello, World!";
const_val = 42; // Константа повинна бути проініціалізована відразу
ptr = &num;
func_ptr = nullptr; // Або присвоюємо адресу функції
nums[0] = 1;
nums[1] = 2;
nums[2] = 3;
nums[3] = 4;
nums[4] = 5;
```

БАЗОВІ ТИПИ ДАНИХ

ТИП CHAR

У C++ змінні типу char здебільшого застосовуються для роботи з окремими символами.

- За замовчуванням char, int, short int, long – **зі знаком (signed)**
- Ключове слово signed можна не вказувати
- Тип char зберігає **один символ** (буква, цифра, знак) → займає **1 байт пам'яті**
- char може бути:
  - **signed char** → діапазон: **-128 … +127**
  - **unsigned char** → діапазон: **0 … 255**
- Використовується для відображення **256 символів ASCII**

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

int main() {
    char grade = 'A'; // Оголошення та ініціалізація змінної char

    cout << "Your grade is: " << grade << endl; // Виведення значення змінної

    return 0;
}
```

Dec	Hex	Name	Char	Ctrl-char	Dec	Hex	Char	Dec	Hex	Char	Dec	Hex	Char
0	0	Null	NUL	CTRL-@	32	20	Space	64	40	@	96	60	`
1	1	Start of heading	SOH	CTRL-A	33	21	!	65	41	A	97	61	a
2	2	Start of text	STX	CTRL-B	34	22	"	66	42	B	98	62	b
3	3	End of text	ETX	CTRL-C	35	23	#	67	43	C	99	63	c
4	4	End of xmit	EOT	CTRL-D	36	24	$	68	44	D	100	64	d
5	5	Enquiry	ENQ	CTRL-E	37	25	%	69	45	E	101	65	e
6	6	Acknowledge	ACK	CTRL-F	38	26	&	70	46	F	102	66	f
7	7	Bell	BEL	CTRL-G	39	27	'	71	47	G	103	67	g
8	8	Backspace	BS	CTRL-H	40	28	(	72	48	H	104	68	h
9	9	Horizontal tab	HT	CTRL-I	41	29	)	73	49	I	105	69	i
10	0A	Line feed	LF	CTRL-J	42	2A	*	74	4A	J	106	6A	j
11	0B	Vertical tab	VT	CTRL-K	43	2B	+	75	4B	K	107	6B	k
12	0C	Form feed	FF	CTRL-L	44	2C	,	76	4C	L	108	6C	l
13	0D	Carriage feed	CR	CTRL-M	45	2D	-	77	4D	M	109	6D	m
14	0E	Shift out	SO	CTRL-N	46	2E	.	78	4E	N	110	6E	n
15	0F	Shift in	SI	CTRL-O	47	2F	/	79	4F	O	111	6F	o
16	10	Data line escape	DLE	CTRL-P	48	30	0	80	50	P	112	70	p
17	11	Device control 1	DC1	CTRL-Q	49	31	1	81	51	Q	113	71	q
18	12	Device control 2	DC2	CTRL-R	50	32	2	82	52	R	114	72	r
19	13	Device control 3	DC3	CTRL-S	51	33	3	83	53	S	115	73	s
20	14	Device control 4	DC4	CTRL-T	52	34	4	84	54	T	116	74	t
21	15	Neg acknowledge	NAK	CTRL-U	53	35	5	85	55	U	117	75	u
22	16	Synchronous idle	SYN	CTRL-V	54	36	6	86	56	V	118	76	v
23	17	End of xmit block	ETB	CTRL-W	55	37	7	87	57	W	119	77	w
24	18	Cancel	CAN	CTRL-X	56	38	8	88	58	X	120	78	x
25	19	End of medium	EM	CTRL-Y	57	39	9	89	59	Y	121	79	y
26	1A	Substitute	SUB	CTRL-Z	58	3A	:	90	5A	Z	122	7A	z
27	1B	Escape	ESC	CTRL-[	59	3B	;	91	5B	[	123	7B	{
28	1C	File separator	FS	CTRL-\	60	3C	<	92	5C	\	124	7C	|
29	1D	Group separator	GS	CTRL-]	61	3D	=	93	5D	]	125	7D	}
30	1E	Record separator	RS	CTRL-^	62	3E	>	94	5E	^	126	7E	~
31	1F	Unit separator	US	CTRL-_	63	3F	?	95	5F	_	127	7F	DEL

ТИП CHAR

ПРИКЛАД РОБОТИ З ТИПАМИ INT, FLOAT, BOOL І STRING В MOBI C++

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

int main() {
    // Робота з int
    int num = 10; // Оголошення та ініціалізація змінної int
    cout << "Integer number: " << num << endl;

    // Робота з float
    float f_num = 3.14; // Оголошення та ініціалізація змінної float
    cout << "Float number: " << f_num << endl;

    // Робота з bool
    bool is_true = true; // Оголошення та ініціалізація змінної bool
    cout << "Boolean value: " << is_true << endl;

    // Робота з string
    string message = "Hello, World!"; // Оголошення та ініціалізація змінної st
    cout << "String message: " << message << endl;

    return 0;
}
```

- ✔ **int** — цілі числа
- ✔ **float** — числа з плаваючою крапкою (обмежена точність)
- ✔ **bool** — логічні значення (true / false)
- ✔ **string** (клас std::string) – послідовність символів

```
Integer number: 10
Float number: 3.14
Boolean value: 1
String message: Hello, World!
```

ТИП VOID

Void – це **спеціальний тип**, що означає *«відсутність значення»*.

Використовується у таких випадках:

- **Функції без повернення результату**

```
void printHello() {        cout << "Hello, world!";        }
```

- **Порожні параметри функцій** (коли функція не приймає аргументів)

```
int getNumber(void);
```

- **Вказівники на невизначений тип**

```
void* ptr; // універсальний вказівник
```

ОПИС ЗМІННИХ І КОНСТАНТ

✔ **Змінна** — область пам’яті з унікальним ім’ям, значення якої можна змінювати під час виконання програми.

✔ **Константа** — область пам’яті з унікальним ім’ям, значення якої **не можна змінити** після оголошення.

Для опису констант використовують ключове слово **const** за наступним шаблоном:

const [модифікатор] тип ідентифікатор=значення;

```
int age = 20;                    // ✓ змінна
float price;                     // ✓ змінна без значення
const double PI = 3.14159;       // ✓ константа (не змінюється)
```

ПРАВИЛА ПЕРЕТВОРЕННЯ ТИПІВ

✔ Якщо операнди мають різні типи → вони перетворюються до **спільного типу**.

✔ Перетворення відбуваються лише у напрямку **розширення діапазону значень**.

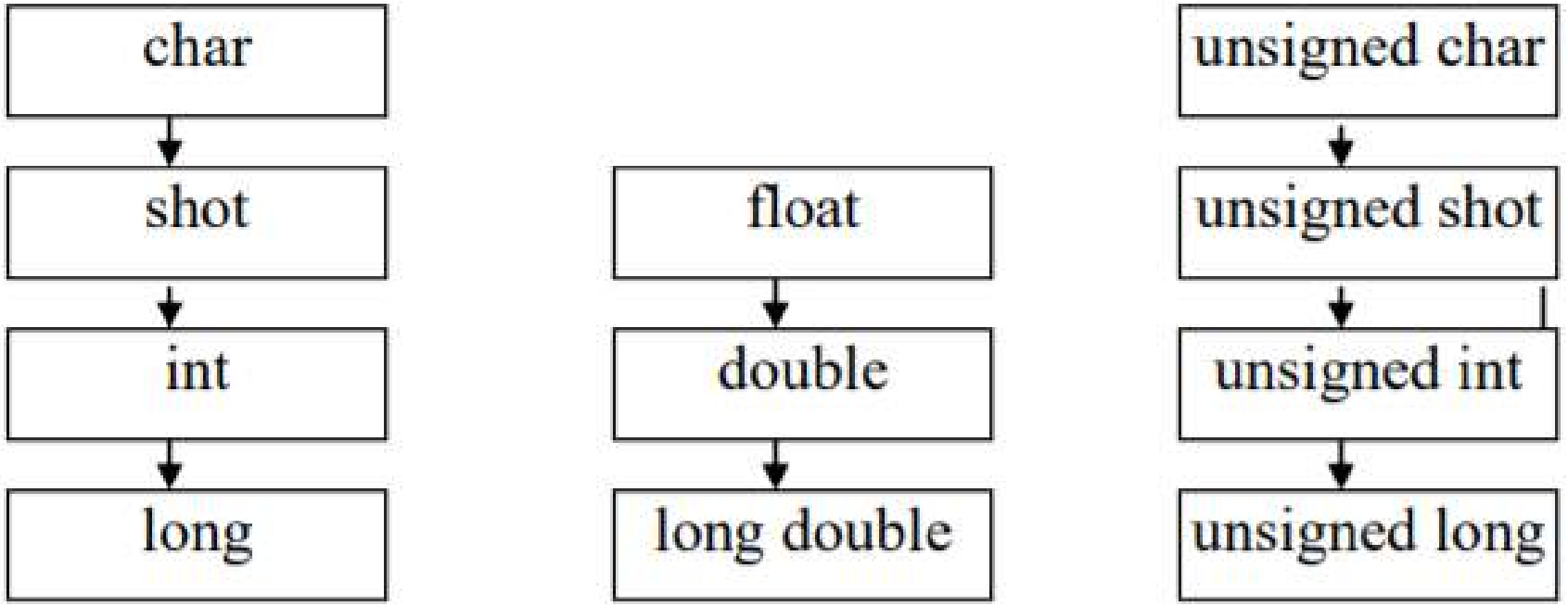
Правила приведення:

1. char, short int → автоматично у int

1. float → автоматично у double
2. Якщо один операнд має **long double** → інший також стає long double
3. Інакше, якщо один має **double** → інший стає double
4. Інакше, якщо один має **long** → інший стає long
5. Інакше, якщо один має **unsigned** → інший стає unsigned

✔ У результаті операція завжди повертає значення того ж типу, у який були приведені операнди.

ПРАВИЛА ПЕРЕТВОРЕННЯ ТИПІВ



Але існують ситуації, де необхідно бути дуже уважним до перетворення типів. Наприклад, при діленні 2-х цілих чисел результат також буде цілим числом, що в загальному випадку є невірним.

ПРАВИЛА ПЕРЕТВОРЕННЯ ТИПІВ

Наряду з явним виначенням типів в мові С передбачено неявне приведення типу: **тип**(змінна).

```
int num1 = 10;
double num2 = 3.14;

// Явне приведення типу (casting)
int result = (int)num2; // Результат буде 3
```

ОПЕРАЦІЇ ЗІ ЗМІННИМИ

Знаки операцій	Назва, пояснення	Асоціативність
`() [] . ->`	Первинні	Зліва направо

`+ - ~ ! * & ++ --`
`sizeof(тип)`
приведення типу
	Унарні	Справа наліво

`* / %`	Мультиплікативні, арифметичні, бінарні	Зліва направо
`+ -`	Адитивні, арифметичні, бінарні	Зліва направо
`>> <<`	Зсув	Зліва направо
`< > <= >=`	Відношення	Зліва направо
`== !=`	Відношення	Зліва направо
`&`	Порозрядне "І", логічна, бінарна	Зліва направо
`^`	Порозрядне виключне "АБО", логічна, бінарна	Зліва направо
`|`	Порозрядне "АБО", логічна, бінарна	Зліва направо
`&&`	Логічне "І", бінарна	Зліва направо
`||`	Логічне "АБО", бінарна	Зліва направо
`?:`	Умовна, тернарна	Справа наліво
`= *= /= %= += -=` `<<= >>= &= |= ^=`	Просте і складне присвоювання	Справа наліво
`,`	Послідовне обчислення	Зліва направо

```cpp
int main() {
    // Арифметичні операції з int
    int num1 = 10;
    int num2 = 5;

    int sum = num1 + num2; // Додавання
    int diff = num1 - num2; // Віднімання
    int product = num1 * num2; // Множення
    int quotient = num1 / num2; // Ділення
    int remainder = num1 % num2; // Остача від ділення

    cout << "Sum: " << sum << endl;
    cout << "Difference: " << diff << endl;
    cout << "Product: " << product << endl;
    cout << "Quotient: " << quotient << endl;
    cout << "Remainder: " << remainder << endl;

    // Операції з char
    char char1 = 'A';
    char char2 = 'B';

    char concatenatedChars = char1 + char2; // Конкатенація char

    cout << "Concatenated chars: " << concatenatedChars << endl;

    // Операції з string
    string str1 = "Hello, ";
    string str2 = "World!";

    string concatenatedStrings = str1 + str2; // Конкатенація string

    cout << "Concatenated strings: " << concatenatedStrings << endl;
```

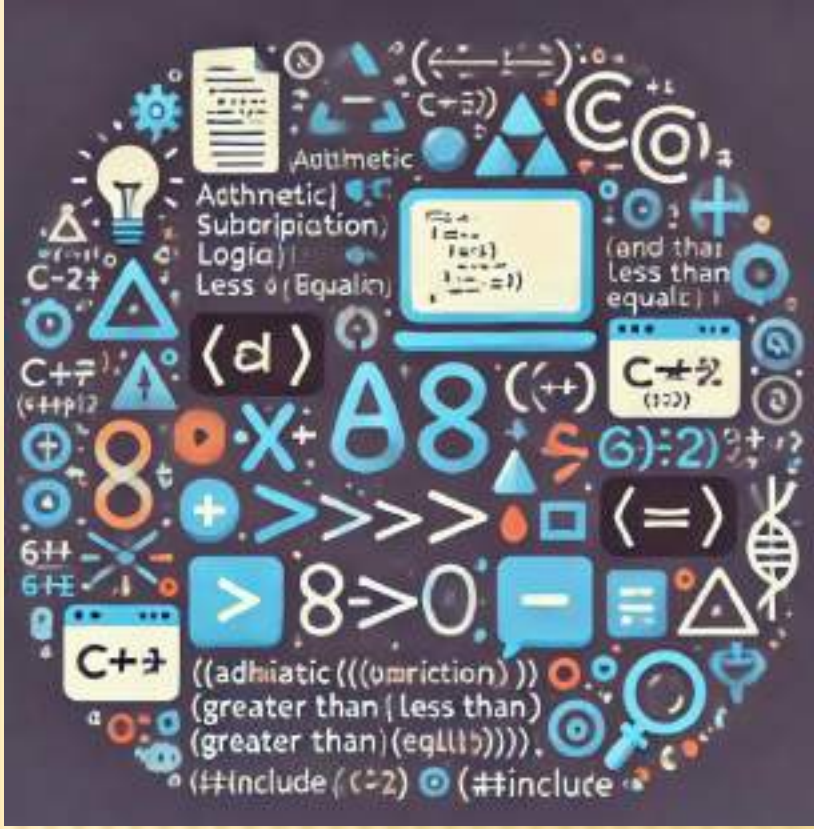
ОПЕРАЦІЇ ЗІ ЗМІННИМИ

Структура мови C++. Змінні, оператори, модулі

Лекція №5

Лекція №6

Операції мови програмування С++



ОПЕРАЦІЇ МОВИ ПРОГРАМУВАННЯ С++

Операції – це дії над даними. Дані, які беруть участь в операції, часто називають операндами. Як операнда може виступати константа, змінна або виклик будь-якої функції. Для кожної операції використовується відповідний їй знак операції, що складається з одного або декількох символів. В результаті виконання операцій завжди виходить якесь значення, що представляє собою результат виконання операції.

У мові С ++ існує велика кількість різноманітних операції. Їх можна класифікувати за різними ознаками, наприклад: за кількістю операндів, за призначенням, або якось ще.

Залежно від кількості операндів в С ++ є одномісні, двомісні і одна тримісна операція.

ОПЕРАЦІЇ РАНГУ 1

Ранг	Операції
1	:: . -> ( ) [ ]
2	! ~ + - ++ -- & * (тип) sizeof new delete тип ( ) typeid dynamic_cast static_cast reinterpret_cast const_cast
3	.* ->*
4	* / % (мультиплікативні)
5	+ – (аддитивні)
6	<< >> (зсув)
7	< <= >= > (порівняння)
8	== != (порівняння)
9	& (поразрядна кон’юнкція І)
10	^ (поразрядне виключаюче АБО)
11	| (поразрядна диз’юнкція АБО)
12	&& (логічна кон’юнкція І)
13	|| (логічна диз’юнкція АБО)
14	? : (умовна операція)
15	= *= /= %= += -= &= ^= |= <<= >>= операція присвоювання
16	throw
17	, (операція кома)

ПРІОРИТЕТ ОПЕРАЦІЙ

📌 **Основне:**

- Пріоритет: `+` і `-` мають нижчий пріоритет, ніж `*`, `/`, `%`.
- Для зміни порядку обчислення застосовують **круглі дужки**.
- Приклад: `2 * (A + B)` — спочатку обчислюється сума, потім множення.

В мові програмування С++ операція ділення (/) поводиться по-різному залежно від типу операндів: Якщо один або обидва операнда – дійсні, то виконується традиційне ділення, якщо обидва операнди – цілі, то виконується ділення без остачі і результат буде цілого типу. Використання цієї операції вимагає підвищеної уважності, наприклад, якщо запрограмувати обчислення математичного виразу як 1/3 * sin (2 * X) то результат незалежно від значення X завжди буде дорівнює нулю, так як вираз 1/3 означає ділення без остачі. Для вирішення проблеми достатньо один з операндів зробити дійсним:

1.0 / 3 * sin (2 * X)

Операція обчислення залишку (%) може бути застосована тільки для цілочисельних операндів.

```
int a = 1;
int b = 3;
double result = a / b; // Результат буде 0
```

ПРІОРИТЕТ ОПЕРАЦІЙ

⚠ **Ділення:**

- Якщо хоча б один операнд дійсний → виконується **звичайне ділення** (результат дробовий).
- Якщо обидва операнди цілі → виконується **ділення без остачі** (результат цілий).
- Приклад помилки:

Дійсне ділення: Якщо один або обидва операнди — дійсні числа (float, double), то результат буде дійсним числом. Наприклад:

```
double a = 1.0;
int b = 3;
double result = a / b; // Результат буде 0.333...
```

ПРІОРИТЕТ ОПЕРАЦІЙ

- Унарний мінус - змінює знак операнда.
  Приклад: -5 → результат **–5**.
- Має **високий пріоритет** серед операторів.
- Виконується раніше, ніж множення (*) чи додавання (+).
- Може застосовуватись як до констант, так і до змінних.

```
int a = 3;
int b = -a;   // b = -3
int c = -a * 2; // c = -6, мінус спрацював першим
```

ІНКРЕМЕНТ ТА ДЕКРЕМЕНТ У C++

Інкремент (++) збільшує значення змінної на 1.

Декремент (--) зменшує значення змінної на 1.

Форми операторів:

- **Префіксна** (++a, --a) — спочатку змінює значення, потім використовує у виразі.
- **Постфіксна** (a++, a--) — спочатку використовує у виразі, потім змінює значення.

```
int a = 5;


int b = ++a;  // префікс: a = 6, b = 6
int c = a--;  // постфікс: c = 6, a = 5
```

ІНКРЕМЕНТ ТА ДЕКРЕМЕНТ У C++

```cpp
int A = 5;
int B;

// Префіксна форма (++A, --A)
B = ++A; // Спочатку збільшується A, потім B = 6
cout << "Префіксний інкремент (++A): A = " << A << ", B = " << B << endl;

B = --A; // Спочатку зменшується A, потім B = 5
cout << "Префіксний декремент (--A): A = " << A << ", B = " << B << endl;

// Постфіксна форма (A++, A--)
B = A++; // Спочатку B = 5, потім A збільшується до 6
cout << "Постфіксний інкремент (A++): A = " << A << ", B = " << B << endl;

B = A--; // Спочатку B = 6, потім A зменшується до 5
cout << "Постфіксний декремент (A--): A = " << A << ", B = " << B << endl;
```

ПРІОРИТЕТ ОПЕРАЦІЙ

Операції	*Значення змінної A*	*Значення змінної B*
A=1	1	
B=A++ B=A−−	2 0	1 1

Префіксная форма запису дає такий результат:

Операції	*Значення змінної A*	*Значення змінної B*
A=1	1	
B=++A B=−−A	2 0	2 0

Найчастіше операції використовують в операторах циклу.

Стандартні математичні функції

Функція C++	Опис	Клас Math
int **abs**(int i)	модуль (абсолютне значення) цілого числа x – $\|x\|$	Math::**Abs**(x)
double **fabs**(double x)	модуль дійсного числа x – $\|x\|$	Math::**Abs**(x)
double **sqrt**(double x)	корінь квадратний – $\sqrt{x}$	Math::**Sqrt**(x)
double **pow**(double x, double y)	піднесення x до степеня y – x^y	Math::**Pow**(x,y)
double **exp**(double x)	експонента e^x	Math::**Exp**(x)
double **log**(double x)	натуральний логарифм – $\ln(x)$	Math::**Log**(x)
double **log10**(double x)	десятковий логарифм – $\lg(x)$	Math::**Log10**(x)
–	логарифм x за основою N – $\log_N(x)$	Math::**Log**(x,N)
double **cos**(double x)	косинус – $\cos(x)$	Math::**Cos**(x)
double **sin**(double x)	синус – $\sin(x)$	Math::**Sin**(x)
double **tan**(double x)	тангенс – $\mathrm{tg}(x)$	Math::**Tan**(x)
double **acos**(double x)	арккосинус – $\arccos(x)$	Math::**Acos**(x)
double **asin**(double x)	арксинус – $\arcsin(x)$	Math::**Asin**(x)
double **atan**(double x)	арктангенс – $\mathrm{arctg}\,(x)$	Math::**Atan**(x)
double **cosh**(double x)	гіперболічний косинус – $(e^x+e^{-x})/2$	Math::**Cosh**(x)
double **sinh**(double x)	гіперболічний синус – $(e^x-e^{-x})/2$	Math::**Sinh**(x)
double **ceil**(double x)	округлення доверху: найменше ціле, не менше за x	Math::**Ceiling**(x)
double **floor**(double x)	округлення донизу: найбільше ціле, не більше за x	Math::**Floor**(x)

Стандартні математичні функції в мові С описані в заготовочному файлі **cmath.**h. Отже для їх використання на початку програми необхідно підключити цей файл за допомогою директиви **#include**.

Дамо перелік основних функцій: Необхідно запам'ятати те, що операнди даних функцій завжди повинні бути дійсними, тобто *a* і *b* числа з плаваючою крапкою.

Операції порівняння та логічні операції в мові С++

Призначення:

- Використовуються для порівняння числової 🔢 або символьної 🔤 інформації.
- Результат завжди ✅ **true** або ❌ **false**.
- Пріоритет нижчий, ніж у арифметичних операцій ➕✖️.

Операції порівняння застосовуються для порівняння (зіставлення) числової або символьної інформації. Результатом виконання операцій є або істина (*true*), або брехня (*false*). обставину.

Оператори порівняння:

- `==` → рівність ⚖️
- `!=` → нерівність 🚫
- `>` → більше ⬆️
- `<` → менше ⬇️
- `>=` → більше або дорівнює ➡️
- `<=` → менше або дорівнює ⬅️

Дійсні числа в пам'яті зберігаються приблизно, при роботі з ними в ході виконання арифметичних операцій можуть накопичуватися похибки, тому не рекомендується виконувати перевірку на рівність або на нерівність для дійсних чисел, тому що результат може бути непередбачуваний.

Якщо для дійсних чисел треба виконати перевірку на рівність (типу A == B), то замінюємо її на перевірку на нерівність виду

| A-B | <= ε, де ε – деяка мала позитивна величина епсилон.

У термінах мови C ++ це буде виглядати так

fabs (A-B) <= eps

```cpp
#include <iostream>
#include <cmath> // Підключаємо бібліотеку для використання fabs()

int main() {
    double A = 0.1 * 3; // Результат має бути 0.3
    double B = 0.3;     // Очікуване значення

    // Неправильна перевірка на рівність
    if (A == B) {
        std::cout << "A дорівнює B (==) - Неправильно!" << std::endl;
    } else {
        std::cout << "A не дорівнює B (==)" << std::endl;
    }

    // Правильна перевірка з використанням епсилон
    double eps = 1e-9; // Невелике число для врахування похибки
    if (fabs(A - B) <= eps) {
        std::cout << "A дорівнює B з точністю до eps" << std::endl;
    } else {
        std::cout << "A не дорівнює B з точністю до eps" << std::endl;
    }
```

В мові С допускається множинне присвоєння: a=b=c=d;

Якщо тип правого операнда не збігається з типом лівого, то значення праворуч перетвориться до типу лівого операнда (якщо це можливо). При множинному присвоюванні краще дотримуватись правила зазначеного вище що старшому типу можна присвоювати молодший, а не навпаки.

Ще цікавою особливість мови С є так звана комбінована операція присвоювання виду:

a оп =b

де ***оп*** знак однієї з бінарних операцій: + - * / % >> << & | ^ && ||.

Присвоєння ***a оп = b*** еквівалентно ***a = a оп b***.

Результатом операції складеного присвоювання є значення і тип лівого операнда. наприклад:

a += b; ***//***a = a + b;

a -=b; //a = a - b;

a *= b*; //a = a * b;

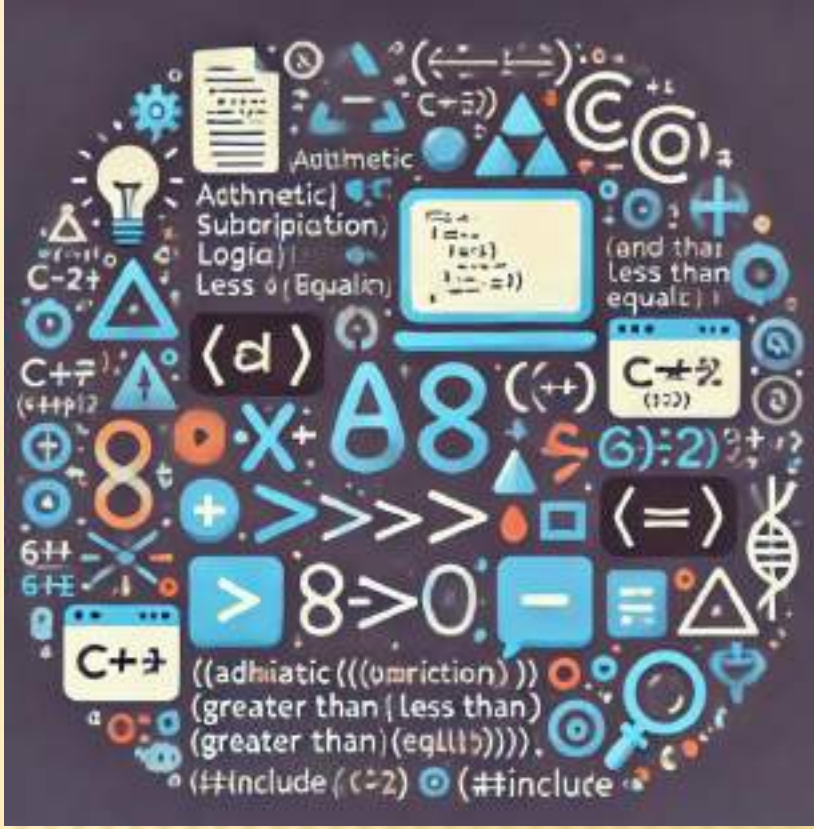
МНОЖИННЕ ТА КОМБІНОВАНЕ ПРИСВОЮВАННЯ

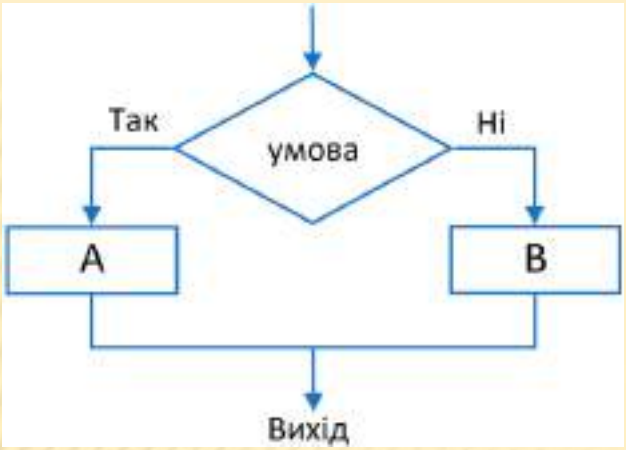
```
int a, b, c, d;

a = b = c = d = 10;

// Тепер a, b, c, і d всі мають значення 10
```

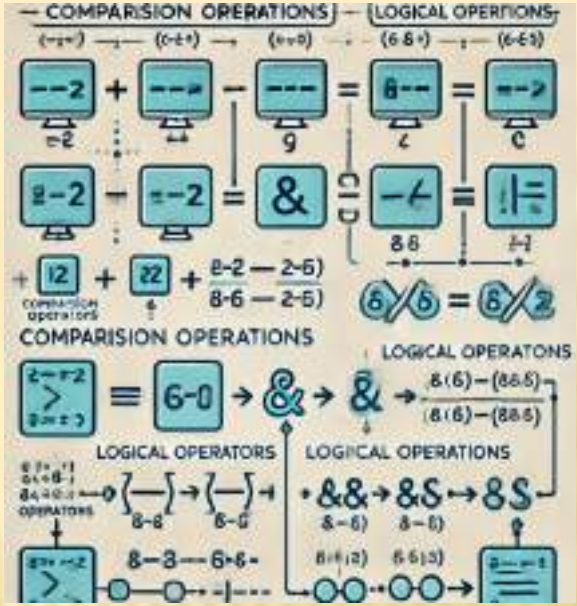
```
int x = 10;
x += 5; // Результат: x = 15 (еквівалентно x = x + 5)

int y = 8;
y -= 3; // Результат: y = 5 (еквівалентно y = y - 3)

int z = 6;
z *= 2; // Результат: z = 12 (еквівалентно z = z * 2)

int w = 20;
w /= 4; // Результат: w = 5 (еквівалентно w = w / 4)
```

Лекція №6

Операції мови програмування С++







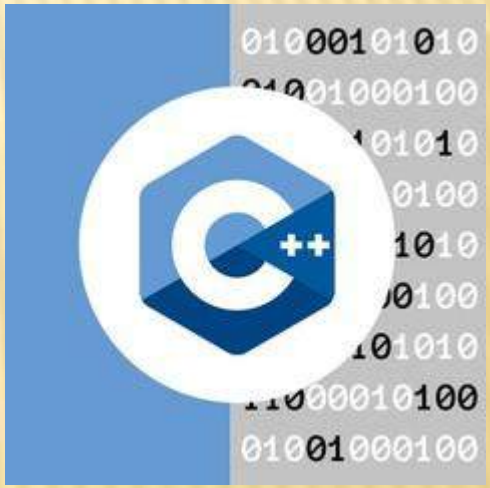
Лекція №7

Операції порівняння та логічні операції в мові С++.
Побітові операції.

ОПЕРАЦІЇ ПОРІВНЯННЯ С++

Операції порівняння застосовуються для порівняння (зіставлення) числової або символьної інформації. Результатом виконання операцій є або істина (true), або брехня (false). Перерахуємо ці операції:

Назва операції	Знак операції в С
Більше	>
Більше або дорівнює	>=
Менше	<
Менше або дорівнює	<=
Дорівнює	==
Не дорівнює	!=



Пріоритет цих операцій нижче, ніж у арифметичних операцій. Приклад використання: **A+B>=C.**

Перевіряємо: сума A і B більше C? Якщо «так», то відповіддю буде *true*, якщо "ні", то *false*.

У використанні операцій порівняння немає нічого складного. Але слід звернути увагу на одну обставину.

Дійсні числа в пам'яті зберігаються приблизно, при роботі з ними в ході виконання арифметичних операцій можуть накопичуватися похибки, тому не рекомендується виконувати перевірку на рівність або на нерівність для дійсних чисел, тому що результат може бути непередбачуваний.

КОНСТРУКЦІЯ IF

if / else – базовий інструмент керування потоком виконання. Він дозволяє вибрати одну з гілок коду залежно від істинності умови

Синтаксис (мінімальний):

```
if (умова) {
    // код виконується, якщо умова == true
}
```

Приклад:

```
if (a < b)  { cout << "a < b\n"; }
```

КОНСТРУКЦІЯ IF / ELSE

if / else – базовий інструмент керування потоком виконання. Він дозволяє вибрати одну з гілок коду залежно від істинності умови

Повна розгалужувальна конструкція виглядає так:

```
if (умова) {
    // гілка для true
} else {
    // гілка для false
}
```

Приклад:

```
if (x < 10) {
    cout << "x менше 10\n";
} else {
    cout << "x не менше 10\n";
}
```

ОПЕРАЦІЇ ПОРІВНЯННЯ C++ (ПРИКЛАД)

Порівняння менше (<), більше (>), менше або рівне (<=), більше або рівне (>=):

```cpp
int a = 5;
int b = 6;

if (a < b) {
    cout << "a менше b" << endl;
}

if (a > b) {
    cout << "a більше b" << endl;
}

if (a <= b) {
    cout << "a менше або рівне b" << endl;
}

if (a >= b) {
    cout << "a більше або рівне b" << endl;
}
```

ОПЕРАЦІЇ ПОРІВНЯННЯ С++

Логічні операції тісно пов'язані з операціями порівняння і використовуються для побудови складних логічних виразів. Є три логічні операції:

Назва	*Знак операції*	*Приклад запису*	*Пояснення*
Логічне заперечення (**НІ**)	**!**	**!X**	Якщо **X** — істина, то результат — не істина и навпаки
Логічне множення кон'юнкція (**І**)	**&&**	**X && Y**	Результат — істина, якщо істинні обидва операнда
Логічне додавання диз'юнкція (**АБО**)	**||**	**X || Y**	Результат — істина, якщо істиною є хоча б один операнд

ОПЕРАЦІЇ ПОРІВНЯННЯ C++

Логічні оператори && (і) та || (або) для комбінування умов:

```
int a = 5;
int b = 6;
int c = 7;

if (a < b && b < c) {
    cout << "a менше b і b менше c" << endl;
}

if (a < b || a < c) {
    cout << "a менше b або a менше c" << endl;
}
```

ОПЕРАЦІЇ ПОРІВНЯННЯ C++

Приклад використання конструкції if-else в C++:

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

int main() {
    int a = 10;

    if (a > 5) {
        cout << "a більше за 5" << endl;
    }
    else {
        cout << "a менше або дорівнює 5" << endl;
    }

    return 0;
}
```

ОПЕРАЦІЇ ПОРІВНЯННЯ С++

Приклад використання вкладених конструкції if-else в С++:

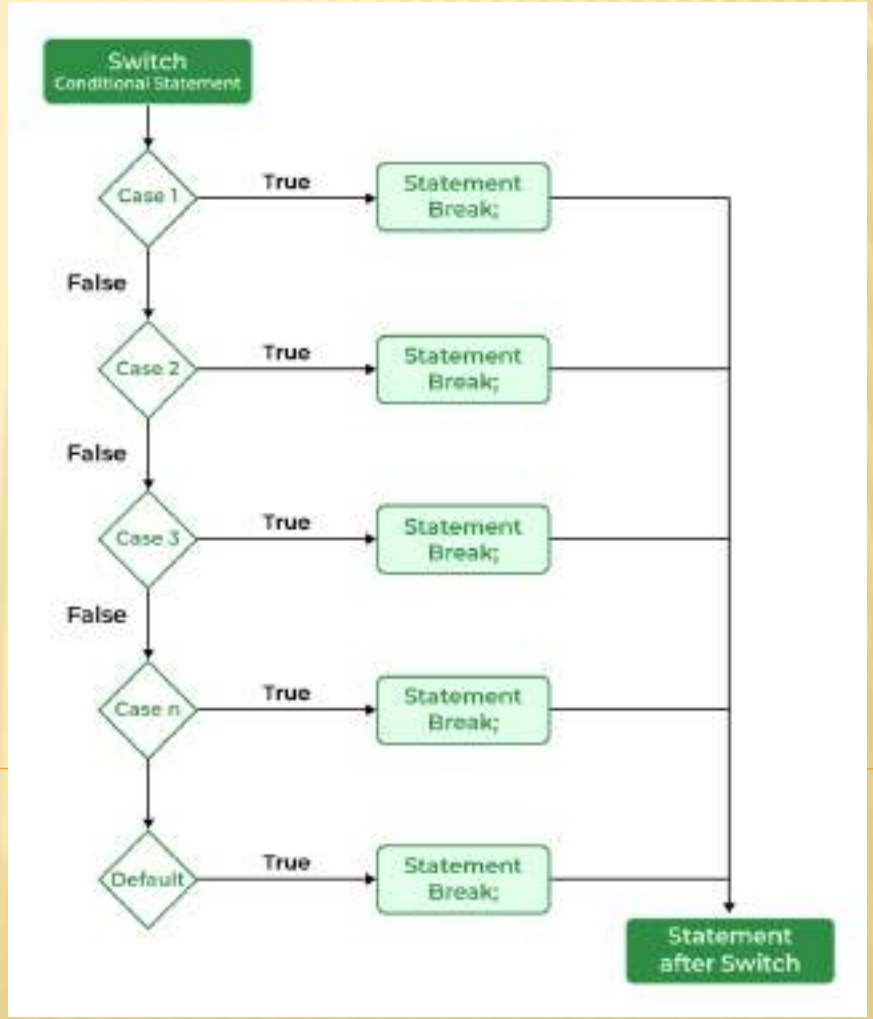
```
int x = -5;

if (x > 0) {
    cout << "x додатне" << endl;
} else if (x < 0) {
    cout << "x від'ємне" << endl;
} else {
    cout << "x дорівнює нулю" << endl;
}
```

SWITCH-CASE В C++

Оператор switch-case в C++ використовується для перевірки змінної на кілька можливих значень. Це зручна альтернатива конструкції if-else, коли потрібно порівняти одну змінну з кількома варіантами.

General form:

```
switch(expression) {
  case x:
    // code block
    break;
  case y:
    // code block
    break;
  default:
    // code block
}
```



SWITCH-CASE В C++

Оператор switch-case в C++ використовується для перевірки змінної на кілька можливих значень. Це зручна альтернатива конструкції if-else, коли потрібно порівняти одну змінну з кількома варіантами.

```
int day = 3;

switch (day) {
    case 1:
        cout << "Понеділок" << endl;
        break;
    case 2:
        cout << "Вівторок" << endl;
        break;
    case 3:
        cout << "Середа" << endl;
        break;
    case 4:
        cout << "Четвер" << endl;
        break;
    case 5:
        cout << "П'ятниця" << endl;
        break;
    case 6:
        cout << "Субота" << endl;
        break;
    case 7:
        cout << "Неділя" << endl;
        break;
    default:
        cout << "Неправильний день" << endl;
}
```

ТЕРНАРНИЙ ОПЕРАТОР

Тернарний оператор (умовний вираз) в C++ — це скорочена форма оператора if-else, яка використовується для виконання простих умовних операцій. Він має наступний синтаксис:

```
умова ? значення_якщо_правда : значення_якщо_неправда;
```

Приклад:

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

int main() {
    int a = 5;
    int b = 10;

    // Тернарна операція
    int min = (a < b) ? a : b;

    cout << "Мінімальне число: " << min << endl;

    return 0;
}
```

ПОРОЗРЯДНІ (ПОБІТОВІ) ОПЕРАЦІЇ

У С++ порозрядні (або побітові) операції працюють на рівні окремих бітів цілих чисел. Ось основні побітові операції та приклад їх використання:

Назва	*Знак операції*	*Приклад запису*	*Результат виконання*
Поразрядне **І**	**&**	**3&5**	**1**
Поразрядне **АБО**	**|**	**3|5**	**7**
Поразрядне виключаюче **АБО**	**^**	**3^5**	**6**
Інверсія	**~**	**~3**	**-4**
Поразрядний зсув вліво	**<<**	**5<<1**	**10**
Поразрядний зсув вправо	**>>**	**5>>1**	**2**

ПОРОЗРЯДНІ (ПОБІТОВІ) ОПЕРАЦІЇ

Операції бітового зсуву можуть бути корисні при декодуванні інформації від зовнішніх пристроїв і для читання інформації про статус. Оператори бітового зсуву можуть також використовуватися для виконання швидкого множення і ділення цілих чисел. Зрушення вліво рівносильний множенню на 2, а зрушення вправо – поділу на 2, як показано в таблиці. Бітові операції найбільш часто застосовуються при розробці драйверів пристроїв, наприклад програм для модемів, дисків і принтерів, оскільки бітові оператори можуть використовуватися для виключення деяких бітів, наприклад парності. (Біт парності використовується для підтвердження того, що інші біти в байті не змінювалися. Він, як правило, є старшим бітом в байті.).

	Бітове представлення результату	значення x
char x;		
x = 7;	00000111	7
x = x << 1;	00001110	14
x = x << 3;	01110000	112
x = x << 2;	11000000	192
x = x >> 1;	01100000	96
x = x >> 2;	00011000	24

УНАРНІ ОПЕРАЦІЇ &, *

Перш ніж розглядати зазначені операції згадаємо, як розташовуються дані у пам’яті. Кожному значенню відповідає наступний шаблон

адреса	ідентіфікатор	значення (згідно до типу)

Змінна – це область пам'яті, до якої ми звертаємося за даними, що знаходяться там, використовуючи ідентифікатор (в даному випадку, ім'я змінної). При цьому у цій поміченій ім'ям області є ще й адреса, яка виражається числом і зрозуміла комп'ютерній системі. Цю електронну адреса можна отримати і записати в особливу змінну. Змінну, що містить адресу області пам'яті, називають покажчиком.

Покажчик – це адреса змінної в пам'яті. Покажчик на змінну – це змінна, спеціально створена для зберігання покажчика на об'єкт певного типу. Знаючи адресу змінної, можна істотно спростити роботу деяких програм. Покажчики мають три головних призначення в С:

- надають швидке звернення до елементів масиву;
- дозволяють функцій модифікувати передані параметри;
- підтримують динамічні структури даних, наприклад списки.

ОПЕРАТОР &

Оператор &. Це унарний оператор, який повертає адресу операнда в пам'яті. Наприклад:

```
m = &count;
```

розміщує в m адресу змінної count. Це адреса внутрішнього розташування змінної в комп'ютері. З самим значенням змінної нічого не робиться. Оператор & можна запам'ятати як «взяття адреси». Тому вищезгаданий оператор присвоювання можна прочитати як «m отримує адресу count».
Для кращого розуміння даного присвоювання припустимо, що змінна count знаходиться за адресою 2000. Також припустимо, що count має значення 100. Після цього присвоювання m буде містити 2000.

ОПЕРАТОР &

Оператор *, що доповнює &. Це унарний оператор, який повертає значення змінної за вказаною адресою. Наприклад: якщо m містить адресу змінної count, то

```
q = * m;
```

розміщує значення count в q. З вищенаведеного прикладу випливає, що q отримає значення 100, оскільки 100 зберігалося за адресою 2000 – адресою, що знаходиться в змінній m. Операцію * можна запам'ятати як «за адресою». В даному випадку оператор можна прочитати як «q отримує значення за адресою m».

Адреса	Ідентифікатор	Значення
2000	count	100

УНАРНІ ОПЕРАЦІЇ &, *

Знаки для множення і для взяття «за адресою» – однакові, втім як і знаки бітового І та «взяття адреси». Ці оператори не мають зв'язку між собою. Як **&** так і ***** мають більш високий пріоритет у порівнянні з іншими арифметичними операціями, за винятком унарного мінуса, що має такий же пріоритет.

Змінні, що містять адреси або покажчики, повинні оголошуватися шляхом приміщення ***** перед ім'ям змінної для вказівки компілятору того, що змінна містить покажчик. Наприклад, для оголошення покажчика *ch* на символ, слід написати

char * ch;

Тут *ch* – це не символ, а покажчик на символ, в чому і полягає принципова відмінність. Тип даних, на який вказує покажчик, як в нашому випадку **char**, називається базовим типом покажчика. Сам покажчик – це змінна, яка використовується для зберігання адреси об'єкта базового типу. Отже, покажчик на символ (або будь-який інший покажчик) має фіксований розмір, який визначається архітектурою комп'ютера, для зберігання адреси. Важливо запам'ятати, що покажчик слід використовувати тільки для вказівки на типи даних, що збігаються з базовим типом.

УНАРНІ ОПЕРАЦІЇ &, * В МОВІ С++

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

int main() {
    int x = 10;
    int *ptr = &x; // ptr містить адресу змінної x

    cout << "Значення x: " << x << endl; // Виводить: Значення x: 10
    cout << "Адреса x: " << &x << endl; // Виводить: Адреса x: 0x7fff5fbff6ec

    cout << "Значення, що міститься за адресою, на яку вказує ptr: " << *ptr <<

    *ptr = 20; // Змінює значення, яке зберігається за адресою ptr (тобто, x тег

    cout << "Нове значення x: " << x << endl; // Виводить: Нове значення x: 20

    return 0;
}
```

ФУНКЦІЇ ВВОДУ-ВИВОДУ С, С++

Почнемо розглядати процедури вводу виводу даних із стандартних функцій базової мови С.

Бібліотека

stdio.h. – С**,** **cstdio** – С++

stdio.h (від англ. standard input/output header) – заголовок стандартної бібліотеки мови Сі, що складається з визначень макросів, констант і оголошення функцій і типів, які використовуються для різних операцій стандартного введення і виведення.

Функції, макроси	Призначення
printf()	виведення інформації у стандартний потік (через буфер на екран)
scanf()	зчитування інформації із стандартного потоку (через буфер з екрану)
int getc(), int getchar	зчитує з екрану або файлу і повертає символ з даного потоку і змінює покажчик позиції файлу
putc(char),putchar( )	записує і повертає символ в потік і змінює покажчик позиції файлу на нього
char *gets(char *str)	зчитує символьний рядок
puts()	виводить символьний рядок

ФУНКЦІЇ ВВОДУ-ВИВОДУ C, C++

printf:

Функція printf використовується для форматованого виведення даних на екран (консоль). Вона приймає рядок формату та додаткові аргументи для виведення у вказаному форматі.

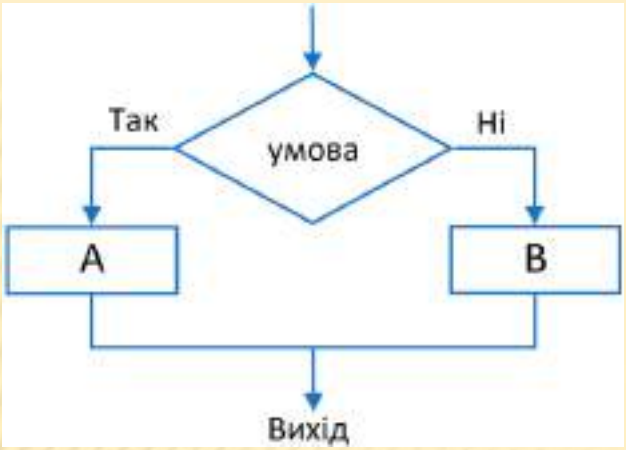
```c
#include <stdio.h>

int main() {
    int num = 10;
    float f = 3.14;

    printf("Це ціле число: %d\n", num);  // Виведе: Це ціле число: 10
    printf("Це число з плаваючою комою: %f\n", f);  // Виведе: Це число з плава

    return 0;
}
```

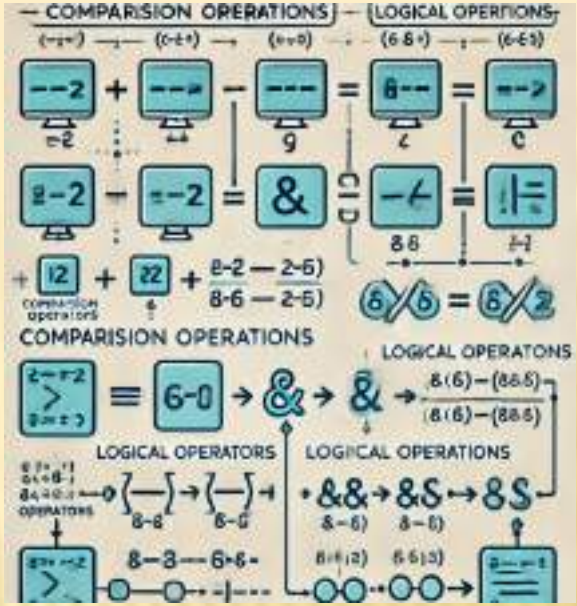
ФУНКЦІЇ ВВОДУ-ВИВОДУ C, C++

scanf:

Функція scanf використовується для зчитування даних з клавіатури. Вона працює аналогічно до printf, але приймає додаткові аргументи, які вказують на те, куди зберігати зчитані дані.

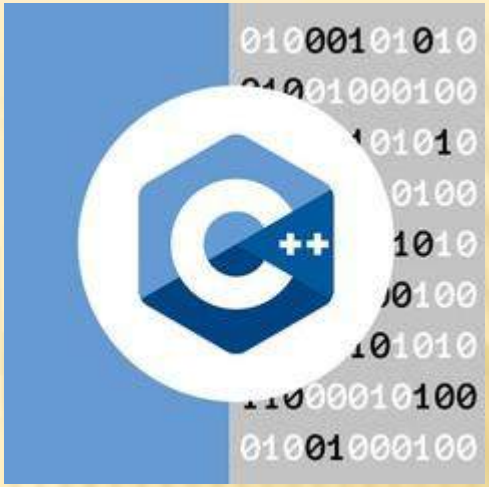
```c
#include <stdio.h>

int main() {
    int num;
    float f;

    printf("Введіть ціле число: ");
    scanf("%d", &num);  // Зчитує ціле число з клавіатури

    printf("Введіть число з плаваючою комою: ");
    scanf("%f", &f);  // Зчитує число з плаваючою комою з клавіатури

    printf("Ви ввели: %d і %.2f\n", num, f);

    return 0;
}
```





Лекція №7

Операції порівняння та логічні операції в мові С++.
Побітові операції.



Лекція №8

Операції вводу-виводу в мові C++

ОПЕРАЦІЇ ПОРІВНЯННЯ С++

Почнемо розглядати процедури вводу виводу даних із стандартних функцій базової мови С.

Бібліотека

stdio.h. – С, **cstdio** – С++

stdio.h (від англ. standard input/output header) – заголовок стандартної бібліотеки мови Сі, що складається з визначень макросів, констант і оголошення функцій і типів, які використовуються для різних операцій стандартного введення і виведення.

Функції, макроси	Призначення
printf()	виведення інформації у стандартний потік (через буфер на укран)
scanf()	зчитування інформації із стандартного потоку (через буфер з екрану)
int getc(), int getchar	зчитує з екрану або файлу і повертає символ з даного потоку і змінює покажчик позиції файлу
putc(char),putchar()	записує і повертає символ в потік і змінює покажчик позиції файлу на нього
char *gets(char *str)	зчитує символьний рядок
puts()	виводить символьний рядок

ФУНКЦІЇ ВВОДУ-ВИВОДУ C, C++

printf:

Функція printf є однією з основних функцій для виведення даних у мові програмування C та C++. Вона використовується для форматованого виведення рядків на консоль.

```
#include <cstdio>

int main() {
    printf("Hello, world!\n");
    return 0;
}
```

ФУНКЦІЇ ВВОДУ-ВИВОДУ С, С++

Функція ***printf***

printf ("управляючий рядок", список змінних)

Для функції виводу управляючий рядок включає:

- керуючі символи;
- текст, представлений для безпосереднього виведення;
- формат, призначений для виведення значень змінних різних типів.

Якщо виводиться тільки текст, який зазначений в управляючому рядку, то список змінних може бути відсутній.

Розглянемо компоненти управляючого рядка, які обов’язково беруться у лапки.

Керуючі символи не виводяться на екран, а керують розташуванням символів, що виводяться. Відмінною рисою керуючого символу є наявність зворотного слеша '\' перед ним.

ОСНОВНІ КЕРУЮЧІ СИМВОЛИ:

'\ n' - новий рядок;
'\ t' - горизонтальна табуляція;
'\ v' - вертикальна табуляція;
'\ b' - повернення на символ;
'\ r' - повернення на початок рядка;
'\ a' - звуковий сигнал.

Формати потрібні для того, щоб вказувати вид, в якому інформація буде виведена на екран. Відмінною рисою формату є наявність символу відсоток '%' перед ним:

% d – ціле число типу int зі знаком в десятковій системі числення;
% u – ціле число типу unsigned int;
% x – ціле число типу int зі знаком в шістнадцятковій системі числення;
% o – ціле число типу int зі знаком в вісімковій системі числення;
% hd – ціле число типу short зі знаком в десятковій системі числення;
% hu – ціле число типу unsigned short;
% hx – ціле число типу short зі знаком в шістнадцятковій системі числення;
% ld – ціле число типу long int зі знаком в десятковій системі числення;
% lu – ціле число типу unsigned long int;
% lx – ціле число типу long int зі знаком в шістнадцятковій системі числення;
% f – дійсний формат (числа з плаваючою точкою типу float);
% c – символьний формат;
% s – строковий формат.

```cpp
#include <cstdio>

int main() {
    int num = 42;
    float pi = 3.14;
    char ch = 'A';
    const char* str = "Hello, world!";

    printf("Integer: %d\n", num);
    printf("Float: %.2f\n", pi); // Виводить дробове число з 2 знаками після ко
    printf("Character: %c\n", ch);
    printf("String: %s\n", str);

    printf("New line: Line1\nLine2\n"); // Друкує дві рядки

    printf("Horizontal tab:\tTabbed Text\n"); // Виводить горизонтальну табуля

    printf("Vertical tab:\vTabbed Text\n"); // Виводить вертикальну табуляцію

    printf("Backspace:\bBack\b space\n"); // Використовує зворотний пробіл

    printf("Carriage return:\rReturn to start\n"); // Використовує відступ в п

    printf("Alert:\aBeep!\n"); // Виводить звуковий сигнал

    return 0;
}
```

ФУНКЦІЇ ВВОДУ-ВИВОДУ C, C++

scanf:

Функція scanf використовується для зчитування даних з клавіатури. Вона працює аналогічно до printf, але приймає додаткові аргументи, які вказують на те, куди зберігати зчитані дані.

```c
#include <stdio.h>

int main() {
    int num;
    float f;

    printf("Введіть ціле число: ");
    scanf("%d", &num);  // Зчитує ціле число з клавіатури

    printf("Введіть число з плаваючою комою: ");
    scanf("%f", &f);  // Зчитує число з плаваючою комою з клавіатури

    printf("Ви ввели: %d і %.2f\n", num, f);

    return 0;
}
```

C++ ВВЕДЕННЯ ВИВЕДЕННЯ ПОТОКАМИ

Бібліотека потокового введення-виведення **iostream** може бути використана в якості альтернативи відомої стандартної бібліотеки вводу-виводу мови С - **stdio**.

В основі ООП підходу, реалізованого засобами **iostream**, лежить припущення про те, що об'єкти мають знання того, які дії слід робити при введенні і виведенні.

При користуванні бібліотекою **iostream** помилки, пов'язані з "змішування" типів даних, виключені. Якщо ви використовуєте в операції введення-виведення змінну типу ***unsigned long***, то викликається підпрограма, відповідальна саме за цей тип.

Бібліотека **stdio** підтримує засоби мови С, що дозволяє використовувати змінне число параметрів. Але така гнучкість дається не даром – на етапі компіляції перевірка відповідності між специфікацією формату, як у функціях ***printf () і scanf (),*** не виконується.

C++ ВВЕДЕННЯ ВИВЕДЕННЯ ПОТОКАМИ

Бібліотека iostream у мові C++ надає більш сучасний спосіб здійснення введення та виведення. Вона надає класи **std::istream** для введення та **std::ostream** для виведення. Крім того, її можна використовувати разом із більшістю інших стандартних бібліотек C++ для зручного та ефективного програмування.

Ось приклад використання iostream для виведення рядка:

```
#include <iostream>

int main() {
    std::cout << "Hello, world!" << std::endl;
    return 0;
}
```

У цьому прикладі std::cout використовується для виведення рядка "Hello, world!" на консоль. Крім того, std::endl використовується для додавання нового рядка.

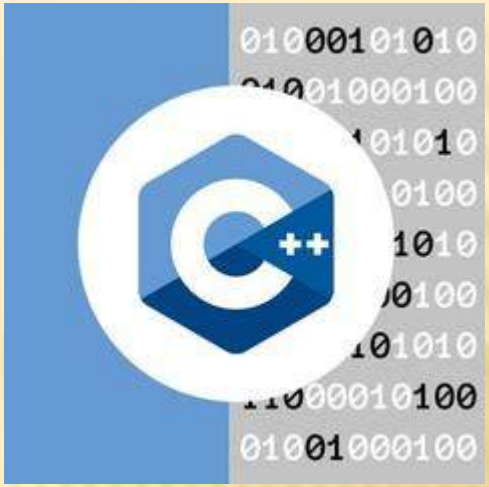
C++ ВВЕДЕННЯ ВИВЕДЕННЯ ПОТОКАМИ

Аналогічно, для введення можна використовувати std::cin:

```cpp
#include <iostream>

int main() {
    int num;
    std::cout << "Enter a number: ";
    std::cin >> num;
    std::cout << "You entered: " << num << std::endl;
    return 0;
}
```

У цьому прикладі, std::cin використовується для зчитування цілого числа, яке користувач вводить.

Бібліотека iostream надає багатий набір можливостей для роботи з введенням та виведенням у C++. Вона включає в себе різні операції, перевантаження операторів та інші функції, що дозволяють зручно та ефективно взаємодіяти з користувачем та файлами.



Лекція №8

Операції вводу-виводу в мові C++

Лекція №9

Цикли в мові С++

ЦИКЛИ C++

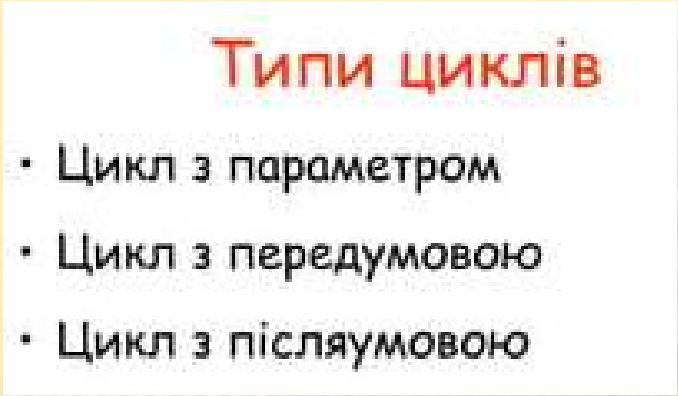
Що таке цикл:

Цикли — це фундаментальний механізм повторення дій у програмі . Вони дозволяють автоматично виконувати багаторазові завдання без дублювання коду.

◇ Навіщо потрібні цикли:

1. **Автоматизація повторень** — обробка елементів масиву, файлів, обчислень.
2. **Робота з колекціями** — зручно переглядати, змінювати або аналізувати кожен елемент.
3. **Керування потоком** — визначають, скільки разів виконується частина коду.
4. **Мінімізація дублювання** — менше коду, більше гнучкості.
5. **Багаторазовість** — спільні рішення для різних даних або об’єктів.
6. **Ефективність** — зменшення навантаження та економія ресурсів.

ТИПИ ЦИКЛІВ

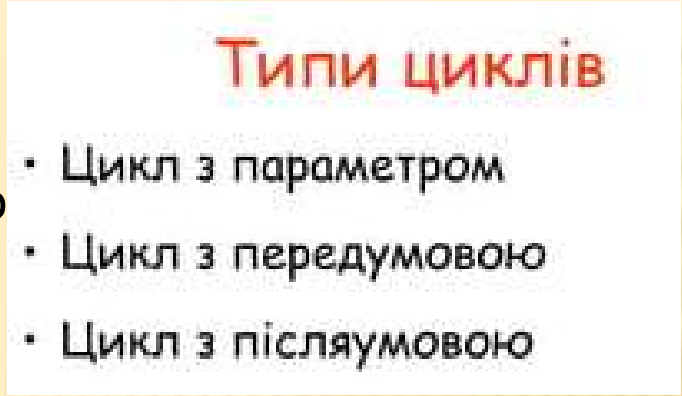


В С ++ визначені три різних циклу:
- цикл з передумовою:
while (вираз-умова) тіло_циклу
- цикл з післяумовою:
do тіло_циклу
while (вираз-умова);
- цикл з параметром:
for (ініціалізація_циклу; вираз-умова; вираз_корекції) тіло_ циклу.

Тіло циклу – це окремий оператор, який завжди завершується крапкою з комою, або складовою оператор, або блок. Тільки опис або визначення не може бути тілом циклу. Оператори циклу задають багаторазове виконання операторів тіла циклу.

Вираз-умова – у всіх операторах скалярний вираз (найчастіше логічний або арифметичний) визначає умову продовження повторення обробки, якщо вона э істиною (відмінна від нуля).

ТИПИ ЦИКЛІВ



Цикл for:

Використовується для виконання певного кількість разів.

Синтаксис:

```
for (ініціалізація; умова; зміна) {
    // тіло циклу
}
```

Це найпопулярніший оператор циклу для всіх С-подібних мов. Популярність пояснюється надзвичайною гнучкістю і міццю цього оператора. Його ще називають циклом з лічильником.

Ініціалізація – це будь-які оператори присвоювання, записані через кому. Тут, як правило, робиться ініціалізація різних змінних, необхідних для роботи в циклі **for**, наприклад, встановлення початкового значення лічильника.

Умова – логічний або арифметичний вираз, записується так само, як і для інших операторів циклу і оператора if. Найчастіше це умова завершення циклу.

Зміна – будь-які записані через кому оператори присвоювання і (або) оператори-вирази (збільшення або зменшення). Найчастіше це зміна значення лічильника.

ТИПИ ЦИКЛІВ

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

int main() {
    for (int i = 1; i <= 5; ++i) { // ініціалізація; умова; зміна
        cout << i << " ";
    }
    return 0;
}
```

ЦИКЛ - FOR

```
for (int i = 0; i < 5; i++) {
    std::cout << "Це повторення номер " << i + 1 << std::endl;
}
```

for (...): Це початок оголошення циклу for.

int i = 0;: Ця частина оголошує нову локальну змінну i та ініціалізує її значенням 0. Ця змінна використовуватиметься для лічильника циклу.

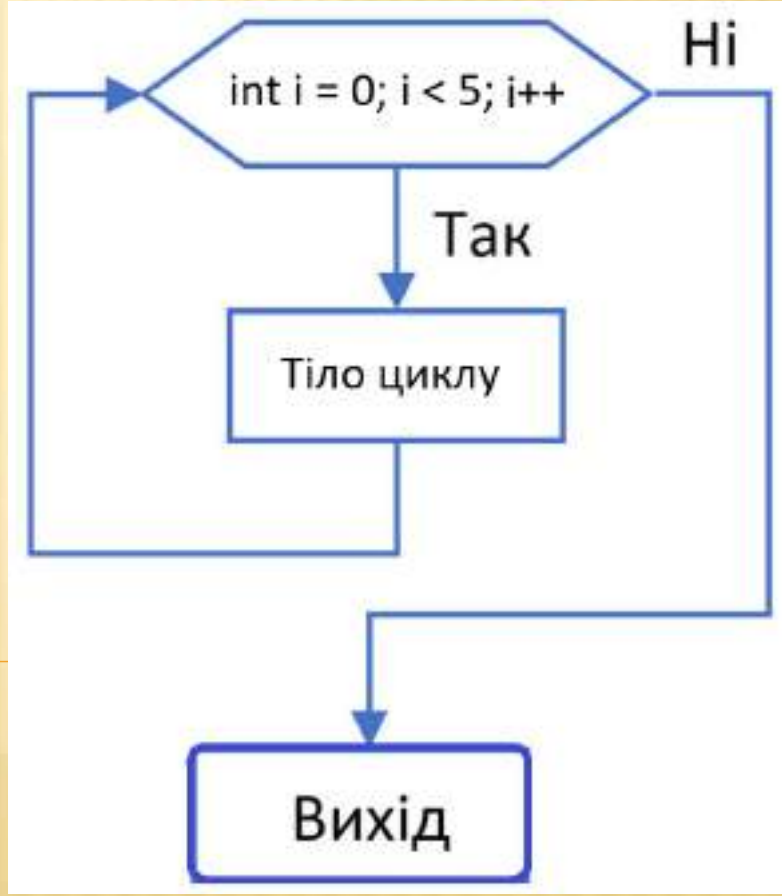
i < 5;: Це умова, яка повинна бути виконана, щоб цикл продовжувався. У цьому випадку, цикл буде виконуватися, поки i менше 5.

i++: Цей оператор збільшує значення i на 1 після кожного виконання блоку коду в циклі.

{...}: Це тіло циклу, тобто блок коду, який виконується при кожній ітерації.

У цьому випадку, std::cout << "Це повторення номер " << i + 1 << std::endl; виводить повідомлення на консоль. i + 1 використовується для виведення номеру повторення (починаючи з 1, а не з нуля).

```
Це повторення номер 1
Це повторення номер 2
Це повторення номер 3
Це повторення номер 4
Це повторення номер 5
```



ЦИКЛ WHILE

Цикл while:
Виконується, поки задана умова виконується.

Синтаксис оператора:
while <умова>:
    <тіло циклу>

Синтаксис:

```
while (умова) {
    // тіло циклу
}
```

Як бачимо – це цикл з передумовою. Спочатку перевіряємо істинність деякої умови, а потім, якщо вона істина, виконуємо оператори (один або кілька), що становлять тіло циклу.

Такий цикл не виконується жодного разу, якщо умова спочатку помилкова.

Умова – це будь-який вираз, що має результат логічного або арифметичного типу, тобто умова для операторів циклу записується так само, як для оператора розгалуження **if**.

ЦИКЛ WHILE

```
int i = 0;
while (i < 5) {
    std::cout << "Це повторення номер " << i + 1 << std::endl;
    i++;
}
```

int i = 0; ця строка оголошує змінну i та ініціалізує її значенням 0. Ця змінна використовуватиметься для лічильника циклу.

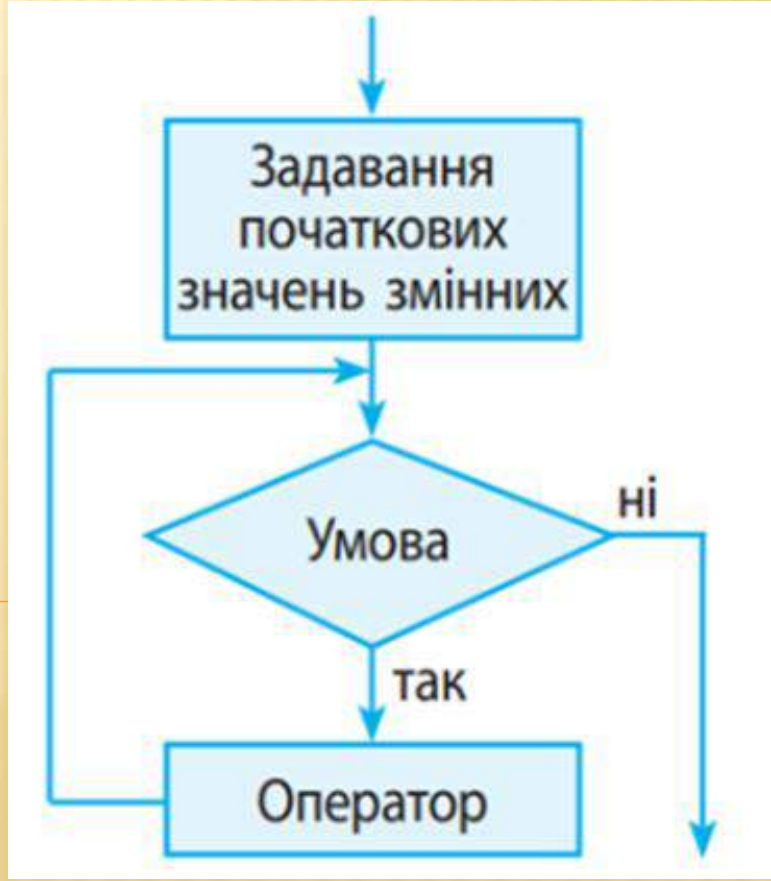
while (i < 5) { ... }: Це початок блоку коду циклу while. Умова i < 5 вказує, що цикл буде виконуватися, поки і менше 5.

{...}: Це тіло циклу, тобто блок коду, який виконується при кожній ітерації.

У цьому випадку, std::cout << "Це повторення номер " << i + 1 << std::endl; виводить повідомлення на консоль. i + 1 використовується для виведення номеру повторення (починаючи з 1, а не з нуля).

i++;: Цей оператор збільшує значення і на 1 після кожного виконання блоку коду в циклі.

```
Це повторення номер 1
Це повторення номер 2
Це повторення номер 3
Це повторення номер 4
Це повторення номер 5
```



ЦИКЛ – “DO – WHILE”

Цикл do-while:

Виконується принаймні один раз, а потім повторюється, поки задана умова виконується.

Синтаксис:

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

Цикл **do** завжди виконується хоча б один раз. І далеко не у всіх завданнях це добре.

ЦИКЛ – “DO – WHILE”

```
int i = 0;
do {
    std::cout << "Це повторення номер " << i + 1 << std::endl;
    i++;
} while (i < 5);
```

int i = 0;: Ця лінія оголошує змінну і та ініціалізує її значенням 0. Ця змінна використовуватиметься для лічильника циклу.

while (i < 5) { ... }: Це початок блоку коду циклу while. Умова i < 5 вказує, що цикл буде виконуватися, поки і менше 5.

{...}: Це тіло циклу, тобто блок коду, який виконується при кожній ітерації.

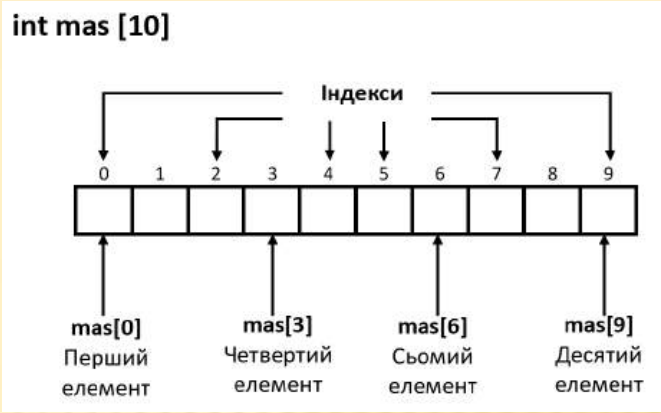
У цьому випадку, std::cout << "Це повторення номер " << i + 1 << std::endl; виводить повідомлення на консоль. i + 1 використовується для виведення номеру повторення (починаючи з 1, а не з нуля).

i++;: Цей оператор збільшує значення і на 1 після кожного виконання блоку коду в циклі.

```
Це повторення номер 1
Це повторення номер 2
Це повторення номер 3
Це повторення номер 4
Це повторення номер 5
```

Лекція №9

Цикли в мові С++



Лекція №10

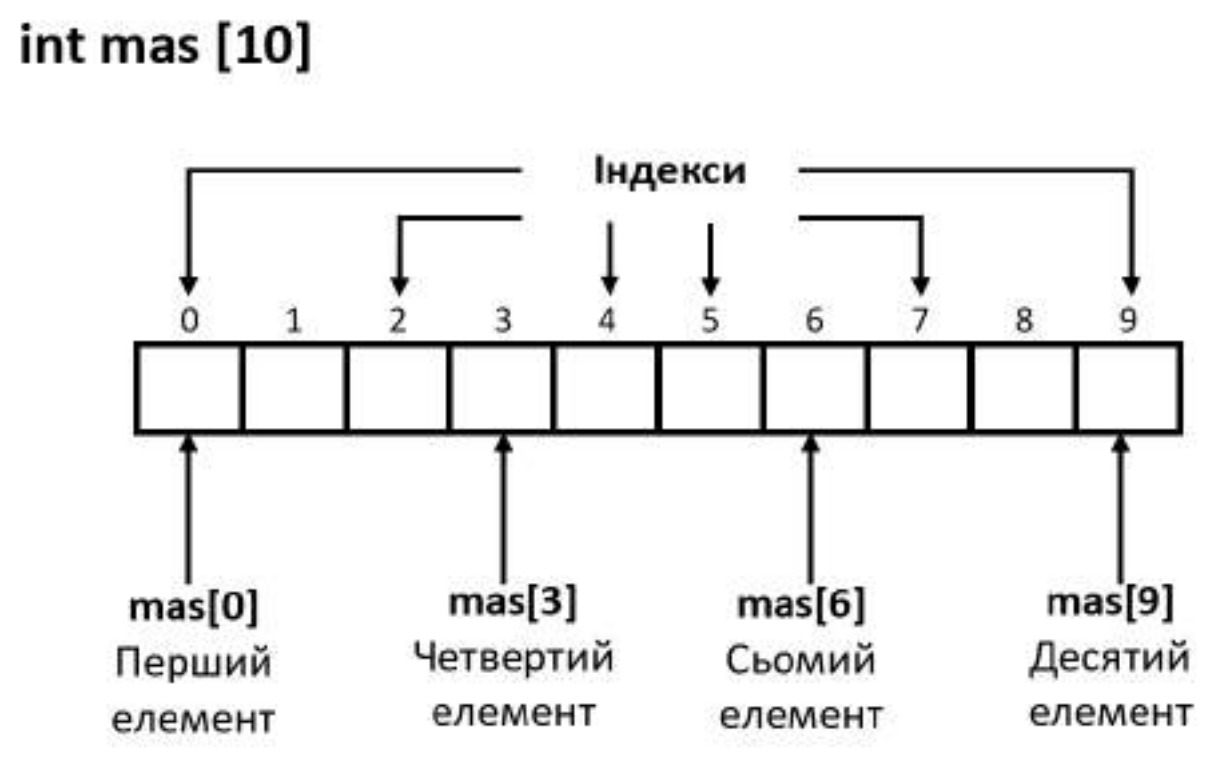
Масиви даних мови С++

МАСИВИ МОВИ С++

Масиви – важлива річ у програмуванні; це структура даних, подана у вигляді групи комірок одного типу, об’єднаних одним іменем. Масиви використовуються для обробки великої кількості однотипних даних. Окрема комірка даних масиву називається елементом. Елементами масиву можуть бути дані будь-якого типу.

Масиви можуть мати один або більше одного вимірів. Залежно від кількості вимірів масиви діляться на одновимірні, двовимірні, тривимірні і так далі до n-вимірного масиву.

Масив – це сховище даних, контейнер. Обробляються його елементи кожний окремо. Кожний елемент масиву ідентифікується номером. Компілятор сам нумерує елементи масиву; нумерація в С починається з нуля. Для доступу до елементу масиву використовуються ім’я змінної-масиву та його індекс.



ОДНОМІРНІ МАСИВИ В С ++(ВЕКТОРИ)

Одновимірна масив – масив, з одним параметром, що характеризує кількість елементів одновимірного масиву. Фактично одновимірний масив – це масив, у якого може бути тільки один рядок, і *n*-та кількість стовпців. Стовпчики в одновимірному масиві – це елементи масиву. Ще одномірні масиви називають векторами. На малюнку 1 показана структура цілочисельного одновимірного масиву a. Розмір цього масиву – 16 комірок.

5	-12	-12	9	10	0	-9	-12	-1	23	65	64	11	43	39	-15
a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]	a[10]	a[11]	a[12]	a[13]	a[14]	a[15]

Максимальний індекс одновимірного масиву *a* дорівнює 15, але розмір масиву 16 елементів, тому що нумерація елементів масиву завжди починається з 0. Індекс елементу – це ціле невід'ємне число, за яким можна звертатися до кожної клітинки масиву і виконувати будь-які дії з нею (елементом).

СИНТАКСИС ОПИСУ МАСИВУ:

тип ідентифікатор[кількість елементів];

Так масив цілих чисел a з 5 елементів можна описати:

```
int numbers[5]; // Оголошує масив "numbers" з розміром 5
```

Завжди відразу після імені масиву йдуть квадратні дужки, в яких задається розмір одновимірного масиву, цим масив і відрізняється від всіх інших змінних.

Масиви можуть бути ініційовані при оголошенні:

Ініціалізація одновимірного масиву виконується в фігурних дужках після знака =, кожен елемент масиву відокремлюється від попереднього комою. В даному випадку компілятор сам визначить розмір одновимірного масиву. Розмір масиву можна не вказувати тільки при його ініціалізації, при звичайному оголошенні масиву обов'язково потрібно вказувати розмір масиву.

```
int numbers[5] = {10, 20, 30, 40, 50}; // Оголошує масив "numbers"
```

ДОСТУП ДО ЕЛЕМЕНТУ МАСИВУ

Доступ до елементів масиву в C++ здійснюється за допомогою індексу, що вказує на конкретний елемент у масиві. Індексація починається з нуля. Ось приклад доступу до елементу масиву:

```
int numbers[5] = {10, 20, 30, 40, 50}; // Оголошує масив "numbers"
```

В цьому прикладі numbers - це масив, а кожен елемент можна доступитися за допомогою індексу:

numbers[0] - перший елемент (10)
numbers[1] - другий елемент (20)
numbers[2] - третій елемент (30)
numbers[3] - четвертий елемент (40)
numbers[4] - п'ятий елемент (50)

Також можна змінювати значення елементів через їх індекс:

```
numbers[2] = 35; // Третій елемент тепер дорівнює 35
```

ДВОВИМІРНІ МАСИВИ В С ++ (МАТРИЦІ)

Припустимо, необхідно обробити деякі дані з таблиці. У таблиці є дві характеристики: кількість рядків і кількість стовпців. Також і в двовимірному масиві, крім кількості елементів масиву, є такі характеристики як, кількість рядків і кількість стовпців двовимірного масиву. Тобто, візуально, двовимірний масив - це звичайна таблиця, з рядками і стовпцями. Фактично двовимірний масив - це одновимірний масив одновимірних масивів. Структура двовимірного масиву, з ім'ям a, розміром m на n показана нижче

a[0][0]	a[0][1]	a[0][2]	a[0][3]	. . .	a[0][n]
a[1][0]	a[1][1]	a[1][2]	a[1][3]	. . .	a[1][n]
a[2][0]	a[2][1]	a[2][2]	a[2][3]	. . .	a[2][n]
. . .	. . .	. . .	. . .	. . .	. . .
a[m][0]	a[m][1]	a[m][2]	a[m][3]	. . .	a[m][n]

СИНТАКСИС ОПИСУ ДВУВИМІРНОГО МАСИВУ:

тип ідентифікатор[кількість рядків] [кількість стовпчиків];

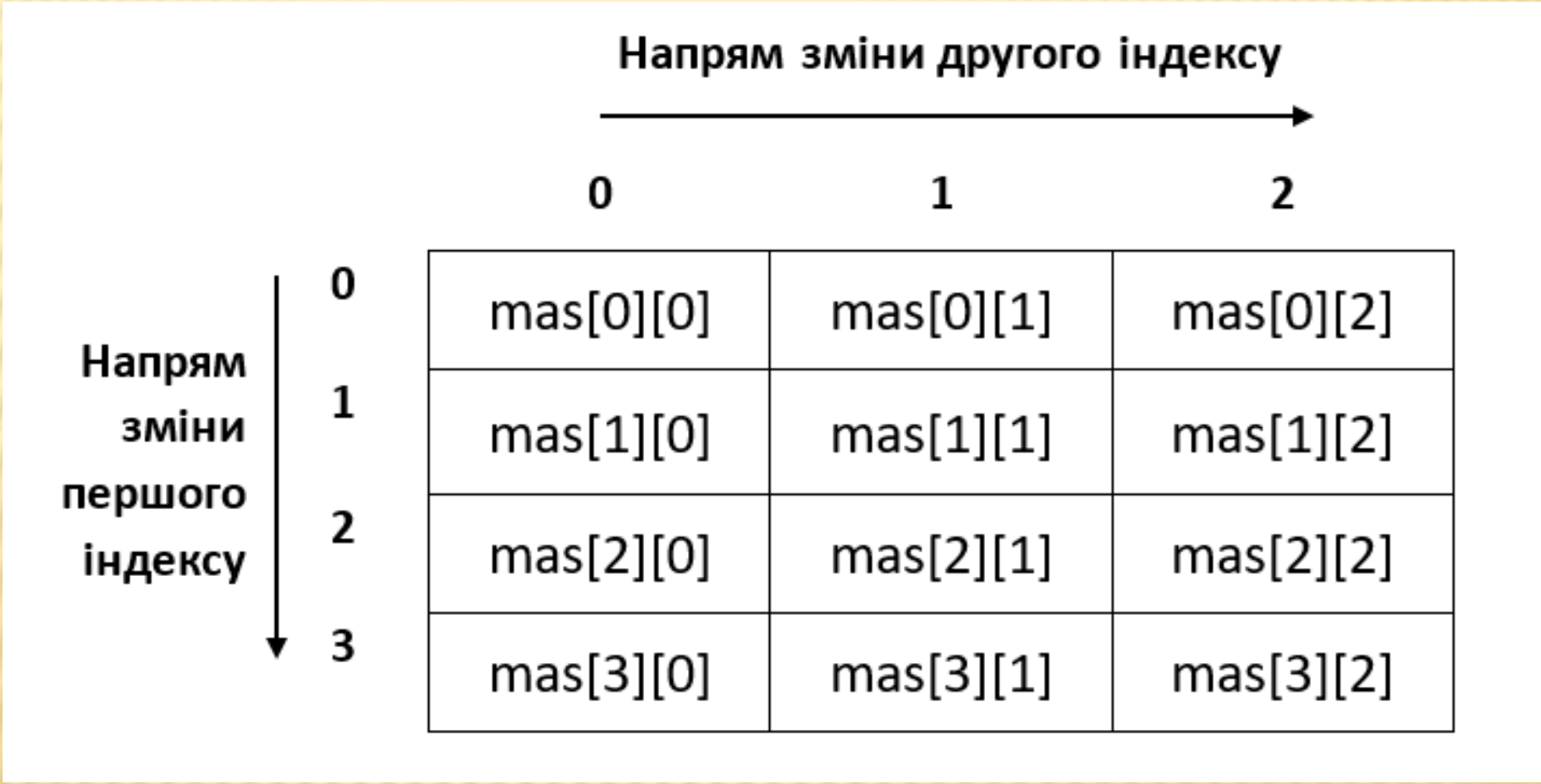
```
int matrix[3][3]; // Оголошення двовимірного масиву 3x3
```

У цьому випадку matrix - це двовимірний масив, що має 3 рядки та 3 стовпці. Доступ до елементів відбувається за допомогою двох індексів:.

Ініціалізаця двовимірного масиву:

```
matrix[0][0] = 1; // Перший рядок, перший стовпець
matrix[1][2] = 6; // Другий рядок, третій стовпець
```

СИНТАКСИС ОПИСУ ДВУВИМІРНОГО МАСИВУ:

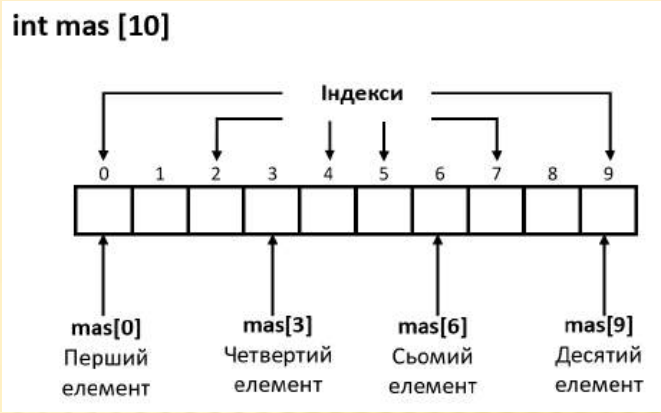


СИНТАКСИС ОПИСУ ДВУВИМІРНОГО МАСИВУ:

```
int matrix[4][5] = {
    {1, 2, 3, 4, 5},
    {6, 7, 8, 9, 10},
    {11, 12, 13, 14, 15},
    {16, 17, 18, 19, 20}
};
```

У цьому прикладі створюється двовимірний масив matrix розміром 4x5 та ініціалізується значеннями від 1 до 20.

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

int main() {
    int matrix[4][5] = {
        {1, 2, 3, 4, 5},
        {6, 7, 8, 9, 10},
        {11, 12, 13, 14, 15},
        {16, 17, 18, 19, 20}
    };

    int sum = 0;

    // Перший цикл проходить по рядках
    for (int i = 0; i < 4; i++) {
        // Другий цикл проходить по стовпцях
        for (int j = 0; j < 5; j++) {
            sum += matrix[i][j]; // Додаємо елемент до суми
        }
    }

    cout << "Сума елементів матриці: " << sum << endl;

    return 0;
}
```



Лекція №10

Масиви даних мови С++

Лекція №11

Підпрограми (процедури і функції)

ПІДПРОГРАМИ

Коли алгоритм розв'язуваної задачі нескладний, і розмір програми невеликий, то вона зазвичай пишеться як одне ціле. Тобто текст програми складається тільки з основної частини – головної програми **(на мові С ++ - це функція main())**. З ростом розміру програми з'являється необхідність розбивати завдання на підзадачі і оформлювати їх у вигляді окремих частин (підпрограм). У цьому випадку основна програма служить тільки для виклику цих підпрограм.

Підпрограма – це частина всієї програми, оформлена особливим чином. Як правило, у тексті підпрограми записується якась логічно завершена частина програми. Активне використання підпрограм при розробці програмного забезпечення складає основу модульного програмування.

Модульне програмування – така технологія програмування, коли алгоритм всієї розв'язуваної задачі розбивається на окремі, логічно завершені частини. Якщо ці частини складні для кодування, то вони знову розбиваються на більш прості частини. Цей процес триває до тих пір, поки не вийдуть прості для програмування алгоритми, призначені для реалізації нескладних дій. Основні принципи модульного програмування були сформульовані ще в 60-х роках 20-го століття.

Головна відмінність підпрограми від основної (головної) програми полягає в тому, що управління може бути передано тільки головній програмі. Підпрограма може бути відкомпільована, але не може бути запущена на виконання.

ПІДПРОГРАМИ

Переваги від використання підпрограм:

- Можливість створення досить великих програм (обмеження – близько 50000 рядків. Розробка програм практично необмеженого розміру вимагає застосування класів. Мова про них піде далі).
- Досить просто повторно використовувати раніше написаний код.
- Розробку окремих частин програми, тобто підпрограм, можна доручити різним людям.
- Скорочується термін розробки програми в цілому за рахунок повторного використання коду і завдяки можливості залучення до програмного проекту цілої групи програмістів.
- Підвищується надійність програми, тому що підпрограми, як правило, не великі, їх можна досконально вивчити, до того ж повторне використання в нових програмах підпрограм, які вже застосовувалися раніше і не давали «збоїв», підвищує надійність нової програми в цілому.
- У ряді випадків зменшується розмір програми: якщо, наприклад, для сортування п'яти масивів, що використані в програмі, застосована одна і та ж підпрограма, а не пишеться практично один і той же код для кожного масиву окремо, то економія буде суттєвою. Звичайно, якщо підпрограма використовується в програмі тільки один раз, то розмір програми в цілому не тільки не скорочується, а навпаки, зростає.

Недоліки від застосування підпрограм:

-Використання підпрограм завжди зменшує швидкість роботи програми. Це стає помітним, коли розмір підпрограми занадто малий, наприклад, один-два оператора.

-- Грамотне застосування підпрограм вимагає більш високої кваліфікації від програміста, ніж робота без підпрограм.

ВИДИ ПІДПРОГРАМ (ПРОЦЕДУРИ І ФУНКЦІЇ)

Існує дві категорії підпрограм: процедури і функції.

Процедура – це підпрограма, яка не повертає через своє ім'я результату роботи, тому вона викликається як окремий оператор. Процедура «спілкується із зовнішнім світом» через список параметрів.

Функція – це підпрограма, яка через ім'я повертає результат своєї роботи. Функція викликається в вираженні, а не як окремий оператор. Через список параметрів вона може отримати вхідні дані і повернути результати роботи.

Традиційно підпрограму оформляють у вигляді функції, якщо результатом роботи є одиночний об'єкт (число, символ, рядок). Приклад алгоритмів, які зручно оформити функцією: обчислення значення певного інтеграла, знаходження мінімального числа в масиві, підрахунок кількості прогалин у рядку і т.д.

Якщо результатом роботи підпрограми є кілька об'єктів, або вихідними даними є масив, то необхідно використовувати процедуру, так як через ім'я підпрограми можна повернути тільки один простий об'єкт (число, адреса, символ), а через список параметрів – будь-яка кількість і одиночних і складених об'єктів .

ФУНКЦІЇ НА МОВІ С/С ++

Формально в мовах С/С ++ немає процедур, а є тільки функції. Якщо ж необхідна саме процедура, то досить створити функцію, яка повертає тип **void**.

Розглянемо яким чином застосувати свою функцію в програмі на мові С ++. Щоб використовувати в програмі свою власну функцію, необхідно виконати три дії:

- задати прототип функції;
- викликати функцію в необхідному місці, наприклад, в функції main ()
- дати визначення функції.

Розберемося трохи докладніше з тим, що ми перерахували.

```
void ім'яФункції (параметри, якщо необхідні)
{
    код (тіло функції); // один або кілька операторів
}
```

СИНТАКСИС ФУНКЦІЇ НА МОВІ С/С ++

Прототип функції – це заголовок функції, який закінчується крапкою з комою. Прототипи всіх функцій, які використовуються в програмі, **зазвичай записують на початку тексту програми** до визначення функцій. Прототип необхідний для того, щоб у компілятора була повна інформація про тип результату роботи функції і про список параметрів, переданих у функцію. Коли в тексті програми зустрінеться звернення до функції, то компілятор перевіряє правильність її виклику, звіряючись з інформацією з прототипу.

Для програміста прототипи теж корисні: можна в стислій формі відразу побачити, які функції використовуються в програмі, які параметри їм необхідні і т.д.

```
// Прототип функції, що приймає два параметри типу int та повертає значення типу do
double calculateAverage(int a, int b);
```

ФУНКЦІЇ НА МОВІ С/С ++

```cpp
#include <iostream>

// Прототип функції
void printMessage(std::string message);

int main() {
    std::string message = "Привіт, світ!";

    // Виклик функції printMessage()
    printMessage(message);

    return 0;
}

// Визначення функції printMessage()
void printMessage(std::string message) {
    std::cout << message << std::endl;
}
```

Виклик функції можливий у будь-який функції, де це буде потрібно. Якщо тип результату функції – `void`, то виклик записується окремим оператором, в інших випадках – у виразі того ж типу, що і тип результату функції. Мови С/С ++ дозволяють записати виклик функції у вигляді окремого оператора навіть в тому випадку, коли вона повертає тип результату, відмінний від `void`. Наприклад, оператор `sin(x);` абсолютно законний, хоча навряд чи варто так робити.

Визначення функції складається з її заголовка і тіла, записаного у вигляді блоку. Припустимо записувати визначення функцій в будь-якій послідовності. Не можна визначати функцію всередині іншої функції (С/С++).

ПІДПРОГРАМИ

```
#include <iostream>

// Прототип функції
double calculateAverage(int a, int b);

int main() {
    int x = 5;
    int y = 10;
    double avg;

    // Виклик функції calculateAverage()
    avg = calculateAverage(x, y);

    std::cout << "Середнє значення " << x << " та "

    return 0;
}

// Визначення функції calculateAverage()
double calculateAverage(int a, int b) {
    return (a + b) / 2.0;
}
```

У цьому прикладі ми використовуємо прототип функції "calculateAverage", який ми оголосили перед функцією "main". У функції "main" ми оголосили дві змінні "x" та "y", та викликаємо функцію "calculateAverage" з цими змінними як параметрами. Після цього результат повертається до змінної "avg", і ми виводимо результат на екран.
Для того, щоб цей код працював правильно, ми повинні оголосити прототип функції "calculateAverage" перед тим, як викликати її у функції "main". У цьому прикладі ми оголосили прототип функції перед функцією "main". Таким чином, компілятор зможе правильно обробляти виклик функції

ФОРМАЛЬНІ І ФАКТИЧНІ ПАРАМЕТРИ

У роботі з підпрограмами доводиться мати справу з двома видами параметрів: список фактичних параметрів і список формальних параметрів.

Список параметрів (фактичних або формальних – все одно) записується в круглих дужках відразу за ім'ям функції. В принципі, список параметрів може бути порожнім, але дужки за ім'ям функції обов'язкові!

Список фактичних параметрів – це ті реальні дані, за допомогою яких можна налаштувати алгоритм підпрограми на обробку конкретних даних. Фактичні параметри вказуються в списку параметрів при виклику функції і передаються в цю функцію.

Список формальних параметрів – це по суті набір вимог, які пред'являє підпрограма до переданих в неї даними. Список фактичних параметрів записується в заголовку функції при її визначенні в дужках. Для кожного параметра необхідно вказати тип і ім'я. При необхідності задається і спосіб передачі. Імена формальних параметрів локалізовані в підпрограмі і доповнюють перелік локальних об'єктів цієї підпрограми. Так, в нашій функції `fmax()` формальні параметри `x` і `y` типу `double` доповнюють список локальних змінних, що складається з однієї змінної `max` типу `double`. У результаті функція `fmax()` має три локальних змінних, відомих тільки в цій функції.

ПІДПРОГРАМИ

```cpp
#include <iostream>

// Прототип функції
void multiplyNumbers(int x, int y);

int main() {
    int a = 4;
    int b = 6;

    // Виклик функції multiplyNumbers()
    multiplyNumbers(a, b);

    return 0;
}

// Визначення функції multiplyNumbers()
void multiplyNumbers(int x, int y) {
    int result = x * y;
    std::cout << "Результат множення: " << result
}
```

У цьому прикладі ми оголосили функцію "multiplyNumbers", яка множить два числа та виводить результат на екран. У функції "main" ми оголосили дві змінні "a" та "b", які ми передаємо у функцію "multiplyNumbers" як фактичні параметри при її виклику.

У визначенні функції "multiplyNumbers" ми оголосили два формальних параметри "x" та "y", які приймають значення фактичних параметрів, переданих у функцію.

Отже, у цьому прикладі фактичні параметри - це змінні "a" та "b", передані у функцію при її виклику, а формальні параметри - це змінні "x" та "y", оголошені у визначенні функції. Фактичні параметри передаються у функцію як значення, тоді як формальні параметри - це змінні, що приймають ці значення.

СПОСОБИ ПЕРЕДАЧІ ДАНИХ В ФУНКЦІЇ

У мові С ++ дані в підпрограму можна передавати трьома способами: **за значенням, за адресою та за посиланням**. У мові С допустимі тільки два способи: за значенням і за адресою.

Передача даних за значенням

Цей спосіб передачі даних в підпрограму є основним і діє по-замовчуванню. Фактичний параметр обчислюється в викликаючій функції і його значення передається на місце формального параметра в функції, що викликається. На цьому зв'язок між фактичним і формальним параметрами припиняється.

В якості фактичний параметр можна використовувати константу, змінну або більш складний вираз. Передача даних за значенням придатна тільки для простих даних, які є вхідними параметрами. Якщо параметр є вихідним даним або масивом, то передача його в функцію за значенням не прийнятна.

ПЕРЕДАЧА ДАНИХ ЗА АДРЕСОЮ

```cpp
#include <iostream>

// Функція для обміну значеннями двох змінних за адресами
void swap(int* x, int* y)
{
    int temp = *x;
    *x = *y;
    *y = temp;
}

int main()
{
    int a = 5, b = 10;

    std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;

    // Виклик функції swap і передача адрес змінних a та b
    swap(&a, &b);

    std::cout << "After swap: a = " << a << ", b = " << b << std::endl;

    return 0;
}
```

За адресою в функцію завжди передаються масиви (розглянемо це в наступних темах). Для масиву це взагалі єдиний спосіб передачі даних в мовах C/C ++. Так само за адресою можна передати ті прості об'єкти, які є вихідними даними (або вхідними та вихідними одночасно).

У разі передачі даних за адресою фактичний параметр може бути тільки змінної (константа або вираз не мають адреси!).

У цьому прикладі функція **swap** отримує вказівники на змінні x та y і обмінює їх значення за допомогою операції розіменування (*x та *y). У головній функції main викликається функція **swap**, передаючи їй адреси змінних a та b за допомогою оператора **&**. Після виклику функції значення a та b будуть обмінені, і вони виводяться на екран

ПЕРЕДАЧА ДАНИХ ПО ПОСИЛАННЮ

```cpp
#include <iostream>

// Функція для обміну значеннями двох змінних по посиланню
void swap(int& x, int& y)
{
    int temp = x;
    x = y;
    y = temp;
}

int main()
{
    int a = 5, b = 10;

    std::cout << "Before swap: a = " << a << ", b = " << b << std::endl;

    // Виклик функції swap і передача змінних a та b по посиланню
    swap(a, b);

    std::cout << "After swap: a = " << a << ", b = " << b << std::endl;

    return 0;
}
```

Це ще один із способів повернути результат роботи функції через список параметрів. Нагадаємо, що застосовується тільки для С ++. У мові С такого варіанту немає.

При передачі даних по посиланню в функцію, куди передаються дані, створюються синоніми вихідних об'єктів. Тому робота в підпрограмі ведеться саме з вихідними об'єктами. Якщо в підпрограмі посилальна змінна змінить значення, то це відразу позначиться на вихідної змінної.

У викликаючій функції параметр, який передається по посиланню, може бути тільки простою змінної будь-якого відомого типу.

Повернемося знову до прикладу обміну, тільки дані передамо по посиланню.

У цьому прикладі функція swap отримує змінні x та y по посиланню за допомогою символу &. Функція обмінює їх значення тим самим способом, що і в попередньому прикладі, але без використання вказівників. У головній функції main викликається функція swap, передаючи їй змінні a та b по посиланню. Після виклику функції значення a та b будуть обмінені, і вони виводяться на екран.

Лекція №11

Підпрограми (процедури і функції)

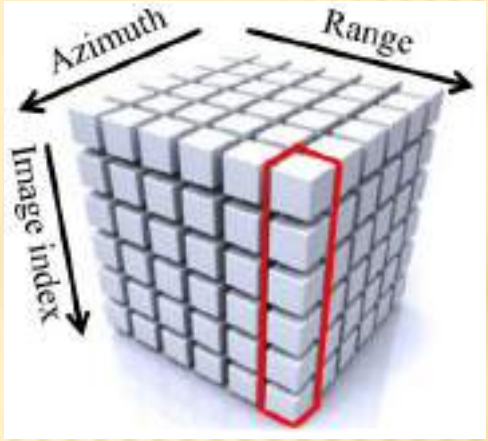
Лекція №12

Структури

СТРУКТУРИ. ВИКОРИСТАННЯ СТРУКТУР

У програмуванні є багато випадків, коли може знадобитися більше однієї змінної для представлення об'єкта. Наприклад, щоб представити самого себе, ви, швидше за все, захочете вказати своє ім'я, день народження, зріст, вага або будь-яку іншу інформацію. наприклад:

```
string myName;
int myBirthDay;
int myBirthMonth;
int myBirthYear;
int myHeight;
int myWeight.
```



Але тепер у вас 6 окремих незалежних змінних. Якщо ви захочете передати інформацію про себе в функцію, вам доведеться передавати кожну змінну окремо. Крім того, якщо ви захочете зберігати інформацію про когось ще, вам доведеться оголосити ще 6 змінних для кожної людини додатково! Неозброєним оком видно, що така реалізація не дуже ефективна.

С ++ дозволяє створювати власні призначені для користувача типи даних. Це типи, які групують кілька окремих змінних разом. Одним з найпростіших користувацьких типів даних є структура. Структура (struct) дозволяє групувати змінні різних типів в єдине ціле.

СТРУКТУРИ. ВИКОРИСТАННЯ СТРУКТУР

Оголошення і визначення структур

Правило. Оголошення структурної змінної здійснюється в 2 етапи:

- оголошення шаблону структури як нового типу даних. На цьому етапі пам‘ять не виділяється. Формується тільки інформація про вміст структури;

- оголошення самої змінної. На цьому етапі виділяється пам‘ять для кожного поля (змінної), що описується в шаблоні структури.

Синтаксис оголошення структури:

```
struct <ім'я> {
<Тип1> <поле1>;
<Тип2> <поле2>;
...
<ТипN> <полеN>;};
```

Оскільки структури визначаються користувачем, то спочатку ми повинні повідомити компілятору, як виглядає наша структура, перш ніж її використовувати. Для цього оголосимо її, використовуючи ключове слово struct. наприклад:

```
struct Employee
{ short id;
  int age;
double salary;};
```

СТРУКТУРИ. ВИКОРИСТАННЯ СТРУКТУР

Ми визначили структуру з ім'ям ***Employee***. Вона містить 3 змінні:

id типу **short**,

age типу **int**

salary типу **double**.

Ці змінні, які є частиною структури, називаються членами (або полями). ***Employee*** – це просто оголошена структура, хоч ми і повідомляємо компілятору, що вона має змінні-члени, пам'ять під неї зараз не виділяється. Імена структур прийнято писати з великої літери, щоб відрізняти їх від імен змінних.

Щоб використовувати структуру ***Employee***, ми просто оголосимо змінну типу ***Employee***:

```
Employee john // ім’я структури Employee починається з великої літери,
              //а змінної john з маленької
```

Тут ми визначили змінну типу ***Employee*** з ім'ям *john*. Як і у випадку зі звичайними змінними, визначення змінної структури призведе до виділення пам'яті під неї.

ПРИКЛАД ВИКОРИСТАННЯ СТРУКТУР

```cpp
#include <iostream>
#include <string>

using namespace std;

struct User {
    string firstName;
    string lastName;
    int age;
    string email;
};

int main() {
    User user1;
    user1.firstName = "John";
    user1.lastName = "Doe";
    user1.age = 30;
    user1.email = "johndoe@example.com";

    cout << "User information: " << endl;
    cout << "Name: " << user1.firstName << " " << user1.lastName << endl;
    cout << "Age: " << user1.age << endl;
    cout << "Email: " << user1.email << endl;

    return 0;
}
```

СТРУКТУРИ. ВИКОРИСТАННЯ СТРУКТУР

Ім'я структурної змінної може бути вказано при оголошенні структури. У цьому випадку воно розміщується після закриваючої фігурної дужки}. Область видимості такої структурної змінної буде визначатися місцем опису структури.

```
struct complex_type {
    double real;
    double imag;
} Number;
```

СТРУКТУРИ. ВИКОРИСТАННЯ СТРУКТУР

struct Book { char title[70]; char author[50]; int year; float price;};

Для змінної типу структура виділяється пам'ять розмір якої можна визначити застосувавши функцію sizeof():

```
Book B;
int size;
size = sizeof(B);
// буде виділено 128 б. (70+50+4+4=128)
```

```cpp
#include <iostream>

using namespace std;

// Оголошення структури Book
struct Book {
    char title[70];
    char author[50];
    int year;
    float price;
};

int main() {
    // Оголошення змінної B типу Book
    Book B;

    // Визначення розміру змінної B
    int size = sizeof(B);

    // Виведення розміру структури
    cout << "Розмір структури Book: " << size << " байтів" << endl;

    return 0;
}
```

ДОСТУП ДО ЧЛЕНІВ СТРУКТУР

Коли ми визначаємо змінну, таку як ***Employee*** john, то john посилається на всю структуру. Для того, щоб отримати доступ до окремих елементів-полях, використовується оператор вибору члена (крапка). Нижче наведено приклад використання оператора вибору членів для ініціалізації кожного поля структури:

```
Employee john; // створюємо структуру Employee для John
john.id = 8; // присвоюємо значення полю id всередині структури john
john.age = 27; // присвоюємо значення полю age всередині структури john
john.salary = 32.17; // присвоюємо значення полю salary всередині структури

Employee james; // створюємо структуру Employee для James
james.id = 9; // присвоюємо значення полю id всередині структури james
james.age = 30; // присвоюємо значення полю age всередині структури james
james.salary = 28.35; // присвоюємо значення полю salary всередині структури
```

СТРУКТУРИ. ВКЛАДЕНІ СТРУКТУРИ

В якості полів можуть виступати і структури. Такі конструкції називають вкладеними структурами, наприклад:

```
struct Employee
{   short id;
    int age;
    double salary;};

struct Company
{   Employee CEO; // Employee - це структура всередині структури Company
    int numberOfEmployees;};

Company myCompany; // опис змінної myCompany типу структура
```

СТРУКТУРИ. ВКЛАДЕНІ СТРУКТУРИ

У цьому прикладі ми створили структуру Point3D, яка містить в собі вкладену структуру Coord, що містить три координати x, y, z. Ми також створили змінну p типу Point3D та ініціалізували її значенням { {1, 2, 3} }. На екран будуть виведені координати цієї точки, які ми отримали доступом до вкладеної структури через оператор крапки.

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

struct Point3D {
    struct Coord {
        int x, y, z;
    } coordinates;
};

int main() {
    Point3D p = {{1, 2, 3}}; // створення точки з координатами (1, 2, 3)
    cout << "Координати точки: " << p.coordinates.x << ", " << p.coordinates.y
    return 0;
}
```

СТРУКТУРИ. МАСИВИ СТРУКТУР

З структур можна створювати масиви також, як масиви інших типів. І все формати визначення масиву структур будуть аналогічні визначенням масивів інших типів:

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Student {
    string name;
    int age;
    double grade;
};

int main() {
    const int SIZE = 3; // розмір масиву структур
    Student students[SIZE] = { // ініціалізація масиву структур
        {"John", 21, 3.5},
        {"Mary", 20, 3.8},
        {"Bob", 19, 3.2}
    };

    // виведення даних про студентів
    for (int i = 0; i < SIZE; i++) {
        cout << "Student " << i+1 << ": " << endl;
        cout << "Name: " << students[i].name << endl;
        cout << "Age: " << students[i].age << endl;
        cout << "Grade: " << students[i].grade << endl;
        cout << endl;
    }

    return 0;
}
```

СТРУКТУРИ. ПОКАЖЧИКИ НА СТРУКТУРУ

Для отримання адреси структурної змінної слід помістити оператор & перед ім'ям структури. Нехай є наступний фрагмент

```cpp
#include <iostream>
#include <string>
using namespace std;

struct Person {
    string name;
    int age;
};

int main() {
    Person person = {"John", 25}; // створення структури Person з даними
    Person* ptrPerson = &person; // створення покажчика на структуру Person

    // виведення даних про персону за допомогою покажчика
    cout << "Name: " << ptrPerson->name << endl;
    cout << "Age: " << ptrPerson->age << endl;

    return 0;
}
```

У цьому прикладі ми створили структуру Person, яка містить поля name (ім'я персони) та age (вік персони). Далі ми створили змінну person типу Person та ініціалізували її значеннями. Потім ми створили покажчик ptrPerson на структуру Person та присвоїли йому адресу змінної person.

Для виведення даних про персону ми використали оператор крапки (->), який дозволяє отримати доступ до полів структури через покажчик. Таким чином, ми вивели дані про персону за допомогою покажчика.

Цей приклад показує, як можна використовувати покажчики на структури для отримання доступу до даних про об'єкти певного типу та їх подальшої обробки.

СТРУКТУРИ І ФУНКЦІЇ

Великою перевагою використання структур, ніж окремих змінних, є можливість передавати всю структуру функції, яка повинна працювати з її полями.

Існує 2 способи передачі структури у функцію:

- передача структури за значенням. При такій передачі робиться копія структурної змінної в пам'яті. Якщо структура має великий розмір, то такий спосіб неефективний. Перевага цього способу в тому, що всі маніпуляції з копією структури у Функції НЕ впливають на вихідну змінну;

- передача покажчика на структуру. У цьому випадку передається тільки покажчик на структуру. Якщо структура займає великий об'єм пам'яті, то такий спосіб забезпечує швидку передачу значень структурної змінної у функцію. Це пов'язано з тим, що НЕ робиться в пам'яті копії структурної змінної. Недоліком цього способу є те, що у функції можна змінити значення вихідної структурної змінної, коли цього НЕ потрібно робити.

Так само, як і при передачі структури у функцію, існують 2 способи повернення:

повернення структури за значенням;

повернення покажчика на структуру.

ПЕРЕДАЧА СТРУКТУРИ У ФУНКЦІЮ ЗА ЗНАЧЕННЯМ

```cpp
#include <iostream>
#include <string>

using namespace std;

struct Student {
    string name;
    int age;
    float gpa;
};

void printStudentInfo(Student student) {
    cout << "Name: " << student.name << endl;
    cout << "Age: " << student.age << endl;
    cout << "GPA: " << student.gpa << endl;
}

int main() {
    Student john = {"John Smith", 20, 3.5};
    printStudentInfo(john);

    return 0;
}
```

У цьому прикладі структура Student передається у функцію printStudentInfo за значенням. У функції створюється копія структури, яку можна маніпулювати без впливу на вихідну структуру, передану у функцію. Функція виводить інформацію про студента на екран. У функції main створюється об'єкт john структури Student, який передається у функцію printStudentInfo за значенням.

ПЕРЕДАЧА ПОКАЖЧИКА НА СТРУКТУРУ У ФУНКЦІЮ

```cpp
#include <iostream>
#include <string>

using namespace std;

struct Student {
    string name;
    int age;
    float gpa;
};

void updateStudentAge(Student* studentPtr, int newAge) {
    studentPtr->age = newAge;
}

int main() {
    Student john = {"John Smith", 20, 3.5};
    updateStudentAge(&john, 21);

    cout << "Name: " << john.name << endl;
    cout << "Age: " << john.age << endl;
    cout << "GPA: " << john.gpa << endl;

    return 0;
}
```

У цьому прикладі покажчик на структуру Student передається у функцію updateStudentAge. У функції використовується оператор ->, щоб дістатися до поля age структури за допомогою покажчика. Функція змінює вік студента на нове значення, передане як другий параметр. У функції main створюється об'єкт john структури Student, і покажчик на цей об'єкт передається у функцію updateStudentAge разом з новим віком. Після виклику функції вік студента змінюється на 21.

Лекція №12

Структури

Лекція №13

Графіка C++. Windows API для малювання

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ

У C++ для малювання можна застосовувати бібліотеку Windows. Але ця бібліотека призначена в основному для роботи у текстовому режимі.

Щоб використовувати такі функції ви повинні включити в програму заголовний файл **windows.h**, завдяки якому можна отримати доступ до великої кількості функцій, що становлять бібліотеку Windows API.

Всі графічні функції C / C ++ в Windows використовують спеціальну структуру, яка називається контекстом вікна. Ви повинні знати, що вона зберігає поточні параметри області, в якій ми малюємо: колір ліній, їх товщину і стиль (пунктир, суцільний, ...), колір заливки областей, розміри вікна і т.д. У програмах на C / C ++ контекст вікна зберігається в змінній типу **HDC**, яка повинна бути пов'язана з вікном, в якому відбувається малювання. Найпростіша команда, яка створює таку змінну, має вигляд

```
HDC hdc = GetDC (GetConsoleWindow ());
```

Зараз не будемо говорити про її сенс – просто включайте цей рядок в початок функції **main** (), а повернутий нею дескриптор використовуйте в усіх графічних функціях. Будь-яка функція WINDOWS, яка щось малює, першим параметром буде приймати змінну hdc. Перед завершенням програми дескриптор контексту консольного вікна рекомендується звільнити командою **ReleaseDC** (NULL, hdc);

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ

```cpp
#include <windows.h>

int main()
{
    // Отримання дескриптора вікна консолі
    HWND consoleWindow = GetConsoleWindow();

    // Отримання дескриптора контексту пристрою вікн
    HDC hdc = GetDC(consoleWindow);

    // Малювання прямокутника
    Rectangle(hdc, 100, 60, 180, 160);

    // Звільнення контексту пристрою вікна консолі
    ReleaseDC(consoleWindow, hdc);

    return 0;
}
```

1. В першому рядку підключається заголовочний файл windows.h, який надає доступ до функцій та констант Windows API.

2. У функції main() отримується дескриптор вікна консолі за допомогою функції **GetConsoleWindow().**

Дескриптор вікна консолі - це унікальний ідентифікатор, що використовується для доступу до вікна консолі в операційній системі Windows. Кожен віконний об'єкт в Windows має свій власний дескриптор, який використовується для звернення до нього з програми.

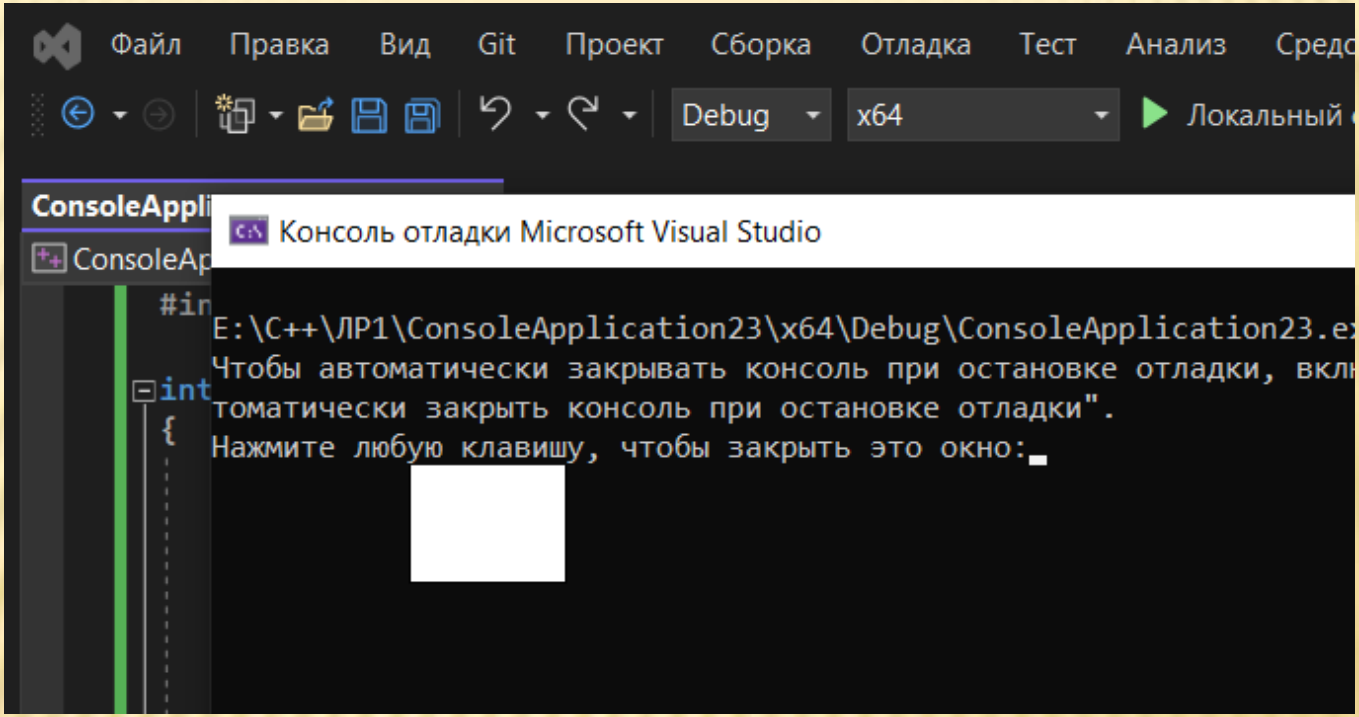
3. Потім отримується дескриптор контексту пристрою вікна консолі за допомогою функції **GetDC()**, якій передається дескриптор вікна консолі.

Дескриптор контексту пристрою вікна консолі - це об'єкт, який забезпечує доступ до графічних ресурсів вікна консолі в операційній системі Windows.

Після цього малюється прямокутник за допомогою функції Rectangle(), якій передається дескриптор контексту пристрою вікна консолі та чотири цілочисельні параметри, що представляють координати прямокутника.

На завершення контекст пристрою вікна консолі звільняється за допомогою функції ReleaseDC().

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ



МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ. ПЕРО (PEN)

Коли ми малюємо лініями, то вони можуть мати різні параметри: товщину, стиль і т.д., тому спочатку необхідно визначити «перо» яким буде виконуватись малювання.

Поточне перо є частиною структури контексту вікна. Щоб лінію намалювати необхідним пером, спочатку ви повинні створити об'єкт типу HPEN

HPEN Pen = *CreatePen* (PS_SOLID, 3, RGB (255,0,0));

Перший аргумент функції ***CreatePen*** визначає чи буде лінія суцільний (PS_SOLID), пунктирною, штрих – пунктирною і т.д. Можна використовувати значення PS_DOT, PS_DASH, PS_DASHDOT і деякі інші.

Другий аргумент визначає товщину лінії.

Третій – задає колір. аргументи макросу RGB є цілими числами зі значеннями від 0 до 255 і задають колір, складений з відтінків / яркостей трьох базових кольорів: червоного (Red), зеленого (Green) і синього (Blue).

Після створення пера (HPEN) його потрібно завантажити в контекст (HDC) функцією ***SelectObject*** (HDC, HPEN). Після цього всі лінії будуть малюватися створеним пером до тих пір, поки в контексті не буде завантажено нове перо

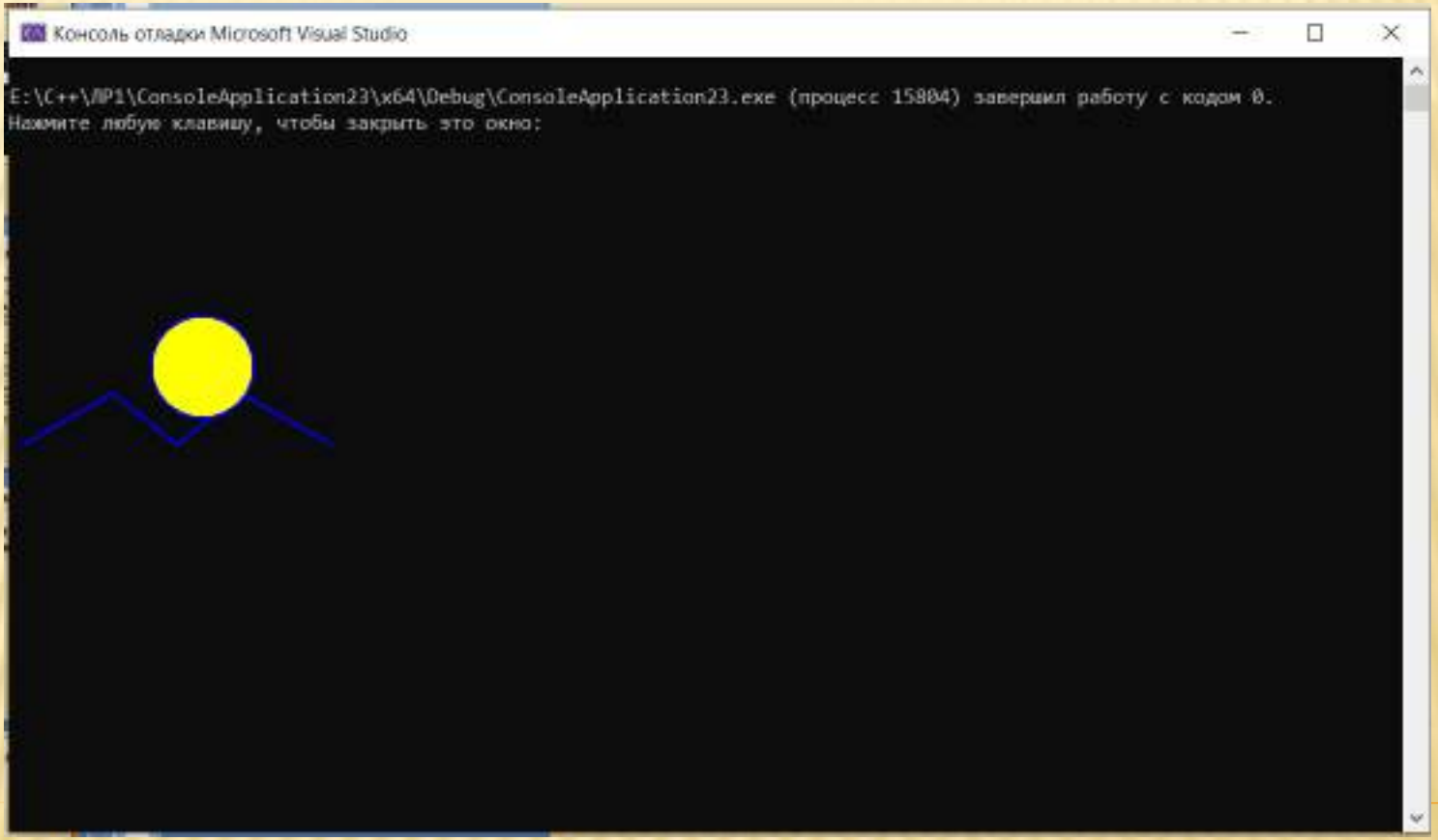
Є ще один момент. В контексті зберігається положення так званого поточного покажчика, який містить координати точки від якої наступна графічна функція починає малювати свою фігуру. При запуску програми цей покажчик встановлюється в точку (0,0). Його місце розташування на екрані невидимо, він означає лише позицію у вікні для деяких функцій.

Не всі графічні функції використовують цей покажчик. Функція ***MoveToEx*** ((HDC, X, Y, NULL); переносить поточну позицію покажчика в точку (X, Y) і не запам'ятовує стару позицію (якщо останній аргумент дорівнює NULL).

МАЛЮВАННЯ У КОНСОЛЬНОМ У ВІКНІ

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

using namespace std;

int main()
{
    HWND hwnd = GetConsoleWindow(); // отримуємо вікно консолі
    HDC hdc = GetDC(hwnd); // отримуємо контекст пристрою вікна консолі

    // створюємо перо з синім кольором
    HPEN hPen = CreatePen(PS_SOLID, 2, RGB(0, 0, 255));
    SelectObject(hdc, hPen); // встановлюємо перо

    MoveToEx(hdc, 10, 300, NULL); // переміщаємо вказівник до початкової точки
    LineTo(hdc, 80, 260); // малюємо першу лінію
    LineTo(hdc, 130, 300); // малюємо другу лінію
    LineTo(hdc, 180, 260); // малюємо третю лінію
    LineTo(hdc, 250, 300); // малюємо четверту лінію

    // створюємо кисть з жовтим кольором
    HBRUSH hBrush = CreateSolidBrush(RGB(255, 255, 0));
    SelectObject(hdc, hBrush); // встановлюємо кисть
    Ellipse(hdc, 110, 200, 190, 280); // малюємо круг

    // звільняємо ресурси
    DeleteObject(hPen);
    DeleteObject(hBrush);
    ReleaseDC(hwnd, hdc);
```

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ. ПЕРО (PEN)



МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ. ПЕРО (PEN)

Графічна область вікна являє собою масив пікселів. Кожен піксель відповідає одній точці на екрані і може мати свій колір. Встановити колір пікселя в точці екрану з координатами (X, Y) можна з допомогою функції

SetPixel (HDC dc, int X, int Y, COLORREF Color);

Тут ***COLORREF*** представляє спеціальний тип (структуру), яку створює макрос RGB. Функція **SetPixel** відображає точку заданого кольору за вказаними координатами і повертає колір, який був встановлений в дійсності (іноді дисплей не в змозі точно відтворити заданий колір).

```
// малюємо пікселі
for (int x = 0; x < 320; x++)
{
    for (int y = 0; y < 240; y++)
    {
        SetPixel(hdc, x, y, RGB(255, 0, 0));
    }
```

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ

Функції малювання

LineTo (hDC, int x, int y); – малює лінію від поточної позиції до точки, зазначеної в аргументах, і переводить покажчик позиції в цю точку.

Ar **(hDC, int left, int top, int right, int bottom, int x1, int y1, int x2, int y2);** малює дугу еліпса / окружності. Аргументи left, top і right, bottom визначають лівий верхній і правий нижній кути прямокутника, в який вписаний еліпс. Наступні значення – координати точок, від яких будуть проведені прямі до центру еліпса. У місцях їх перетину з еліпсом, починається і закінчується дуга (самі прямі не малюються). Дуга малюється в напрямку проти годинникової стрілки. Використані позначення пояснені на наступному малюнку.

ArcTo(hDC, int left, int top, int right, int bottom, int x1, int y1, int x2, int y2); використовує і оновлює положення поточного покажчика. Спочатку від поточного покажчика до початку дуги функція малює відрізок, потім малює дугу і встановлює поточний покажчик в її кінцеву точку.

AngleArc**(HDC hdc, int X, int Y, DWORD R, FLOAT startAngle, FLOAT sweepAngle);** – малює дугу окружності і відрізок від поточної позиції до початку цієї дуги. Дуга є частиною кола радіуса R з центром в точці X, Y. Дуга задається двома кутами, вимірюваними в градусах і відлічуваними проти годинникової стрілки від позитивного напрямку осі X. Параметр startAngle задає початок дуги, а sweepAngle її кутову міру. Функція AngleArc починає малювати від положення поточного покажчика і встановлює його в кінцеву точку своєї дуги. Дуга малюється поточним пером.

МАЛЮВАННЯ У КОНСОЛЬНОМУ ВІКНІ

Polyline*(hdc, POINT * pt, int cPoints);** – малює ламану, що проходить через точки, координати яких знаходяться в масиві ***pt. Структура POINT визначена в бібліотеці у такий спосіб.
Функція Polyline не використовує і не оновлює покажчик поточної позиції.

***Polyline*To (hdc, POINT * pt, int cPoints);** – використовує і оновлює покажчик поточної позиції. ламана починає малюватися від поточного положення покажчика до першої точки масиву pt, і далі проходить через інші точки цього масиву. функція переводить покажчик поточної позиції в кінцеву точку ламаної.

PolyPolyline (HDC hdc, constPOINT * lppt, constDWORD * lpdwPolyPoints, DWORD cCount); – малює відразу кілька ламаних. Другий аргумент приймає масив координат вершин всіх шматків ламаної, третій аргумент lpdwPolyPoints - це масив кількостей вершин в кожному відрізку, четвертий передає кількість відрізків.

***TextOut* (HDC hdc, int xStart, int yStart, LPCTSTR lpString, int countChars);** – відображає текстову рядок в заданому місці вікна, використовуючи поточні шрифт, колір фону і символів. Аргументи xStart і yStart визначають точку прив'язки текстового рядка lpString. Рядок не зобов'язана закінчуватися нульовим символом, оскільки аргумент countChars містить кількість символів, яке слід надрукувати. З огляду на, що ми говорили про завершальний символ A функцій Windows, фрагмент виклику цієї функції може мати наступний вигляд

```
char str [] = "Три ламаних";    TextOutA (hdc, 360, 40, str, strlen(str));
```

Функція TextOutA не повинна бути останньою в ланцюжку графічних функцій (інакше текст не відображається).

Щоб змінити колір тексту і колір фону, на якому він відображений, використовують функції **SetBkColor** і **SetTextColor**. Функція **SetTextColor** «малює» текст заданим кольором. Функція **SetBkColor** зафарбовує фон між буквами заданим кольором, а також проміжки пунктирних ліній пера, створюваного функцією **CreatePen**.

МАЛЮВАННЯ ГЕОМЕТРИЧНИХ ФІГУР

Є багато функцій, які малюють геометричні фігури (зафарбовані плоскі області). Але також, як для ліній потрібно спершу призначити перо, для фігур треба встановити пензель, яка буде зафарбовувати їх середину.

Є два способи оголосити пензель. Перший – поставити пензель із суцільною заливкою, другий – вказати пензель із заданим стилем заливки. Для цього існують дві функції: **CreateSoldBrush**() і **CreateHatchBrush**(). після створення пензель повинен бути завантажений в контекст пристрою функцією **SelectObject**. Наприклад, в наступному фрагменті ми створюємо суцільний червоний пензель і робимо його активним.

```
HBRUSH hBrush; // створюємо об'єкт – пензель
hBrush = CreateSolidBrush (RGB (255,0,0)); // створюємо суцільний пензель
SelectObject (hdc, hBrush); // робимо пензель активним
```

Після цього графічні функції, які малюють фігури, будуть зафарбовувати їх суцільним червоним пензлем. Цей пензель активний та тих пір, поки у контексті не буде завантажений інший пензель.

Функція оголошення несуцільного пензля має вигляд:

```
HBRUSH CreateHatchBrush (int fnStyle, RGB (r, g, b));
```

Аргумент fnStyle може набувати таких значень:

HS_DIAGONAL – штрихує по діагоналі
HS_CROSS – клітинка
HS_DIAGCROSS – діагональна сітка
HS_FDIAGONAL – по діагоналі в іншу сторону
HS_HORIZONTAL – горизонтальний "тільник"
HS_VERTICAL – вертикальний "паркан"

МАЛЮВАННЯ ГЕОМЕТРИЧНИХ ФІГУР

Функція **Rectangle** малює зафарбований прямокутник, тип заливки якого визначається поточним пензлем.

Rectangle (hDC, left, top, right, bottom);

Аргументами цієї функції є контекст консолі і координати лівого верхнього і правого нижнього кутів прямокутника. контур цього прямокутника і інших фігур малюється поточним пером.

RoundRect малює округлений прямокутник.

RoundRect (hDC, left, top, right, bottom, width, height);

Перші п'ять параметрів збігаються з параметрами попередньої функції. Для заокруглення кутів використовуються дуги еліпса, ширина і висота якого задаються в аргументах width і height.

Функція **FillRect** зафарбовує прямокутну область вікна заданої пензлем (без контуру). Прямокутник визначається другим аргументом через покажчик на структуру RECT. Пензель, що передається третім аргументом, може задаватися дескриптором кисті або константою, що ідентифікує системний колір.

FillRect (HDC hDC, const RECT * lprc, HBRUSH hbr);

Це може бути корисним для завдання фону консольного вікна, якщо передати їй у другому аргументі розміри консолі.

МАЛЮВАННЯ ГЕОМЕТРИЧНИХ ФІГУР

Функція **FrameRect**(HDC, RECT *, HBRUSH) малює контур навколо прямокутної області, заданої другим аргументом, використовуючи пензель, переданий в третьому аргументі. Товщина контуру завжди дорівнює одній логічній одиниці (зазвичай один піксель).

```
RECT rct = {300,40,400,180};
FrameRect (hdc, & rct, CreateSolidBrush (RGB (255,0,0)));
```

Функція **Ellipse** малює еліпс або коло.

Ellipse (HDC hdc, int left, int top, int right, int bottom); Її аргументи задають координати лівого верхнього і правого нижнього кутів прямокутника, в який вписаний еліпс.

Функція **Chord** малює сегмент (область між дугою еліпса і її хордою). Контур сегмента малюється поточним пером, а його нутро зафарбовується поточним пензлем.

Chord (hDC, int left, int top, int right, int bottom,
int x1, int y1, int x2, int y2);

Аргументи left, top і right, bottom визначають лівий верхній і правий нижній кути прямокутника, в який вписаний еліпс. Наступні значення – координати точок, від яких будуть проведені прямі до центру еліпса. У місцях їх перетину з еліпсом, починається і закінчується дуга.

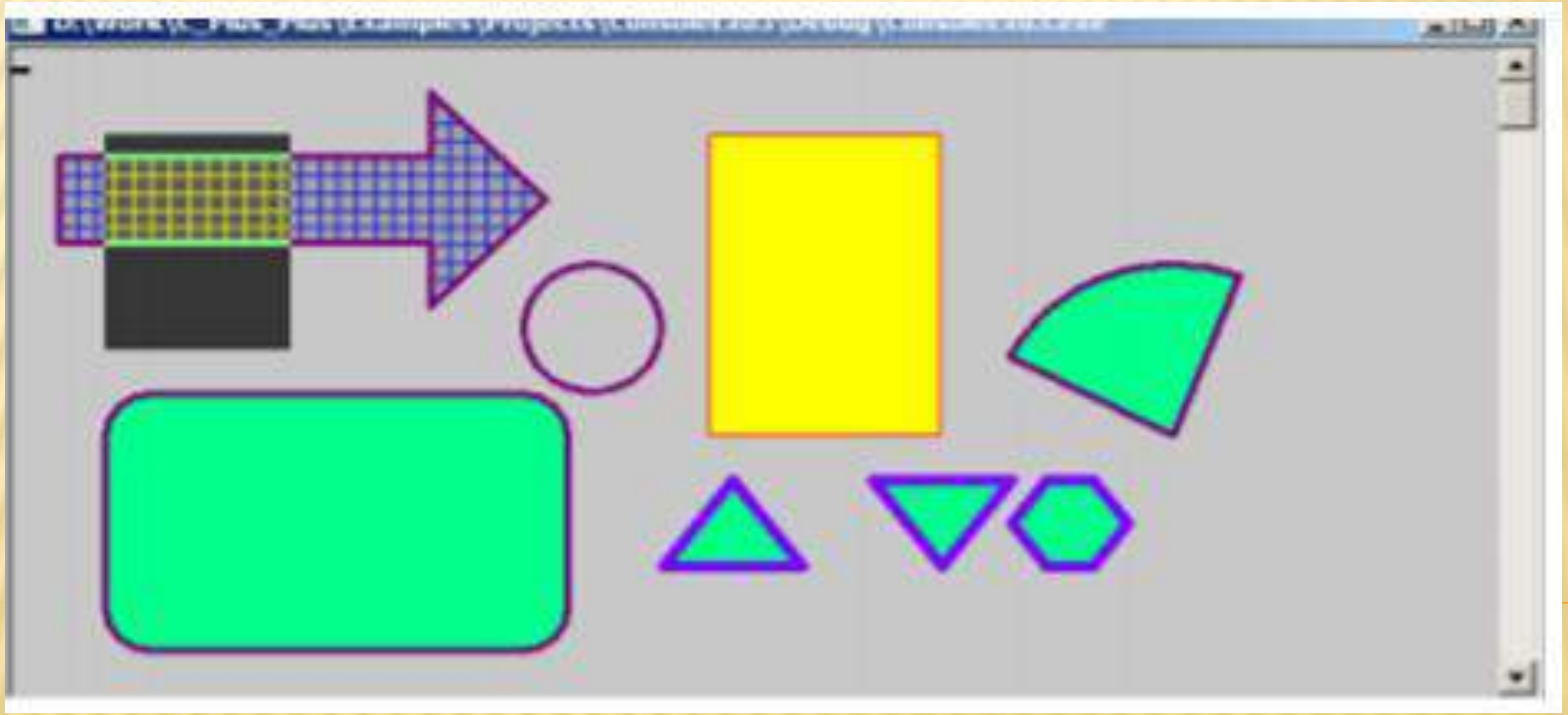
Функція **Pie** малює сектор (область між дугою еліпса і двома її радіальними відрізками). Контур сектора малюється поточним пером, а його внутрішність зафарбовується поточним пензлем.

Pie (hDC, int left, int top, int right, int bottom,int x1, int y1, int x2, int y2);

МАЛЮВАННЯ ГЕОМЕТРИЧНИХ ФІГУР

Функція **PolyPolygon** малює відразу кілька багатокутників, аналогічно тому, що робить функція PolyPolyline.

BOOL PolyPolygon (HDC hdc, const POINT * lppt, const DWORD * lpdwPolyPoints, DWORD cCount)



МАЛЮВАННЯ ГЕОМЕТРИЧНИХ ФІГУР

У Windows API є багато функцій роботи з графічними областями.

Область – це прямокутна, багатокутна, еліптична фігура або комбінація декількох таких фігур. Над її пікселями можна виконувати різні операції. Вона може бути зафарбована, обрамлена контуром, її кольору можуть бути інвертовані, її можна використовуватися для перевірки виконання клацання миші всередині неї.

Щоб працювати з областю, спочатку вона повинна бути створена. Для цього є цілий ряд функцій, наприклад, CreateRectRgn, CreateRoundRectRgn, CreateEllipticRgn, CreatePolygonRgn і ще кілька інших. Вони використовують свої параметри для опису геометрії області і повертають дескриптор області - об'єкт типу HRGN. Після створення області з нею можна виконувати різні операції.

Для створення прямокутної області використовуються функції **CreateRectRgn** і **CreateRectRgnIndirect**.

Функція **CreateRectRgn** створює прямокутну область, використовуючи координати лівого верхнього і правого нижнього кутів прямокутника.

HRGN CreateRectRgn (int left, int top, int right, int bottom);

Лекція №13

Графіка C++. Малювання у консольному вікні

Лекція №14

Вказівники і масиви

Тип даних вказівник

При виконанні ініціалізації змінної, їй автоматично присвоюється вільна адреса в пам’яті, і, будь-яке значення, яке ми присвоюємо змінній, зберігається за цією адресою в пам’яті. Наприклад:

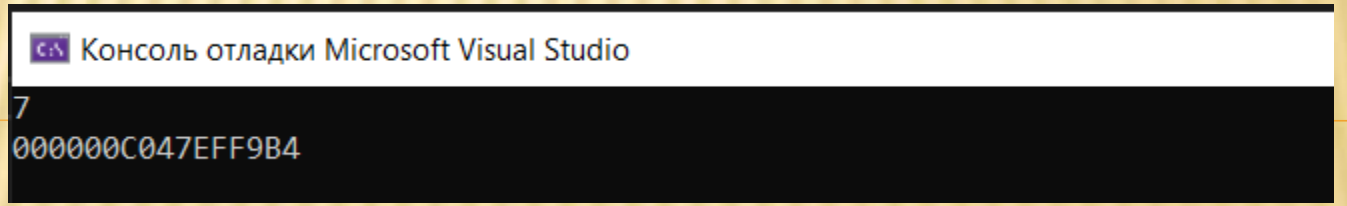
```
int a = 7;
```

При виконанні цієї команди процесором, виділяється частина оперативної пам’яті. В якості прикладу припустимо, що змінній b присвоюється комірка пам’яті під номером 150. Всякий раз, коли програма зустрічає змінну b в виразі вона розуміє, що для того, щоб отримати значення — їй потрібно зазирнути в комірку пам’яті під номером 150.

Хороша новина — нам не потрібно турбуватися про те, які конкретно адреси в пам’яті виділені для певних змінних. Ми просто посилаємося на змінну через присвоєний їй ідентифікатор, а компілятор конвертує цей ідентифікатор у відповідну адресу в пам’яті. Однак цей підхід має деякі обмеження, які ми обговоримо на цій лекції.

Тип даних вказівник

```cpp
#include <iostream>

int main()
{
    int a = 7;
    std::cout << a << '\n'; // виводимо значення змінної a
    std::cout << &a << '\n'; // виводимо адресу в пам'яті змінної a

    return 0;
}
```



Оператор розіменування *

Оператор розіменування * дозволяє отримати значення по вказаній адресі:

```cpp
#include <iostream>

int main()
{
    int a = 7;
    std::cout << a << '\n'; // виводимо значення змінної a
    std::cout << &a << '\n'; // виводимо адресу змінної a
    std::cout << *&a << '\n'; /// виводимо значення комірки в пам'яті змінної a

    return 0;
```

Вказівники

Вказівник (або ***“покажчик”***) — це змінна, значенням якої є адреса комірки в пам’яті. Вказівники оголошуються так само, як і звичайні змінні, тільки із зірочкою між типом даних і ідентифікатором:

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

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

Синтаксично мова C++ приймає оголошення вказівника, коли зірочка знаходиться поруч з типом даних, з ідентифікатором або навіть посередині. Зверніть увагу, ця зірочка НЕ є оператором розіменування. Це всього лише частина синтаксису оголошення вказівника.

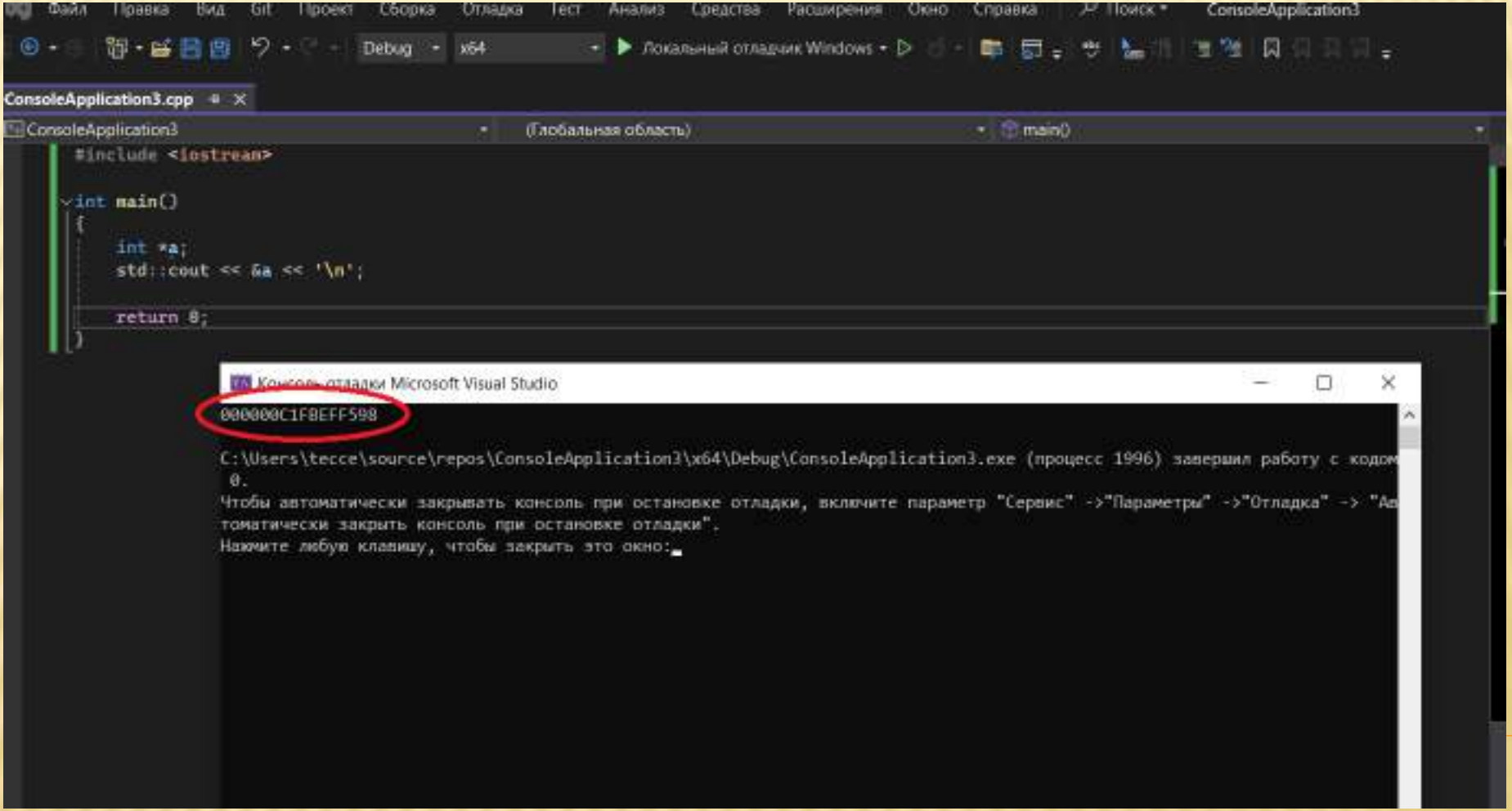
Однак, при оголошенні кількох вказівників, зірочка повинна знаходитися біля кожного ідентифікатора. Це легко забути, якщо ви звикли вказувати зірочку біля типу даних, а не біля імені змінної. Наприклад:

```
int* iPtr3, iPtr4; // iPtr3 - це вказівник на значення типу int
```

З цієї причини, при оголошенні вказівника, рекомендується вказувати зірочку біля імені змінної.

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

Вказівники

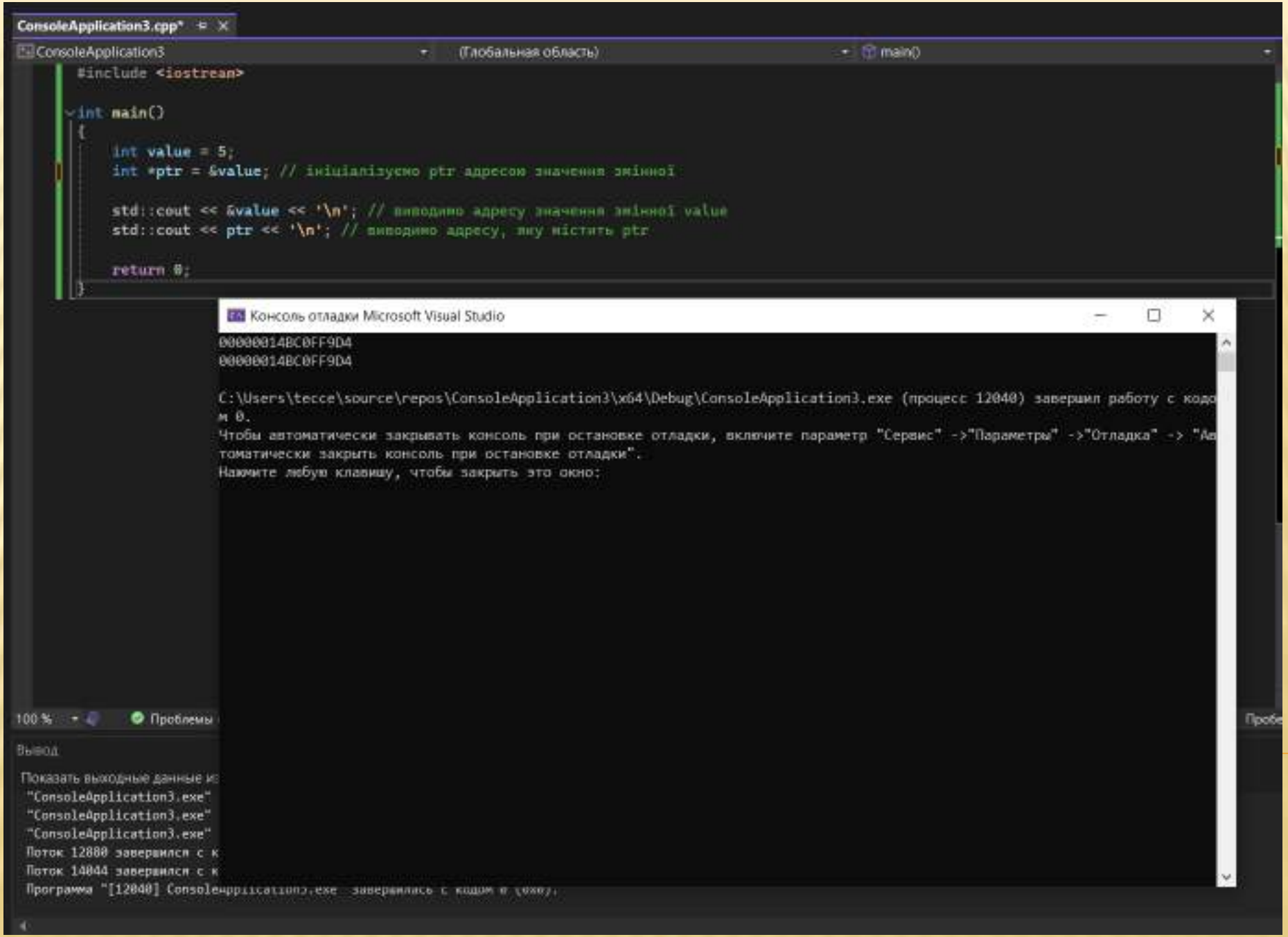

Файл Правка Вид Git Проект Сборка Отладка Тест Анализ Средства Расширения Окно Справка Поиск ConsoleApplication3
Debug x64 Локальный отладчик Windows
ConsoleApplication3.cpp
ConsoleApplication3 (Глобальная область) main()
#include <iostream>
int main()
{
int *a;
std::cout << &a << '\n';
return 0;
}
Консоль отладки Microsoft Visual Studio
000000C1F8EFF598
C:\Users\tecce\source\repos\ConsoleApplication3\x64\Debug\ConsoleApplication3.exe (процесс 1996) завершил работу с кодом
0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Ав
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:

Присвоювання значень вказівнику

Оскільки вказівники містять тільки адреси, то при присвоюванні значення вказівнику — це значення повинно бути адресою. Для отримання адреси змінної використовується оператор адреси:

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

вказівник ptr містить адресу значення змінної value, і він вказує на це значення. У мові програмування C++ вказівник - це змінна, яка містить адресу в пам'яті. Призначення вказівника - забезпечити доступ до значення, яке зберігається в цій адресі пам'яті


ConsoleApplication3.cpp*
ConsoleApplication3
(Глобальная область)
main()
#include <iostream>
int main()
{
int value = 5;
int *ptr = &value; // ініціалізуємо ptr адресою значення змінної
std::cout << &value << '\n'; // виводимо адресу значення змінної value
std::cout << ptr << '\n'; // виводимо адресу, яку містить ptr
return 0;
}
Консоль отладки Microsoft Visual Studio
00000014BC0FF9D4
00000014BC0FF9D4
C:\Users\tecce\source\repos\ConsoleApplication3\x64\Debug\ConsoleApplication3.exe (процесс 12040) завершил работу с кодо
м 0.
Чтобы автоматически закрывать консоль при остановке отладки, включите параметр "Сервис" ->"Параметры" ->"Отладка" -> "Ав
томатически закрыть консоль при остановке отладки".
Нажмите любую клавишу, чтобы закрыть это окно:
100 %
Проблемы
Вывод
Показать выходные данные из
"ConsoleApplication3.exe"
"ConsoleApplication3.exe"
"ConsoleApplication3.exe"
Поток 12880 завершился с к
Поток 14044 завершился с к
Программа "[12040] ConsoleApplication3.exe" завершилась с кодом 0 (0x0).

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

```
1 int iValue = 7;
2 double dValue = 9.0;
3
4 int *iPtr = &iValue; // ок
5 double *dPtr = &dValue; // ок
6 iPtr = &dValue; // неправильно: вказівник типу int не може вказувати на адресу змінної типу double
7 dPtr = &iValue; // неправильно: вказівник типу double не може вказувати на адресу змінної типу int
```

Наступне не є допустимим:

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

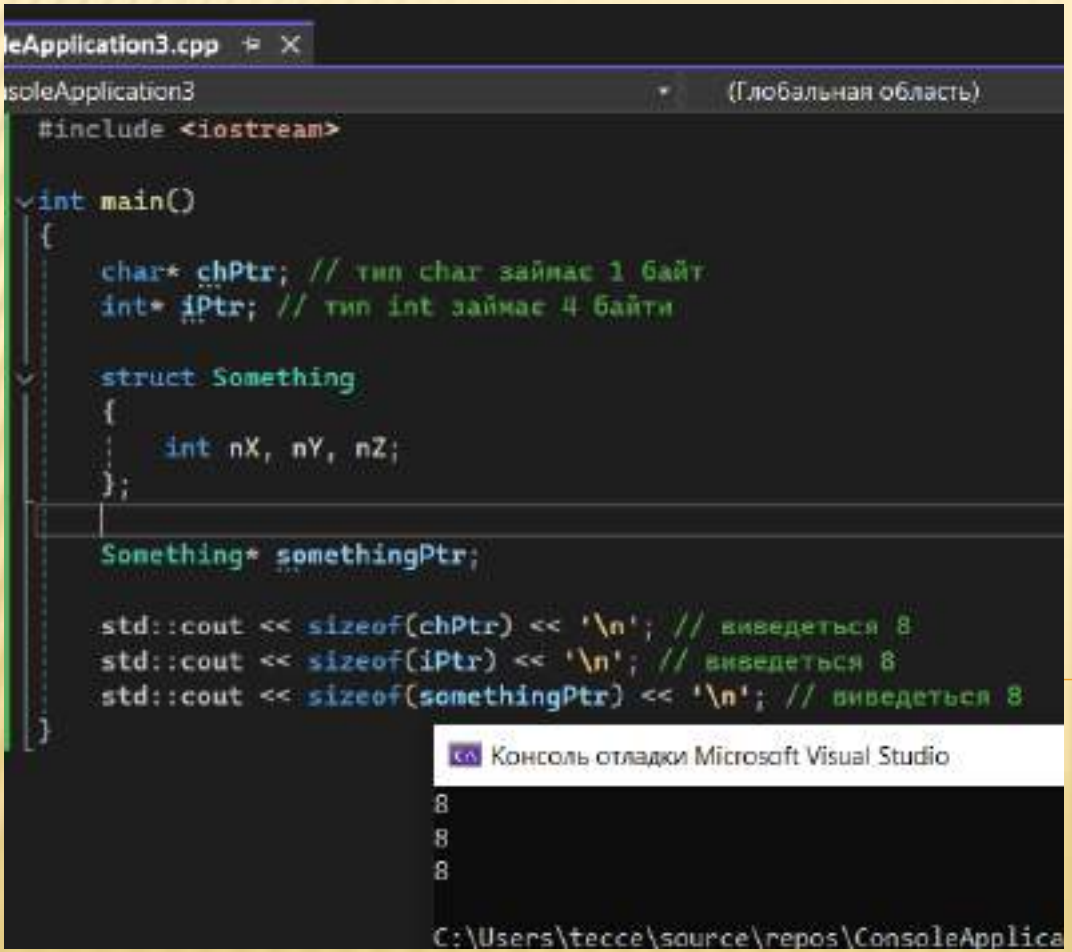
Це пов'язано з тим, що вказівники можуть містити тільки адреси, а цілочисельний літерал 7 не має адреси в пам'яті. Якщо ви все ж зробите це, то компілятор повідомить вам, що він не може перетворити цілочисельне значення в цілочисельний вказівник.

Мова C++ також не дозволить вам напряму присвоювати адреси в пам'яті вказівнику:

```
1 double *dPtr = 0x0012FF7C; // не ок, розглядається як присвоювання цілочисельного літералу
```

Розмір вказівників

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



Як ви можете бачити, розмір вказівника завжди один і той же. Це пов'язано з тим, що вказівник — це всього лише адреса в пам'яті, а кількість біт, необхідна для доступу до адреси в пам'яті на певному пристрої, — завжди постійна.

ВКАЗІВНИКИ НА МАСИВ

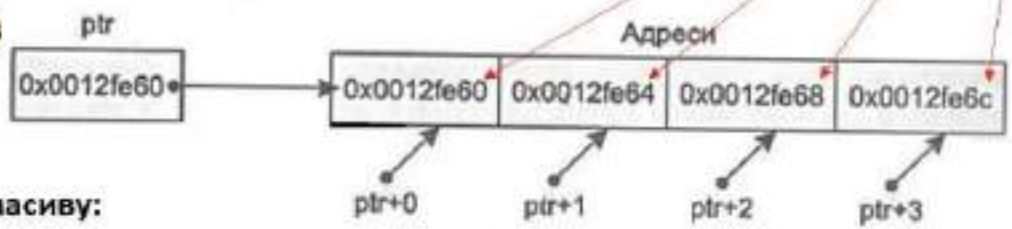
Арифметика вказівників

Дано: int ptr[4] – масив з 4 елементів.

Тип int – тому 1 елемент = 4 байта.

Всі йдуть один за одним тому:

Ptr – Вказівник на початок розміщення масиву в пам'яті!



Значення елементів масиву:

ptr[0]=*(ptr+0)
ptr[1]=*(ptr+1)
ptr[2]=*(ptr+2)
ptr[3]=*(ptr+3)

Чим корисні вказівники?

Зараз ви можете подумати, що вказівники є непрактичними і взагалі непотрібними. Навіщо використовувати вказівник, якщо ми можемо використати вихідну змінну?

Вказівники корисні в наступних випадках:

1. ***Випадок №1:*** **Масиви реалізовані за допомогою вказівників.** Вказівники можуть використовуватися для ітерації по масиву.

2. ***Випадок №2:*** **Вони є єдиним способом динамічного виділення пам'яті в C++.** Це, безумовно, найбільш поширений варіант використання вказівників.

3. ***Випадок №3:*** **Вони можуть використовуватися для передачі великої кількості даних в функцію без копіювання цих даних.**

4. ***Випадок №4:*** **Вони можуть використовуватися для передачі однієї функції в якості параметру іншій функції.**

5. ***Випадок №5:*** **Вони використовуються для досягнення поліморфізму при роботі зі спадкуванням.**

6. ***Випадок №6:*** **Вони можуть використовуватися для представлення однієї структури/класу в іншій структурі/класі, формуючи, таким чином, "ланцюжки".**

Вказівники застосовуються в багатьох випадках. Не хвилюйтеся, якщо ви багато чого не розумієте з вищесказаного. Тепер, коли ми розібралися зі вказівниками на базовому рівні, ми можемо почати заглиблюватися в окремі випадки, в яких вони корисні, що ми і зробимо на наступних лекціях.

Лекція №14

Вказівники і масиви

Лекція №15

Робота з рядками

Тип даних «текстовий рядок»

Рядок – послідовність (масив) символів. Якщо у виразі зустрічається одиночний символ, він повинен бути укладений в одинарні лапки. При використанні у виразах рядок береться в подвійні лапки. Ознакою кінця рядка є нульовий символ \0. У С ++ рядки можна описати за допомогою масиву символів (масив елементів типу char), в якому слід передбачити місце для зберігання ознаки кінця рядка.

Нехай заданий рядок ***s*** (див. мал.):

Пам'яті під неї відведемо 6 байт, тому опис буде наступним:

```
char s[6];
```

Корисна інформація (текст "ABC") розміщується в перших трьох байтах, а потім в третьому байті знаходиться нуль-символ '\ 0'.

Рядок ***s*** можна було б ініціювати, тобто на етапі компіляції задати їй початкове значення:

```
char s[6] = {'A', 'B', 'C', '\0'};
```

Так як ***s*** – це масив (хоча і символів), тому ініціалізація записана точно так же, як це зазвичай робиться для масиву. Але ***s*** – це ще й рядок, тому допускається спрощений варіант ініціалізації, схожий з ініціалізацією простих змінних:

```
char s[6] = "ABC";
```

При ініціалізації для рядка допустимо не задавати обсяг виділеної пам'яті:

```
char s[] = "ABC";
```

Тепер компілятор сам підрахує, скільки потрібно відвести пам'яті під рядок, і виділить під неї необхідний обсяг.

Коли пам'ять під рядок виділена, з цим рядком можна починати працювати. Рядки – вельми своєрідні об'єкти. З одного боку – це масиви, тому можна звертатися за індексом до потрібної комірки, змінювати її вміст, наприклад:

```
s[1]='i';
```

В результаті рядок ***"ABC"*** перетвориться в ***"AiC"***.

З іншого боку, робота з рядком схожа на роботу з простою змінною, наприклад, виведення рядка на екран монітора (використовуємо стандартну операцію виведення мови С ++):

```
cout << "s=" << s << endl;
```

Погодьтеся, що це набагато простіше, ніж виведення з використанням циклу:

```
for(i = 0; i < n; i++)
  cout << x[i] << endl;
```

Дії з рядками

1. Виведення рядка на екран монітора

а) Використовуємо стандартну операцію виведення мови С ++: **`cout << "s=" << s << endl;`**

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

int main() {
    string s = "Hello, world!";
    cout << "s=" << s << endl;
    return 0;
}
```

б) Використовуємо функцію виведення на екран символьного рядка (функція мови С): **`puts(s);`**

```
#include <stdio.h>

int main() {
    char s[] = "Hello, world!";
    puts(s);
    return 0;
}
```

Дії з рядками

Введення рядка

Використовуємо стандартну операцію введення мови С ++:**cin >> s;**

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

int main() {
    string s;
    cout << "Enter a string without spaces: ";
    cin >> s;
    cout << "You entered: " << s << endl;
```

застаріла функція gets(), яка була вилучена зі стандарту C++ (починаючи з C++11) через проблеми з безпекою. Замість неї рекомендується використовувати функцію getline() або std::cin.getline() для зчитування рядків, які можуть містити пробіли.

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

int main() {
    string s;
    cout << "Enter a string with spaces: ";
    getline(cin, s);
    cout << "You entered: " << s << endl;
```

Дії з рядками

Обчислення довжини рядка:

```
int k;
k = strlen(s);
```

тут ***k*** – це поточна довжина рядка, тобто кількість корисної інформації (формально – кількість елементів до нуль-символу). Наприклад, для рядка

```
char s[6] = "ABC";
```

k дорівнюватиме 3.

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

int main() {
    char s[6] = "ABC"; // Рядок має бути завершений нуль-символом
    int k = strlen(s); // Обчислення довжини рядка
    cout << "Length of string s: " << k << endl;
    return 0;
}
```

Дії з рядками

Об'єм пам'яті, відведеної під рядок, можна обчислити за формулою:
int size = sizeof(s)/sizeof(char);

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

int main() {
    char s[10] = "Hello";
    int size = sizeof(s) / sizeof(char);
    cout << "Memory allocated for string s: " << size << " bytes" << endl;
    return 0;
}
```

Цей код виведе кількість байтів, які виділені для масиву **s**, враховуючи кожен елемент масиву як **char**. У цьому прикладі вийде 10 байтів (якщо **char** займає 1 байт).

Дії з рядками

Зчеплення рядків (конкатенація)

```
strcat(s, s1);
```

Тут в кінець рядка ***s*** додається вміст рядка ***s1***. Результат зберігається в рядку ***s***. Рядок ***s1*** не змінюється.

Важливо впевнитися, що рядок **s** має достатньо великий розмір, щоб зберегти в собі як вміст вихідного рядка, так і вміст рядка **s1**, а також нуль-символ, який вказує на кінець рядка. В іншому випадку, це може призвести до непередбачуваних результатів, таких як переповнення буфера.

Дії з рядками

Зчеплення рядків (конкатенація)

```cpp
#include <iostream>
#include <cstring>
using namespace std;

int main() {
    char s[20] = "Hello ";
    char s1[] = "world!";

    strcat(s, s1); // Додаємо вміст s1 в кінець s

    cout << "Result: " << s << endl;

    return 0;
}
```

Дії з рядками

Копіювання рядків:

strcpy(s, s1);

Дані з рядка ***s1*** побайтно копіюються в рядок **s**. Колишній зміст рядка ***s*** втрачається, а рядок ***s1*** залишається незмінною.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char s[20]; // Рядок s повинен мати достатньо місця для зберігання s1
    char s1[] = "Hello, world!";

    strcpy(s, s1); // Копіюємо вміст s1 в s

    printf("Copied string: %s\n", s);

    return 0;
}
```

Дії з рядками

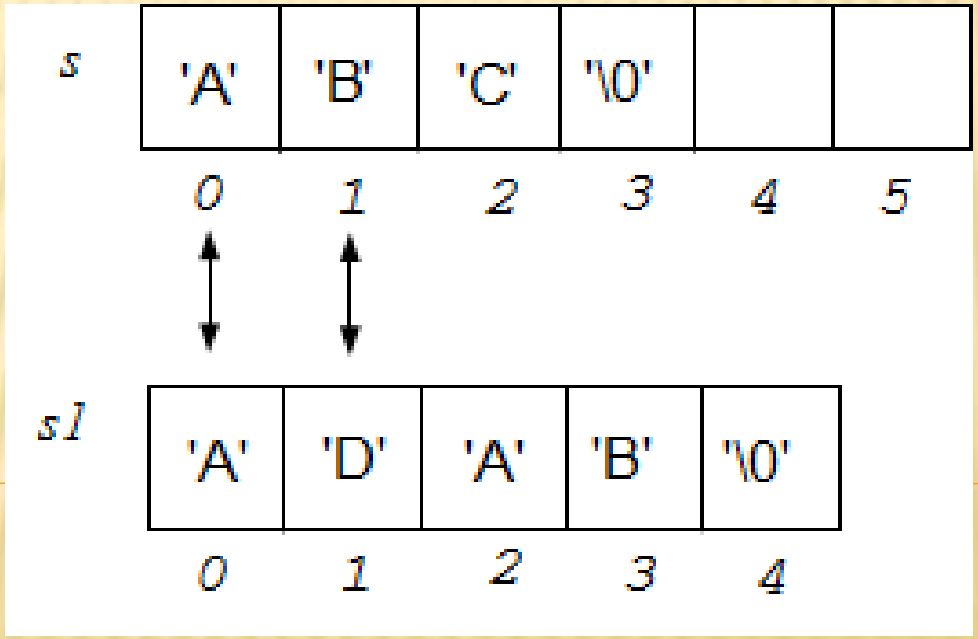
Порівняння рядків:

```
char s[6] = "ABC";
char s1[] = "ADAB";
int k = strcmp(s, s1);
```

У функції **strcmp()** виконується побайтне порівняння кодів символів: **s**[0] і ***s1***[0] рівні – йдемо далі; **s**[1] і ***s1***[1] – не рівні, код символу 'D' більше коду 'B', отже рядок ***s1*** більше **s**.

Результат роботи функції:

- k = 0 - рядки рівні;
- k <0 - рядок s менше ***s1***;
- k> 0 - рядок s більше ***s1***.



Дії з рядками

```c
#include <stdio.h>
#include <string.h>

int main() {
    char s[6] = "ABC";
    char s1[] = "ADAB";
    int k = strcmp(s, s1);

    if (k == 0) {
        printf("The strings are equal.\n");
    } else if (k < 0) {
        printf("String s is less than string s1.\n");
    } else {
        printf("String s is greater than string s1.\n");
    }

    return 0;
}
```

Рядки Visual Studio

Для роботи з рядками у середовищі **Visual Studio** передбачений клас **string** призначений для роботи з рядками типу **char***, які представляють собою рядок із завершальним нулем.

Щоб скористатися наявними можливостями класу **string** в **MS Visual Studio (C ++)**, потрібно підключити бібліотеку **<string>** і простір імен **std**.

#include <string>using namespace std;

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

int main() {
    string s1 = "Hello";
    string s2 = "World";
    string s3 = "Hello";

    // Порівняння рядків за допомогою оператора ==
    if (s1 == s2) {
        cout << "s1 and s2 are equal" << endl;
    } else {
        cout << "s1 and s2 are not equal" << endl;
    }

    // Порівняння частини рядка з іншим рядком за допомогою функції compare()
    int result = s1.compare(0, 5, s3); // Порівнюємо перші 5 символів s1 з рядком s3
    if (result == 0) {
        cout << "The first 5 characters of s1 are equal to s3" << endl;
    } else if (result < 0) {
        cout << "The first 5 characters of s1 are less than s3" << endl;
    } else {
        cout << "The first 5 characters of s1 are greater than s3" << endl;
    }
```

порівнюємо рядки **s1** та **s2** за допомогою оператора **==**, а потім порівнюємо перші 5 символів рядка **s1** з рядком **s3** за допомогою функції **compare()**

Лекція №15

Робота з рядками

Лекція №16

Робота з файлами

РОБОТА З ФАЙЛАМИ

Файл – проіменована сукупність даних, розташованих на зовнішньому носії.

На початку роботи для отримання доступу до даних файл необхідно відкрити. При відкритті файлу встановлюється вказівник на поточну позицію (як правило на початок даних у файлі). Після виконання будь-якої операції над даними вказівник автоматично переміщується на одну позицію даних вперед. Після завершення роботи з даними файлу його необхідно закрити.

Інформація про файл зберігається в управляючій структурі, яка має тип FILE.

Розрізняють два типи файлів: текстові та двійкові (бінарні).

Текстовий файл зберігає інформацію у вигляді послідовності символів. Виведення виконується аналогічно виводу на екран. Текстові файли можуть бути створені та відредаговані у будь-якому текстовому редакторі.

Бінарні файли призначені для зберігання послідовності байтів. Структура такого файлу визначається програмно.

Файли розташовуються на зовнішніх носіях і мають наступну структуру:

На початку записана інформація про файл **BOF** (*Begin of FILE*), його ім'я, тип, довжина і т.д. В кінці розміщується ознака кінця файлу **EOF** (*End of File*).

Для роботи з файлами використовують наступні макроси:

NULL – визначає пустий вказівник;

EOF — ознака кінця файлу

FOPEN_MAX – максимальна кількість одночасно відкритих файлів.

РОБОТА З ФАЙЛАМИ

Для запису даних в файл потрібно виконати

1. Описати покажчик на файл FILE * filename;
2. Відкрити файл (функція fopen)
3. Записати даних в файл (функція **fprintf**)

Функція **fprintf** аналогічна функції printf, єдиною відмінністю є перший параметр – покажчик на файл. За допомогою цієї функції вивід здійснюється не на екран, а в файл.

4. Закрити файл (функція **fclose**).

int fclose (FILE * filename);

Повертає 0 при успішному закритті файлу і NULL в іншому випадку.

Оголошення змінної типу файлу:

FILE * ідентифікатор;

Приклад

FILE * f; // f – ім'я файлу (ідентифікатор у програмі)

Для відкриття файлу використовується функція **fopen**:

fopen (ім'я фізичного файлу, режим доступу)

Ім'я фізичного файлу – шлях до файлу на диску та його ім'я на диску.

Режим доступу – рядок, який вказує режим відкриття файлу і тип файлу

РОБОТА З ФАЙЛАМИ

```c
#include <stdio.h>

int main() {
    FILE *file_ptr; // оголошення покажчика на файл

    // Відкриття файлу у режимі запису
    file_ptr = fopen("test.txt", "w");

    if (file_ptr == NULL) {
        printf("Помилка відкриття файлу!");
        return 1;
    }

    // Запис рядка у файл
    fprintf(file_ptr, "Це рядок, який буде записаний у файл!");

    // Закриття файлу
    fclose(file_ptr);

    printf("Рядок успішно записано у файл!");

    return 0;
```

У цьому прикладі ми:

1. Оголосили покажчик на файл file_ptr типу FILE *.
2. Відкрили файл test.txt у режимі запису за допомогою функції fopen().
3. Записали рядок тексту у файл за допомогою функції fprintf().
4. Закрили файл за допомогою функції fclose().
5. Якщо все пройде успішно, ми отримаємо повідомлення "Рядок успішно записано у файл!" у вихідному потоці.

РОБОТА З ФАЙЛАМИ

Режими відкриття

Значення	Опис
r	Файл відкривається тільки для читання. Вказівник встановлюється на початок.
w	Файл відкривається тільки для запису. Якщо відповідний фізичний файл існує, він буде перезаписаний. Вказівник встановлюється на початок.
a	Файл відкривається для запису в кінець (для дозаписи) або створюється, якщо не існує. Вказівник встановлюється в кінець
r+	Файл відкривається для читання і запису. Вказівник встановлюється на початок.
w+	Файл відкривається для запису і читання. Якщо відповідний фізичний файл існує, він буде перезаписаний. Вказівник встановлюється на початок.
a+	Файл відкривається для запису в кінець (для дозаписи) або створюється, якщо не існує. Вказівник встановлюється в кінець

РОБОТА З ФАЙЛАМИ

```
FILE *file_ptr;
file_ptr = fopen("file.txt", "r");
if (file_ptr == NULL) {
    printf("Помилка відкриття файлу!");
    return 1;
}
```

Режим "r" - відкриття файлу для читання. Якщо файл не існує, функція fopen() поверне NULL.

РОБОТА З ФАЙЛАМИ

```c
#include <stdio.h>

int main() {
    FILE *fp;
    char filename[] = "example.txt";

    // Відкриття файлу для читання
    fp = fopen(filename, "r");
    if (fp == NULL) {
        printf("Помилка відкриття файлу.\n");
        return 1;
    }

    // Використання файлу...

    // Закриття файлу
    if (fclose(fp) == 0) {
        printf("Файл успішно закрито.\n");
    } else {
        printf("Помилка закриття файлу.\n");
    }

    return 0;
}
```

У цьому прикладі функція fclose використовується для закриття файлу після його використання. Якщо функція повертає 0, це означає успішне закриття файлу, в іншому випадку буде виведено повідомлення про помилку.

РОБОТА З ФАЙЛАМИ

int fgetc (FILE*fp);

Здійснює введення чергового символу із вказаного файлу. Якщо повертає EOF, то сталась помилка або досягнуто кінця файлу.

int fputc (int ch, FILE*fp);

Виводить символ з кодом ch у вказаний файл. Повертає змінну з кодом ch або EOF, якщо сталась помилка.

char * fgets (char *s, int n, FILE *fp);

Ця функція здійснює введення в стрінг s (буфер) символів із файлу, який ідентифікується файловою змінною fp, допоки не виконається одна з умов:

- виникає символ '\n' (початок нового рядку);
- досягнутий кінець файлу;
- прочитано n-1 символів.

Після цього стрінг доповнюється '\0' . В першому випадку перед '\0' записується '\n'.

Якщо читання з файлу завершено успішно, то повертає вказівник на стрінг s, в разі помилки повертає NULL.

int fputs (char *s, FILE*fp);

Ця функція виводить стрінг s у вказаний файл. Повертає 0 або EOF, якщо сталась помилка.

```c
int main() {
    FILE *inputFile, *outputFile;
    char filename[] = "input.txt";
    char outputFilename[] = "output.txt";
    char buffer[100]; // Буфер для fgets

    // Відкриваємо вхідний файл для читання
    inputFile = fopen(filename, "r");
    if (inputFile == NULL) {
        printf("Помилка відкриття вхідного файлу.\n");
        return 1;
    }

    // Відкриваємо вихідний файл для запису
    outputFile = fopen(outputFilename, "w");
    if (outputFile == NULL) {
        printf("Помилка відкриття вихідного файлу.\n");
        fclose(inputFile); // Закриваємо вхідний файл перед виходом
        return 1;
    }

    // Використання fgetc та fputc для копіювання файлу
    int c;
    while ((c = fgetc(inputFile)) != EOF) {
        fputc(c, outputFile);
    }

    // Закриваємо файли
    fclose(inputFile);
    fclose(outputFile);

    // Відкриваємо вхідний файл ще раз для читання
    inputFile = fopen(filename, "r");
    if (inputFile == NULL) {
        printf("Помилка відкриття вхідного файлу.\n");
        return 1;
    }

    // Використання fgets для читання рядка
    printf("Зміст файлу '%s':\n", filename);
    while (fgets(buffer, sizeof(buffer), inputFile) != NULL) {
        printf("%s", buffer); // Виводимо рядок на екран
    }

    // Закриваємо вхідний файл
    fclose(inputFile);
```

РОБОТА З ФАЙЛАМИ

РОБОТА З ФАЙЛАМИ

int fprintf (FILE *fp , char *format, ...);

Ця функція здійснює форматований вивід своїх аргументів в файл, який ідентифікується файловою змінною **fp, під керівництвом стрінгу format. Відносно цієї функції справедливо** все те, що було сказано про функцію стандартного форматованого виводу **printf (див.**раніше).

int fscanf (FILE *fp , char *format, ...);

Здійснює форматоване, визначене стрінгом **format, введення з вказаного файлу fp.** Відносно цієї функції справедливо все те, що було сказано про функцію стандартного форматованого виводу **scanf.**

int feof (FILE *fp);

Повертає не **0 (тобто ІСТИНУ), якщо при читанні з файлу був досягнутий кінець файлу і 0** (тобто **ХИБНІСТЬ) у супротивному випадку.**

int ferror (FILE *fp);

Повертає не **0 (тобто ІСТИНУ), якщо при виконанні операцій введення або виведення з файлом fp виникали помилки і 0 (тобто ХИБНІСТЬ) у супротивному випадку.**

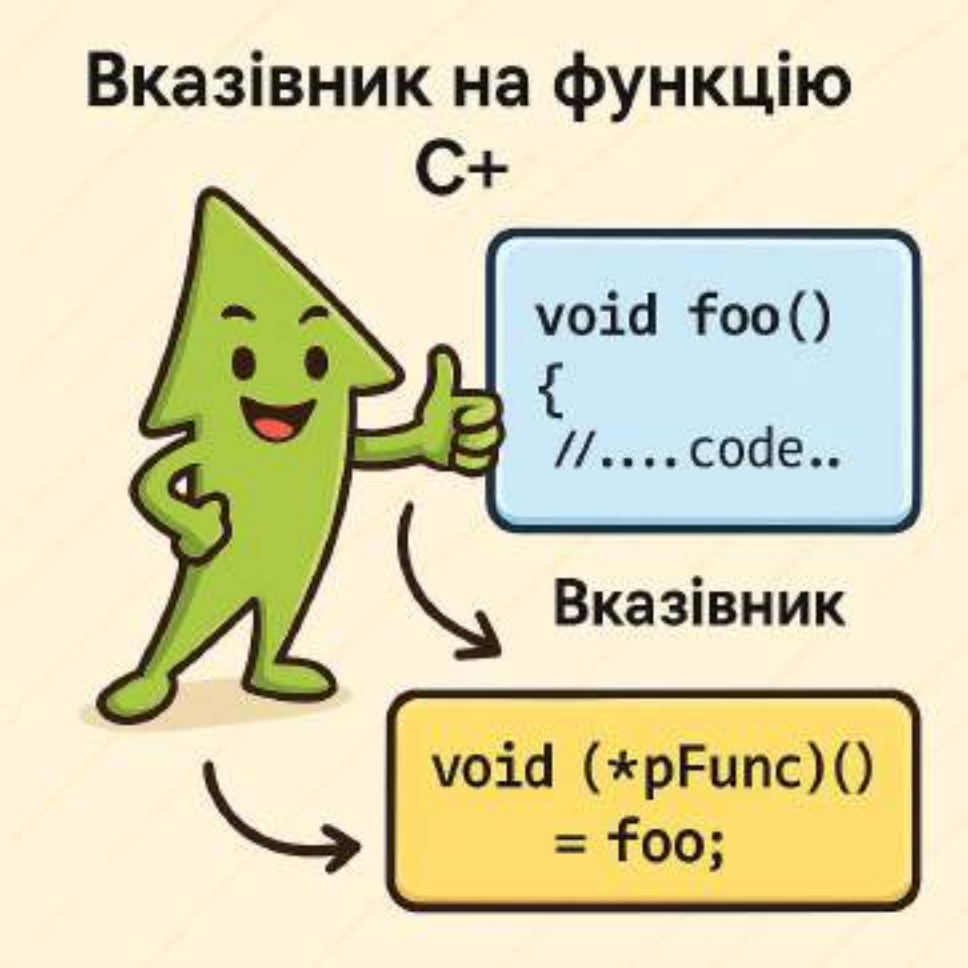
void rewind (FILE *fp);

Пересуває вказівник поточної позиції в файлі на початок файлу. При цьому вказівники кінця файлу та помилок зануляються. Після виклику цієї функції файл можна читати повторно спочатку.

Лекція №16

Робота з файлами

Лекція №17


Вказівник на функцію
C+
void foo()
{
//....code..
Вказівник
void (*pFunc)()
= foo;

ВКАЗІВНИК НА ФУНКЦІЮ

🤔 Основні причини використання вказівників на функції в C++

1. 👉 Передача функції як аргументу

У C++ можна передавати функцію у вигляді вказівника до іншої функції. **Наприклад:** щоб написати універсальний алгоритм, який викликає різні дії, залежно від того, яку функцію йому "передали".

2. 🔁 Заміна умовних конструкцій (if-else / switch)

Іноді замість великого ланцюга if-else краще використати масив вказівників на функції — це компактніше і швидше працює.

3. 🧩 Зручність для створення колбеків (callback-функцій)

Коли потрібно, щоб одна функція викликала іншу в певний момент (наприклад, коли користувач натискає кнопку або завершується якась операція).

4. 🔄 Динамічне вибирання функцій

Програмі не потрібно знати заздалегідь, яку саме функцію викликати. Вона може обрати потрібну функцію під час виконання.

5. 🧱 Реалізація гнучких інтерфейсів

Вказівники на функції дають змогу будувати більш абстрактні і гнучкі структури, наприклад, в бібліотеках, де користувач сам підставляє потрібні функції.

ВКАЗІВНИК НА ФУНКЦІЮ

Синтаксис оголошення вказівника на функцію в C++ виглядає наступним чином:

```
тип_повернення (*ім'я_вказівника)(список_параметрів)
```

де:
тип_повернення - тип даних, що повертається функцією.
ім'я_вказівника - ім'я вказівника на функцію.
список_параметрів - перелік параметрів, які приймає функція, разом із їх типами.

Наприклад, оголошення вказівника на функцію, яка приймає два цілочисельних параметри та повертає ціле число, буде виглядати так:

```
int (*functionPtr)(int, int);
```

Це оголошення показує, що functionPtr є вказівником на функцію, яка приймає два цілочисельних параметри та повертає ціле число.

Якщо вам необхідно оголосити вказівник на функцію зі змінним числом параметрів або з параметрами різних типів, то типи параметрів повинні відповідати параметрам функції, на яку вказує вказівник.

ВКАЗІВНИК НА ФУНКЦІЮ

```cpp
#include <iostream>

// Функція, яка приймає вказівник на функцію як аргумент
void performOperation(int a, int b, int (*operation)(int, int)) {
    int result = operation(a, b);
    std::cout << "Result: " << result << std::endl;
}

// Приклад функції додавання
int add(int x, int y) {
    return x + y;
}

// Приклад функції віднімання
int subtract(int x, int y) {
    return x - y;
}

int main() {
    // Вказівники на функції
    int (*addPtr)(int, int) = &add;
    int (*subtractPtr)(int, int) = &subtract;

    // Виклик функції з вказівником на функцію в якості аргумента
    performOperation(5, 3, addPtr); // Результат: 8
    performOperation(5, 3, subtractPtr); // Результат: 2

    return 0;
}
```

У цьому прикладі функція **performOperation** приймає два числа та вказівник на функцію, яка приймає два цілочисельні аргументи і повертає ціле число. У функції **main** визначаються дві функції: **add** та **subtract**, які відповідають за додавання і віднімання відповідно.

Згодом в **main** створюються вказівники на ці функції (**addPtr** і **subtractPtr**). Ці вказівники передаються у функцію **performOperation**, де вони використовуються для виклику відповідних операцій. Таким чином, використання вказівників на функції дозволяє передавати функції як аргументи і виконувати їх залежно від потреб програми

ВКАЗІВНИК НА ФУНКЦІЮ

Вказівнику на функцію може бути присвоєний інший вказівник на функцію зі схожим прототипом. Це може бути корисним у випадках, коли ви хочете, щоб вказівник на функцію вказував на різні функції з аналогічним інтерфейсом. Використання такого підходу дозволяє вам легко змінювати функціональність, на яку вказує вказівник, без необхідності змінювати виклики функцій в інших частинах коду

```
void (*pf)(void); // вказівник на функцію без параметрів
void (*qf)(void); qf = pf;
```

Вказівник на функцію – це змінна, що містить адресу деякої функції, тому може бути використаний для виклику цієї функції.

Лекція №17

Вказівник на функцію

Лекція №18

Рекурсивні функції

РЕКУРСИВНІ ФУНКЦІЇ

Рекурсивна функція в мові програмування C++ - це функція, яка викликає сама себе. Це досить потужний інструмент для розв'язання задач, що можуть бути виражені у вигляді повторюваних або рекурсивних дій.

Основна ідея рекурсивної функції полягає в тому, що функція може викликати себе з меншими вхідними значеннями, доки не буде досягнуто **базового випадку**, який завершить рекурсію. Без правильно заданого базового випадку функція буде викликати себе безкінечно, що призведе до переповнення стеку (stack overflow) і припинення виконання програми.

РЕКУРСИВНІ ФУНКЦІЇ

```cpp
#include <iostream>

// Рекурсивна функція для обчислення факторіала числа n
int factorial(int n) {
    // Базовий випадок: якщо n == 0, факторіал буде 1
    if (n == 0)
        return 1;
    // Рекурсивний виклик: факторіал числа n дорівнює n * факторіал(n-1)
    else
        return n * factorial(n - 1);
}

int main() {
    int n;
    std::cout << "Введіть число для обчислення факторіала: ";
    std::cin >> n;
    std::cout << "Факторіал " << n << " дорівнює " << factorial(n) << std::endl;
    return 0;
}
```

У цьому прикладі функція factorial викликає сама себе з меншим значенням n, поки не досягне базового випадку (якщо n дорівнює 0), після чого рекурсія завершується, і повертається результат.

УМОВА ЗАВЕРШЕННЯ РЕКУРСІЇ

Рекурсивні виклики функцій працюють точно так же, як і звичайні виклики функцій. Однак, програма, наведена вище, ілюструє найбільш важливу відмінність простих функцій від рекурсивних: ви повинні вказати умову завершення рекурсії, в протилежному випадку — функція виконуватиметься нескінченну кількість разів (фактично до тих пір, поки не закінчиться пам'ять в стеці викликів).

Умова завершення рекурсії — це умова, при якій рекурсивна функція перестане викликати саму себе. В цій умові зазвичай використовується оператор if.

РЕКУРСИВНІ АЛГОРИТМИ

Рекурсивні алгоритми - це алгоритми, що використовують рекурсію для вирішення задач. Рекурсивні алгоритми часто використовуються для вирішення завдань, які мають структуру подібну до самих себе.

Деякі типові приклади рекурсивних алгоритмів включають:

Обход дерева: Наприклад, обхід бінарного дерева в глибину (preorder, inorder, postorder) може бути реалізований за допомогою рекурсії. Функція рекурсивно викликає саму себе для кожного вузла дерева.

Обчислення факторіалу: Як показано в попередньому прикладі, функція обчислення факторіалу може бути реалізована за допомогою рекурсії.

Сортування: Наприклад, сортування злиттям (merge sort) може бути реалізоване рекурсивно. Алгоритм розбиває вихідний масив на дві половини, сортує кожну половину рекурсивно, а потім об'єднує дві відсортовані половини у відсортований масив.

Пошук в глибину: Пошук в глибину в графі також може бути реалізований за допомогою рекурсії. Функція рекурсивно відвідує кожен вузол графа, починаючи з визначеного вузла.

Пошук шляху: Наприклад, знаходження всіх можливих шляхів в графі може бути реалізоване за допомогою рекурсії, де функція рекурсивно переходить до наступного вузла в графі.

Генерація перестановок та комбінацій: Рекурсивні алгоритми можуть бути використані для генерації всіх можливих перестановок або комбінацій елементів масиву.

Математичні обчислення: Наприклад, обчислення чисел Фібоначчі може бути реалізоване за допомогою рекурсії.

Графічні об'єкти: Рекурсивні алгоритми можуть бути використані для побудови складних графічних об'єктів, таких як фрактали.

При реалізації рекурсивних алгоритмів важливо правильно визначити базовий випадок (умову завершення рекурсії) та впевнитися, що рекурсивні виклики зближаються до базового випадку у кінцеву кількість кроків.

РЕКУРСІЯ VS ІТЕРАЦІЇ

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

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

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

Загалом, рекурсія є хорошим вибором, якщо виконується більшість з наступних тверджень:

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

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

// Рекурсивна функція для обчислення n-го числа Фібоначчі
int fibonacci(int n) {
    if (n == 0) return 0;          // базовий випадок 1
    if (n == 1) return 1;          // базовий випадок 2
    return fibonacci(n - 1) + fibonacci(n - 2); // рекурсія
}

int main() {
    int n;
    cout << "Введіть номер елемента послідовності Фібоначчі: ";
    cin >> n;

    cout << "Число Фібоначчі з індексом " << n << " = " << fibonacci(n) << endl;

    return 0;
}
```

⚠ Цей рекурсивний варіант працює повільно при великих n, бо багато викликів повторюються. Але як приклад — ідеально 👌

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

// Ітеративна функція для обчислення n-го числа Фібоначчі
int fibonacci(int n) {
    if (n == 0) return 0;
    if (n == 1) return 1;

    int prev1 = 0;  // F(0)
    int prev2 = 1;  // F(1)
    int current;

    for (int i = 2; i <= n; i++) {
        current = prev1 + prev2;
        prev1 = prev2;
        prev2 = current;
    }

    return current;
}

int main() {
    int n;
    cout << "Введіть номер елемента послідовності Фібоначчі: ";
    cin >> n;

    cout << "Число Фібоначчі з індексом " << n << " - " << fibonacci(n) << endl;

    return 0;
}
```

Лекція №18

Рекурсивні функції

Windows Forms на C++

Лекція №19

Що таке Windows Forms?

- **Windows Forms** — це платформа для створення графічних інтерфейсів користувача у десктопних додатках на базі **Microsoft .NET Framework**.

📌 **Підтримує мови:** C++/CLI, C#, VB.NET

📌 **Середовище розробки:** Visual Studio

📌 **Призначення:** Створення віконних додатків під Windows

🔧 **Основні характеристики**

- **Бібліотека компонентів**
  Готові елементи: кнопки, текстові поля, списки, таблиці тощо
- **Подієво-орієнтована модель**
  Обробка подій — реакція на дії користувача (клік, натиск клавіш)
- **Інтеграція з .NET**
  Робота з базами даних (ADO.NET), серіалізація, async/await
- **Кастомізація**
  Можна створювати власні компоненти
- **Візуальне проектування**
  Drag-and-drop у Visual Studio + автоматичне створення коду

🧩 Приклади використання Windows Forms

- **Windows Forms** — це платформа для створення графічних інтерфейсів користувача у десктопних додатках на базі **Microsoft .NET Framework.**

🏢 Бізнес-додатки

- Інтерфейси для роботи з базами даних (CRM, облік товарів)
- Програми для звітності та управління документами
- ERP-системи з формами введення/редагування даних

🛠️ Утиліти для Windows

- Конвертери форматів (PDF → DOCX, MP4 → MP3 тощо)
- Текстові редактори та переглядачі логів
- Системні інструменти (наприклад, лончери, автоапдейтери)

Підтримка та розвиток

💡 **Windows Forms досі підтримується Microsoft**, але:

Для нових проєктів рекомендовано розглядати:

- **WPF (Windows Presentation Foundation)** — підтримка XAML, розширена графіка
- **UWP (Universal Windows Platform)** — адаптивні інтерфейси, інтеграція з Windows 10/11

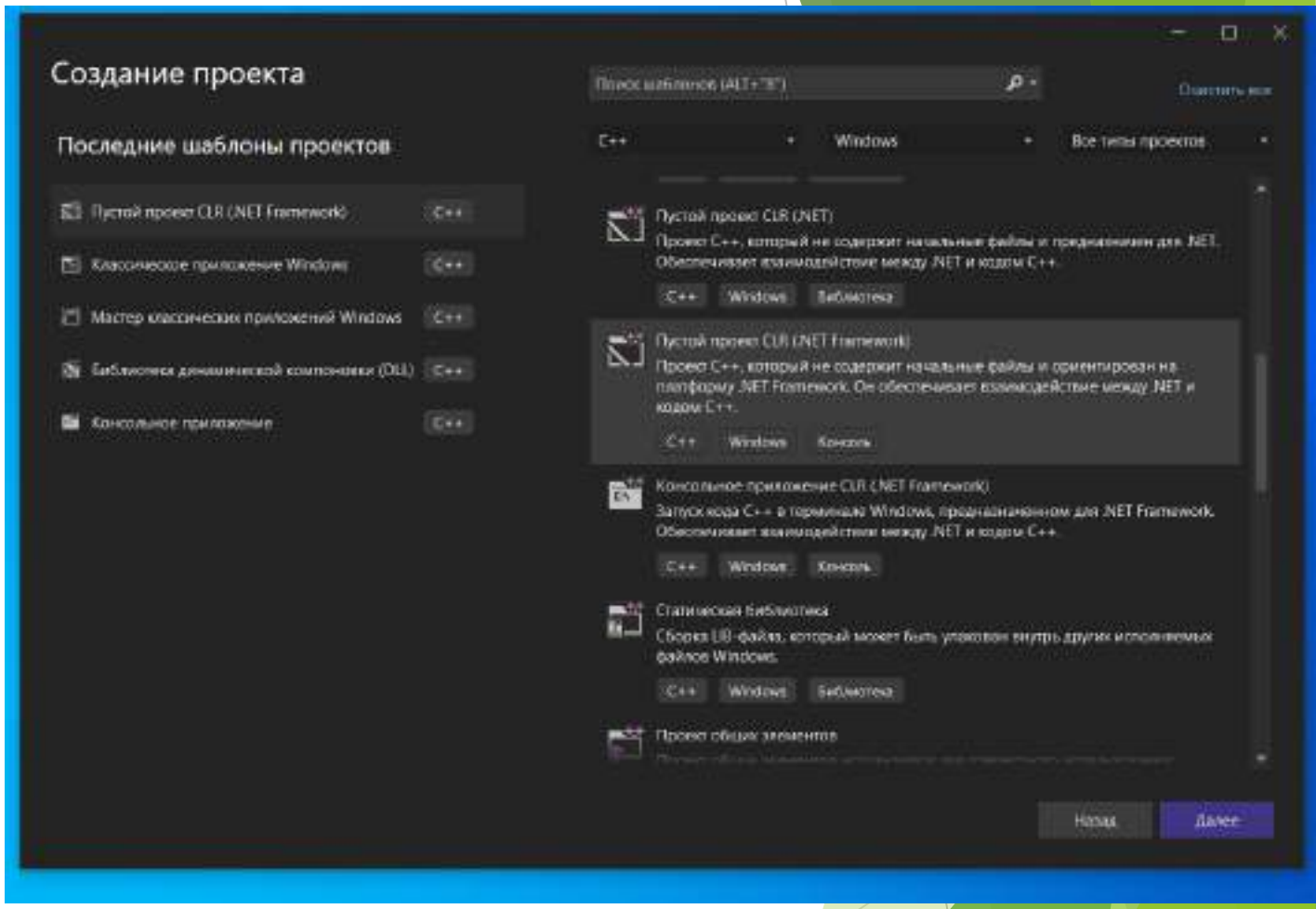
📌 Тим не менш, **Windows Forms залишається актуальним** для швидкої розробки простих і стабільних настільних додатків.

🧩 Створення Windows Forms на C++/CLI (через CLR-проєкт)

◇ Щоб створити справжній **Windows Forms** додаток із візуальним дизайнером форм у **C++**, обери:

Порожній проект CLR (.NET Framework)
Мова: **C++**
Платформа: **Windows**
Тип: **Консоль / GUI (в залежності від налаштувань)**

🧠 Цей шаблон дозволяє використовувати **.NET-бібліотеки**, зокрема простір імен System::Windows::Forms, що потрібно для форм.

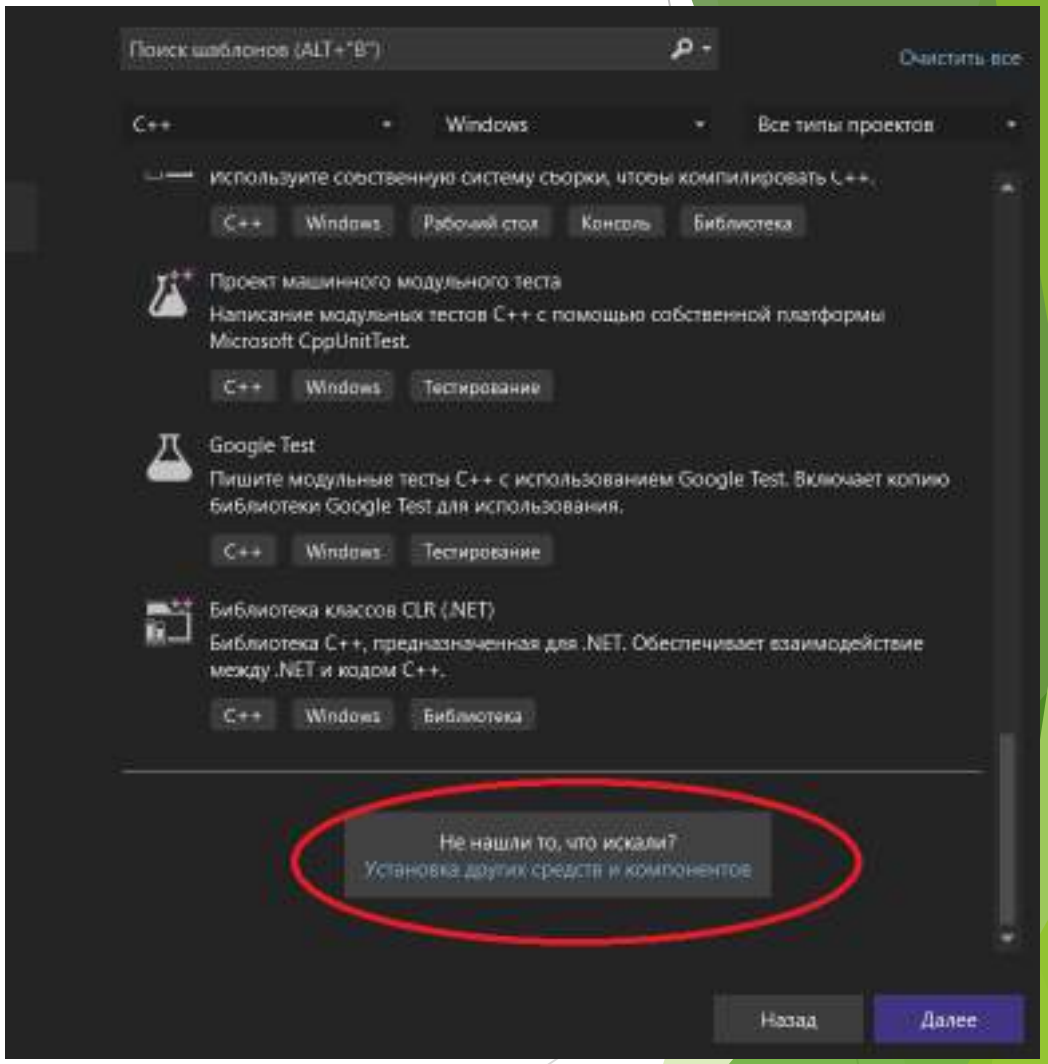


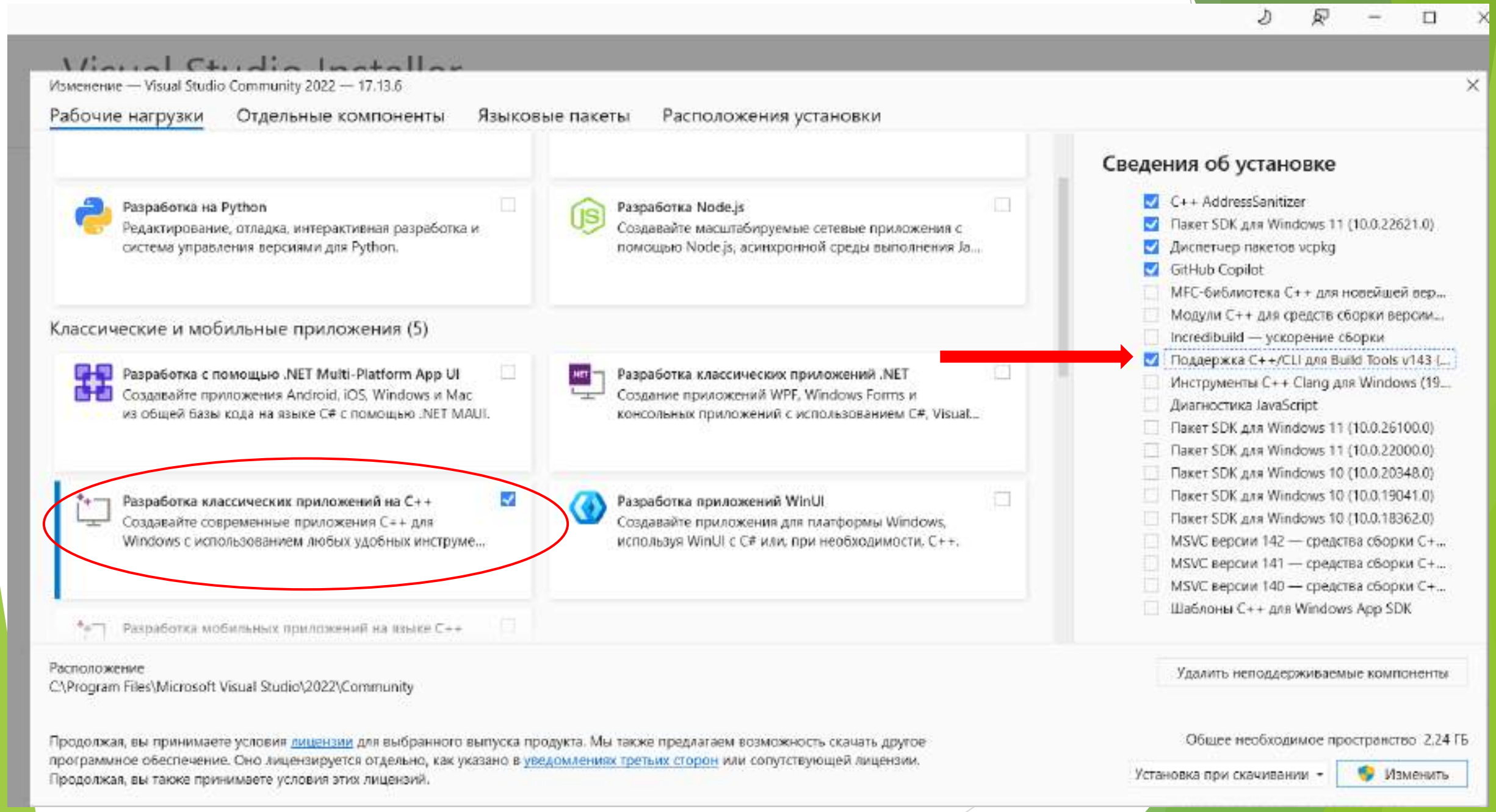
▶ Створення Windows Forms проєкту на C++

🔧 Так, дуже важливий момент! На цьому етапі Visual Studio може **не показувати шаблони для Windows Forms на C++/CLI.**

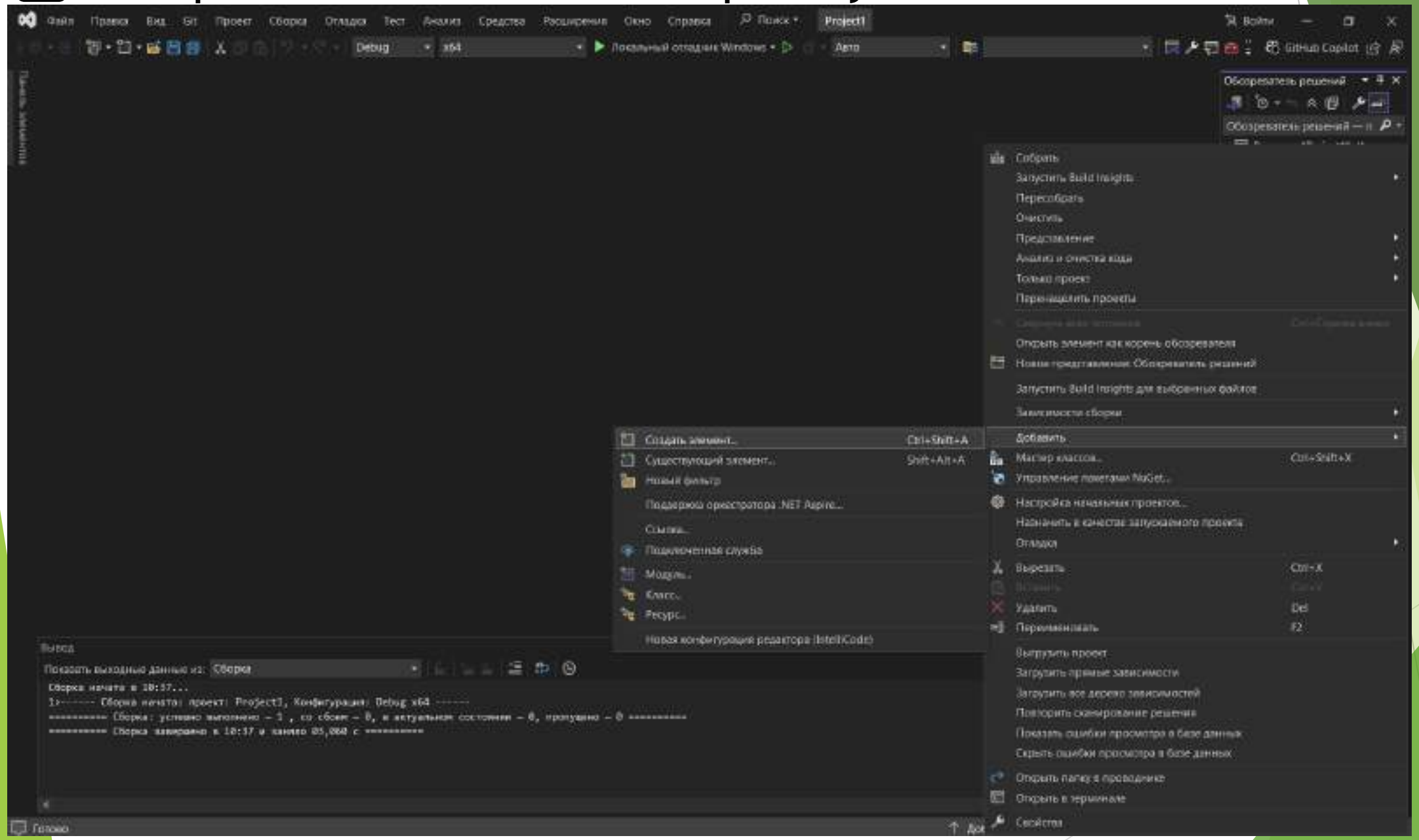
☑ **Установити необхідний робочий набір:**

✔ **"Розробка класичних програм на .NET (C++/CLI)"**
або
✔ **"Розробка для .NET Desktop" + C++/CLI підтримка**

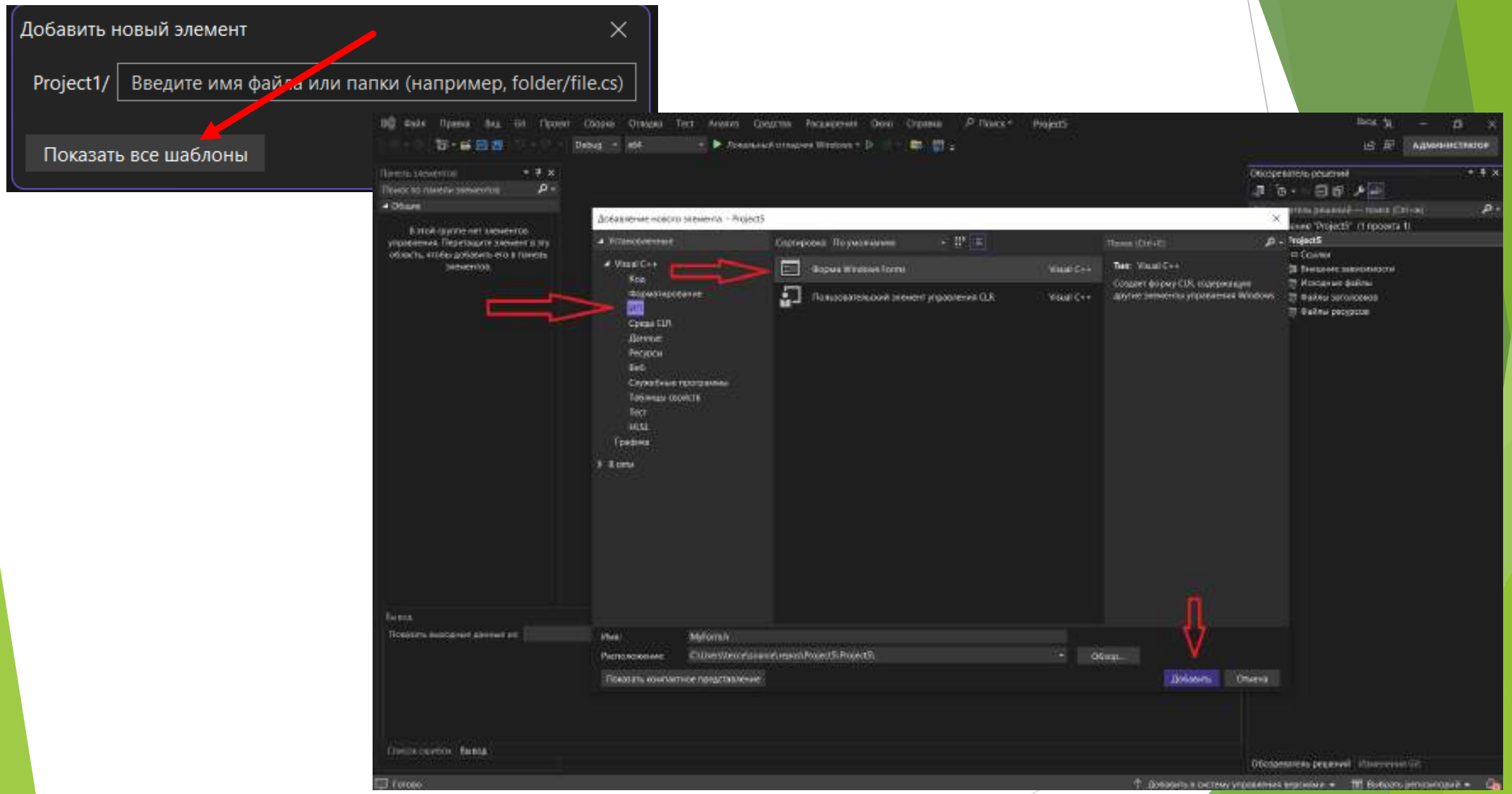



Visual Studio Installer
Изменение — Visual Studio Community 2022 — 17.13.6
Рабочие нагрузки
Отдельные компоненты
Языковые пакеты
Расположения установки
Разработка на Python
Редактирование, отладка, интерактивная разработка и система управления версиями для Python.
Разработка Node.js
Создавайте масштабируемые сетевые приложения с помощью Node.js, асинхронной среды выполнения Ja...
Классические и мобильные приложения (5)
Разработка с помощью .NET Multi-Platform App UI
Создавайте приложения Android, iOS, Windows и Mac из общей базы кода на языке C# с помощью .NET MAUI.
Разработка классических приложений .NET
Создание приложений WPF, Windows Forms и консольных приложений с использованием C#, Visual...
Разработка классических приложений на C++
Создавайте современные приложения C++ для Windows с использованием любых удобных инструме...
Разработка приложений WinUI
Создавайте приложения для платформы Windows, используя WinUI с C# или, при необходимости, C++.
Разработка мобильных приложений на языке C++
Сведения об установке
C++ AddressSanitizer
Пакет SDK для Windows 11 (10.0.22621.0)
Диспетчер пакетов vcpkg
GitHub Copilot
MFC-библиотека C++ для новейшей вер...
Модули C++ для средств сборки версии...
Incredibuild — ускорение сборки
Поддержка C++/CLI для Build Tools v143 (...
Инструменты C++ Clang для Windows (19...
Диагностика JavaScript
Пакет SDK для Windows 11 (10.0.26100.0)
Пакет SDK для Windows 11 (10.0.22000.0)
Пакет SDK для Windows 10 (10.0.20348.0)
Пакет SDK для Windows 10 (10.0.19041.0)
Пакет SDK для Windows 10 (10.0.18362.0)
MSVC версии 142 — средства сборки C+...
MSVC версии 141 — средства сборки C+...
MSVC версии 140 — средства сборки C+...
Шаблоны C++ для Windows App SDK
Расположение
C:\Program Files\Microsoft Visual Studio\2022\Community
Удалить неподдерживаемые компоненты
Продолжая, вы принимаете условия лицензии для выбранного выпуска продукта. Мы также предлагаем возможность скачать другое программное обеспечение. Оно лицензируется отдельно, как указано в уведомлениях третьих сторон или сопутствующей лицензии. Продолжая, вы также принимаете условия этих лицензий.
Общее необходимое пространство 2,24 ГБ
Установка при скачивании
Изменить

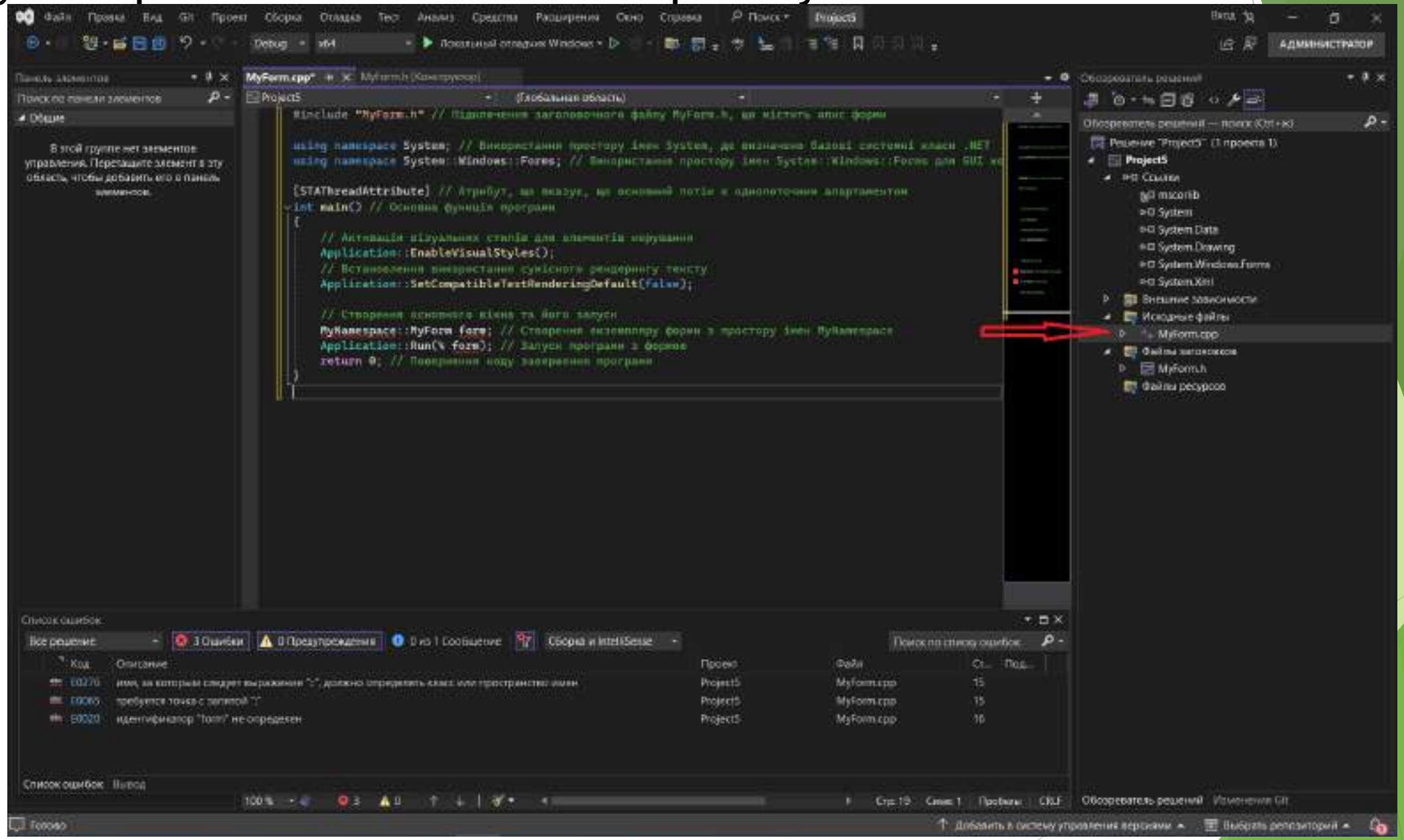
Створення Windows Forms проєкту на C++



Створення Windows Forms проєкту на C++



Створення Windows Forms проєкту на C++



Створення Windows Forms проєкту на C++

```
#include "MyForm.h"
using namespace System;
using namespace System::Windows::Forms;

[STAThread]
void main() {
	Application::EnableVisualStyles();
	Application::SetCompatibleTextRenderingDefault(false);
	Project5::MyForm form;
	Application::Run(%form);
}
```

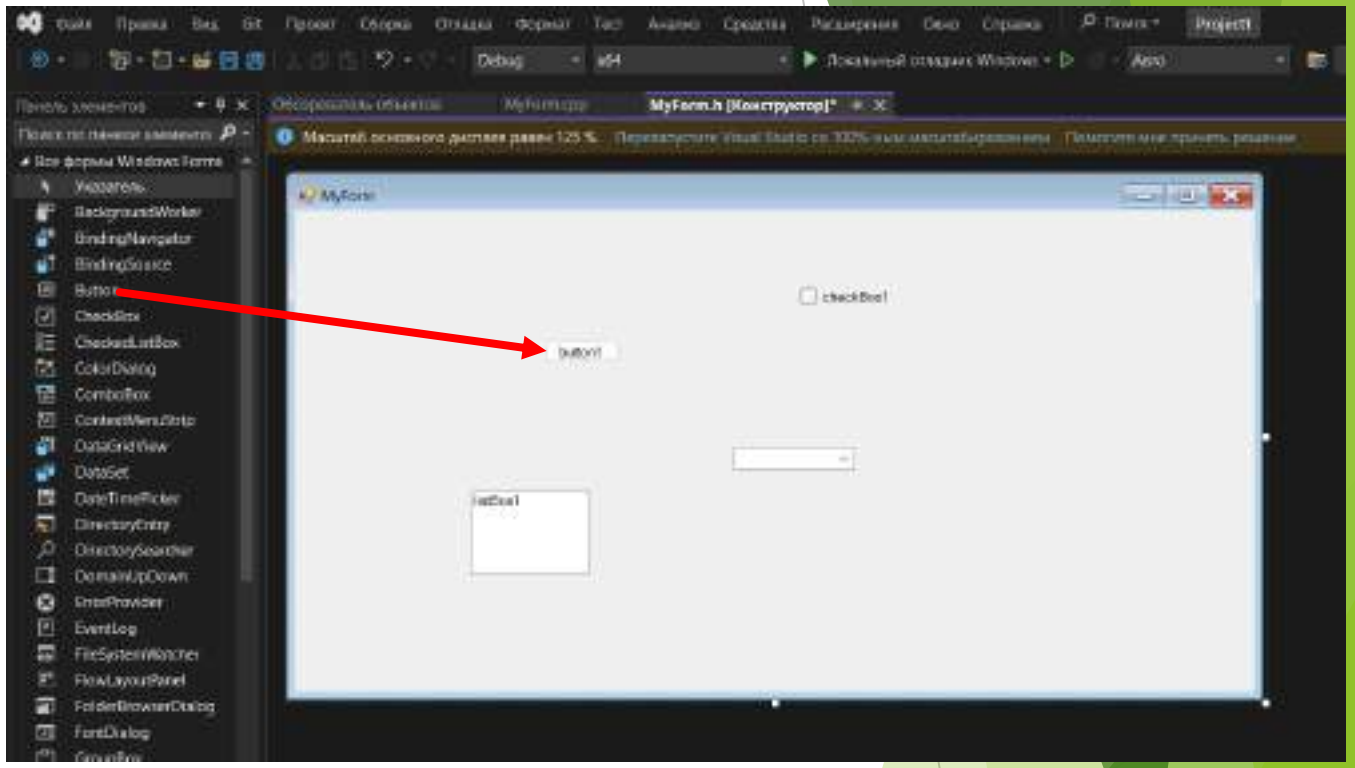
▶ Створення Windows Forms проєкту на C++

☑ **Як відкрити Панель елементів (Toolbox) у Visual Studio:**

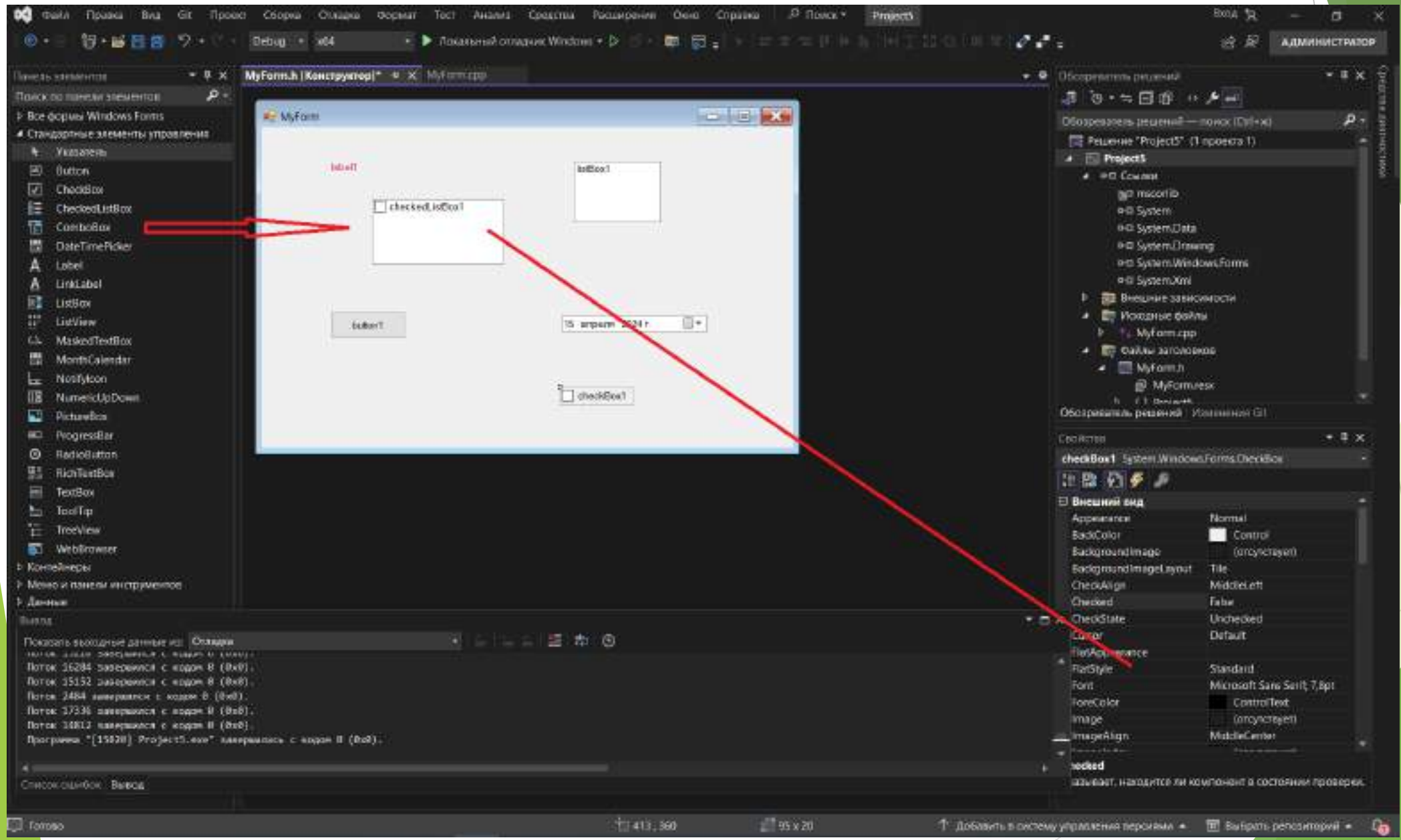
📌 **Спосіб 1: Через меню**
Перейди в меню **"Вид" (View)** →
🔽 Обери **"Панель элементов" / "Toolbox"**

📌 **Спосіб 2: Клавіатурне скорочення**
Натисни **Ctrl + Alt + X**

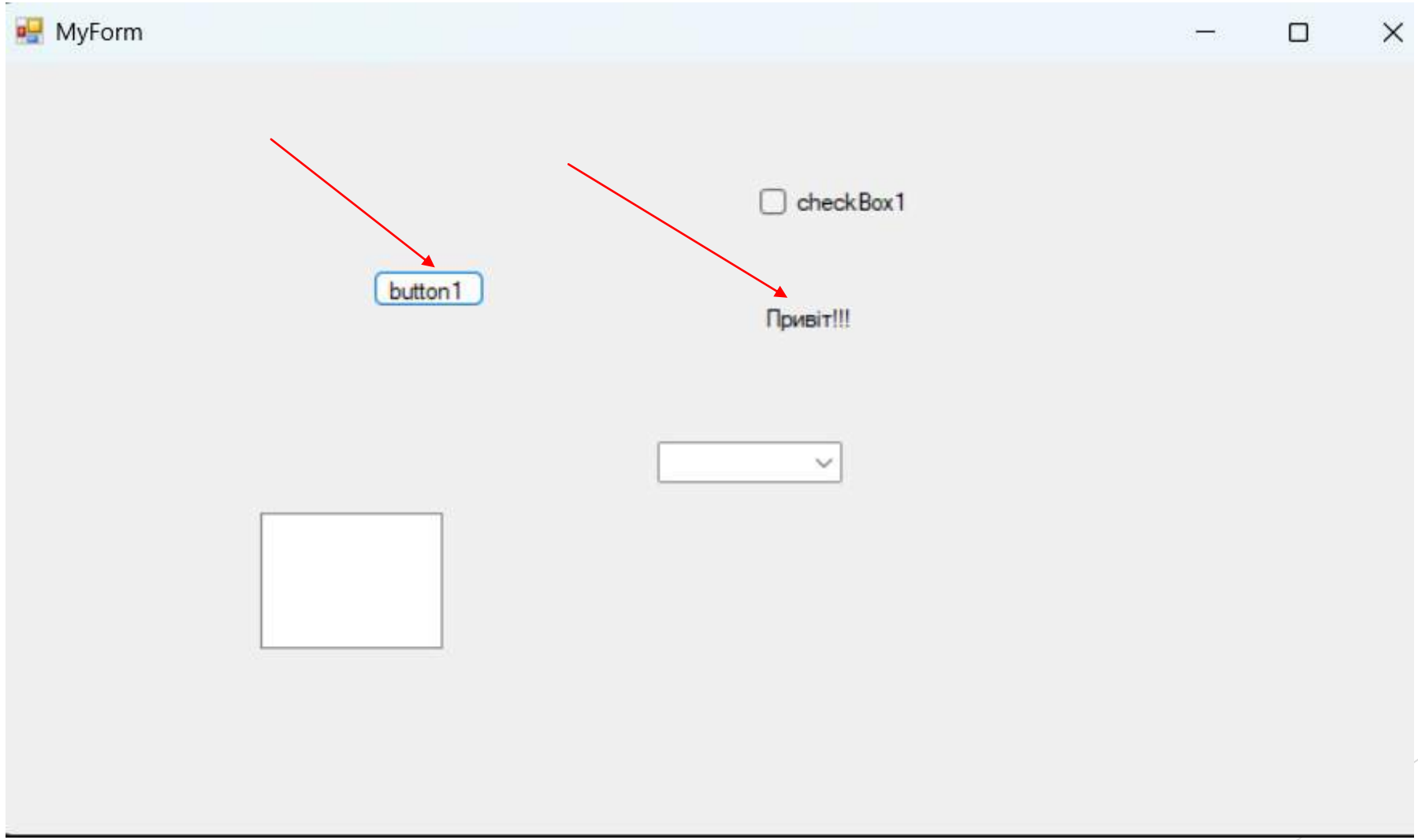
📌 **Спосіб 3: Через пошук**
У полі пошуку вгорі Visual Studio (Ctrl + Q) введи: "Toolbox" або "Панель элементов" і натисни Enter



Створення Windows Forms проєкту на C++



Створення Windows Forms проєкту на C++



Windows Forms на C++

Лекція №19

Робота з класами та об'єктами в C++

Лекція №20

Що таке ООП?

Визначення:

Об'єктно-орієнтоване програмування (ООП) — це парадигма програмування, в якій основними одиницями є **об'єкти**, що поєднують **дані (атрибути)** і **поведінку (методи)**.

Основні поняття:

Поняття	Опис
Клас	Шаблон для створення об'єктів
Об'єкт	Конкретний екземпляр класу
Атрибути	Змінні, що описують стан об'єкта
Методи	Функції, які визначають поведінку об'єкта

Принципи ООП:

- **Інкапсуляція** – приховування деталей реалізації
- **Наслідування** – створення нового класу на основі існуючого
- **Поліморфізм** – можливість об'єктів по-різному реалізовувати одні й ті самі методи

Що таке ООП?

Приклад:

```
class Dog {
public:
    void bark() {
        cout << "Woof!";
    }
};
```

Клас Dog

Клас — шаблон або креслення для створення об'єктів.

Об'єкт — конкретний екземпляр класу.

```
Dog myDog;
myDog.bark();
```

Об’єкт myDog

Метод bark()

Схема :
[Клас → Об'єкт → Виклик методу]

🧩 Члени класу: змінні та методи

- Змінні-члени (атрибути) — описують стан об'єкта.
- Методи-члени (функції) — визначають поведінку об'єкта.

🔐 **Модифікатори доступу: public, private, protected.**

```
class Car {
private:
    string model;
public:
    void drive() {
        cout << "Driving...";
    }
};
```

Атрибут: model

Метод: drive()

Модифікатор: private/public

Модифікатори доступу в C++

- public – доступ відкритий для всіх (інших класів, функцій, об'єктів).
- private – доступ обмежений лише межами самого класу. Інші не мають доступу.
- protected – доступ дозволено з класу та його нащадкам (похідним класам).

```
class Example {
public:
    int pubVar; // доступний для всіх
protected:
    int protVar; // доступний для
нащадків
private:
    int privVar; // доступний лише
всередині класу
};
```

Інкапсуляція

- Інкапсуляція — це приховування внутрішніх деталей реалізації класу від зовнішнього доступу.
- Забезпечує захист даних об'єкта від неправильного використання.
- Здійснюється через модифікатори доступу (private, public) та get/set методи.

```
class BankAccount {
private:
    double balance;
public:
    void deposit(double amount);
    double getBalance();
};
```

Конструктори

🔧 Конструктор — це спеціальний метод класу, який автоматично викликається при створенні об’єкта.

✔ Назва конструктора збігається з назвою класу.

✔ Може бути перевантаженим — кілька конструкторів з різними параметрами.

```
class Person {
public:
    string name;
    Person(string n) { name = n; }
};
```

Деструктори в C++

- Деструктор — це спеціальний метод класу, який автоматично викликається при знищенні об’єкта.
- Назва деструктора збігається з назвою класу, але з символом '~' на початку.
- Деструктор не приймає параметрів і не повертає значення. У класі може бути лише один деструктор.

```
~Person() {
    cout << "Person object destroyed";
}
```

Створення та використання об'єктів

- Об'єкт — це екземпляр класу, який містить власні копії змінних-членів класу.
- Після створення об’єкта ми можемо викликати його методи або змінювати атрибути.
- Синтаксис: Ім'я_класу Ім'я_об'єкта;

```
class Person {
public:
    string name;
    void sayHello() {
        cout << "Hello, I'm " << name;
    }
};

Person p;
p.name = "Anna";
p.sayHello();
```

Висновки

- Клас — це шаблон, за яким створюються об'єкти.
- Об'єкти мають свої атрибути (змінні) та методи (функції).
- Інкапсуляція дозволяє приховати реалізацію класу і захистити дані.
- Конструктори автоматично викликаються при створенні об'єкта.
- Деструктори автоматично викликаються при знищенні об'єкта.
- Об'єкти створюються легко та зручно, що забезпечує гнучкість і модульність коду.
- ООП — потужний підхід до створення програм, який спрощує розробку великих проектів.

Обробка винятків та налагодження програм у C++

Лекція №21

🔍 Мета заняття:

- Ознайомится з механізмами **обробки винятків** у C++
- Навчити використовувати інструменти **налагодження програм** у середовищі розробки (Visual Studio)

📓 Ключові поняття:

- try, catch, throw
- Класи стандартних винятків (std::exception)
- Точки зупину, перегляд змінних, крокове виконання
- Практика безпечного кодування

💡 Чому це важливо?

Обробка винятків підвищує **надійність** програм, а налагодження дозволяє **швидко знаходити помилки** і покращувати якість коду.

Поняття винятку

🔧 Що таке виняток?

- Ситуація, яка виникає під час виконання програми та порушує нормальний потік виконання
- Наприклад: ділення на нуль, вихід за межі масиву, порушення доступу до пам’яті

❗ Види помилок:

- Синтаксичні — виявляються під час компіляції (наприклад, відсутня крапка з комою)
- Логічні — неправильна реалізація алгоритму
- Виконання (runtime errors) — наприклад, відкриття неіснуючого файлу

🎯 Мета обробки винятків — надати контроль над помилками під час виконання програми

🧩 Блоки try-catch у C++

🔄 Основна конструкція:

```
try {
    // Код, що може спричинити виняток
} catch (тип_винятку) {
    // Обробка винятку
}
```

📌 Особливості:

- Один блок try може мати декілька блоків catch
- Порядок catch важливий — від специфічного до загального
- Уникнення аварійного завершення програми

Блоки try-catch у C++

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

int main() {
    int x = 5;
    int y = 0;

    try {
        int z = x / y; // Спроба ділення на 0
        cout << "Результат: " << z << endl;
    } catch (...) {
        cout << "Сталася помилка: ділення на нуль!" << endl;
    }

    return 0;
}
```

🔄 Ключове слово throw

🚀 Ключове слово throw використовується для створення винятку:

- throw "Ділення на нуль!";
- throw std::runtime_error("Помилка введення");

🔁 throw можна використовувати:

- У функції (після виявлення помилки)
- Усередині конструктора/методу
- Для повторного вкидання винятку

🧩 Стандартні винятки (exception)

🛠️ Бібліотека <exception> надає класи для винятків:

- std::exception — базовий клас
- std::runtime_error, std::logic_error
- std::out_of_range, std::invalid_argument

📥 Приклад:

```
try {
    throw std::out_of_range("Індекс за межами масиву");
} catch (const std::exception& e) {
    std::cout << e.what();
}
```

Користувацькі винятки

🧩 Створення власного класу винятку:

```
class MyException : public std::exception {
    const char* what() const noexcept override {
        return "Моя помилка";
    }
};
```

📌 Використовуйте користувацькі винятки, коли стандартні не описують ситуацію повністю

Налагодження програм — огляд

🧭 Що таке налагодження (debugging)?

- Процес виявлення і виправлення помилок у програмі

🔧 Основні інструменти:

- Breakpoints (точки зупину)
- Watches (перегляд змінних)
- Step Over / Into (покрокове виконання)
- Call Stack (стек викликів)

▶ Практика в Visual Studio

💻 Як налагоджувати у Visual Studio:

1. Поставте точку зупину натиснувши на поле біля номера рядка
2. Натисніть F5 для запуску в режимі налагодження
3. Використовуйте F10 (Step Over) або F11 (Step Into)
4. Переглядайте змінні у вікні Locals або Autos

▶ Стратегії тестування та захисту коду

🌓 Профілактика винятків:

- Завжди перевіряйте введення користувача
- Перевіряйте ділення на нуль
- Обробляйте винятки в точках доступу до ресурсів (файли, пам’ять)

☑ Добра практика — не допустити винятків замість їх обробки

▶ Стратегії тестування та захисту коду

🌓 Профілактика винятків:

- Завжди перевіряйте введення користувача
- Перевіряйте ділення на нуль
- Обробляйте винятки в точках доступу до ресурсів (файли, пам’ять)

☑ Добра практика — не допустити винятків замість їх обробки