МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ
Національний університет «Запорізька політехніка»

КОНСПЕКТ ЛЕКЦІЙ

з курсу

«Програмування»

для студентів денної та заочної форм навчання спеціальності

124 – «Системний аналіз»

2025

Конспект лекцій з курсу «Програмування» для студентів денної та заочної форм навчання спеціальності 124 – «Системний аналіз» / Укл.: Д.В. Широкорад, А.Є. Рябенко. - Запоріжжя: НУ «Запорізька політехніка», 2025. - 58 с.

Укладачі**:** Д.В. Широкорад, доцент, к.ф.-м..н.,

Рецензент: Г.В. Корніч, професор, д.ф.-м.н.

Відповідальний за випуск Е.В. Терещенко, доцент, к.ф.-м.н.

Затверджено на засіданні кафедри “Системний аналіз та обчислювальна математика».
Протокол № 15 від 10.04.2025

Рекомендовано до видання НМК ФКНТ
Протокол № 8 від 17.04.2025

ЗМІСТ

ВСТУП

Курс спрямований на студентів, які тільки розпочинають знайомство з програмуванням. У межах курсу учасники опанують синтаксис і базові конструкції мови C++, зокрема умовні оператори, цикли, роботу з одновимірними та двовимірними масивами, функції, операції з рядками та обробку текстових файлів.

У межах курсу студенти розвинуть навички алгоритмічного мислення та навчаться створювати, компілювати й налагоджувати консольні програми, а також здобудуть практичний досвід роботи з інструментами розробки, включаючи компілятори, системи збірки, засоби відлагодження та контроль версій. Після завершення курсу учасники зможуть самостійно розробляти прості програми на C++, налаштовувати робоче середовище для розробки та будуть готові до подальшого поглибленого вивчення об’єктно-орієнтованого програмування й сучасних стандартів C++..

1 ОСНОВИ МОВИ ПРОГРАМУВАННЯ С++

1.1 Значення мов програмування та роль С++

Мови програмування слугують інструментом комунікації між людьми та комп'ютерами. С++ поєднує в собі низькорівневий контроль ресурсів із високорівневими абстракціями:

Висока продуктивність: компіляція у машинний код легко піддається оптимізації

Контроль пам'яті: явне керування виділенням і звільненням пам'яті (heap vs stack)

Гнучкість парадигм: процедурне, об'єктно-орієнтоване, генеричне через шаблони

Стандартна бібліотека: алгоритми й контейнери із STL для швидкої розробки

За допомогою С++ створюють ядра операційних систем, високопродуктивні сервери, ігрові рушії та вбудовані пристрої.

1.2 Історія та стандарти С++

Історія С++ – це етапи вдосконалення від «С с классами» (1983) до сучасних модулів і концептів:

- С++98/03: базовий ООП, STL
- С++11: лямбда-вирази, auto, smart pointers, multithreading
- С++14/17: вдосконалення синтаксису, filesystem, optional, variant
- С++20/23: корутини, модулі, концепти, розширені можливості бібліотеки

1.3 Інструментарій і workflow розробника

Для ефективного навчання важливо зрозуміти інструменти розробника:

Компакт опис toolchain: препроцесор ➔ компілятор ➔ лінкер ➔ виконуваний файл

Build-системи: Make, CMake, Ninja – вибір залежить від складності проєкту

Debugging: налаштування точок зупину, розбір стеку викликів у GDB/LLDB/IDE

Контроль версій: базові команди Git: init, commit, branch, merge, push

Встановлення середовища та перевірка

Linux: install build-essential, Git

Windows: MinGW-w64 або Visual Studio Community + Git for Windows

IDE/редактори: VS Code, CLion, Visual Studio

Перевірка компілятора: g++ --version, clang++ --version

1.4 Ідентифікатори та ключові слова

Ідентифікатор - ім’я змінної, функції, константи тощо

Правила ідентифікації:

Починаються з літери латиниці (A-Z, a-z) або знака підкреслення

Можуть містити літери, цифри та підкреслення

Чутливі до регістру (value ≠ Value)

Заборонено використовувати ідентифікатори, що збігаються з ключовими словами

Ключові слова (найчастіші): int, double, char, bool, if, else, for, while, return, include, namespace, using тощо

1.5 Типи даних та змінні

Прості (скалярні) типи:

int, short, long, unsigned - цілі числа

float, double, long double - числа з плаваючою комою

char - один символ

bool - логічний тип (true/false)

Оголошення змінних:

```
int age = 20;
double price = 99.99;
char letter = 'A';
```

```
bool flag = true;
```

Ініціалізація під час оголошення рекомендується для запобігання невизначеним значенням

1.6 Літерали

Целочисельні: десяткові (100), шістнадцяткові (0xFF), вісімкові (077)
Дробові: звичайні (3.14), наукова нотація (1e-3)
Символьні: 'a', '
', 'A'
Стрічкові: "Hello", широкі L"Привіт"
Булеві: true, false

1.7 Оператори та вирази

Арифметичні: +, -, , /, %
Присвоєння: =, +=, -=, =, /=, %=
Порівняння: ==, !=, <, >, <=, >=
Логічні: &&, ||, !
Унарні: ++, --, унарний - та +
Приклад виразу:

```
int a = 10, b = 3;
int sum = a + b;
bool ok = (a > b) && (b != 0);
```

1.8 Інструкції та блоки коду

Інструкція - одиночна команда, закінчується ;
Блок - група інструкцій у фігурних дужках { ... }
Приклад блоку:

```
{
  int x = 5;
  x += 2;
```

```
    std::cout << x << std::endl;
}
```

1.9 Коментарі та стилістика коду

Однорядковий: // це коментар
Багаторядковий: / коментар /
Стиль оформлення:
Відступи по 2-4 пробіли
Єдиний стиль іменування: camelCase або snakecase
Коментарі пояснюють складні ділянки коду, але не дублюють очевидне

1.10 Компіляція та запуск

Процес: препроцесор ➔ компіляція ➔ лінкування

Команда g++:

```
g++ -std=c++17 main.cpp -Wall -o main
./main
```

Прапори:

*** std=c++17 - стандарт мови**

*** Wall - увімкнення всіх попереджень компілятора**
Основи вводу/виводу (I/O)

C++ надає механізми для взаємодії з користувачем і файлами через потоки:
cin/cout/cerr: стандартні потоки вводу/виводу та помилок
Форматування: маніпулятори <iomanip>: setw, setprecision
Буферизація: розуміння flush, відмінності
та std::endl

1.11 Перша програма на C++:

```
int main() {
   std::cout << "Привіт, C++ курс!" << std::endl;
   int a, b;
   std::cout << "Введіть два числа: ";
   if (!(std::cin >> a >> b)) {
      std::cerr << "Невірні дані!" << std::endl;
      return 1;
   }
   std::cout << "Сума: " << (a + b) << std::endl;
   return 0;
}
```

2 ПРОГРАМУВАННЯ РОЗГАЛУЖЕНЬ

Програмування розгалужень-це механізм прийняття рішень у програмі. Завдяки умовним конструкціям програма може виконувати різні дії залежно від виконання певної умови. Це дозволяє реалізувати логіку "якщо-тоді-інакше" та реагувати на різні ситуації під час виконання програми.

2.1 Умовний оператор if та if...else

Визначення: Умовний оператор if виконує блок коду, якщо задана логічна умова є істинною (true). Якщо умова хибна (false), код всередині if пропускається. Розширена форма-if...else-дозволяє вказати альтернативний блок коду, який виконається, коли умова є хибною. Синтаксис:

```
if (умова) {
  // код, що виконується якщо умова істинна
} else {
  // код, що виконується якщо умова хибна
}
```

Пояснення синтаксису: У круглих дужках після if записується логічний вираз (умова), який може приймати значення true або false. Якщо умова істинна, виконуються інструкції у фігурних дужках після if. Якщо є блок else, він виконається у випадку, коли умова для if не виконана (істинна). Приклад використання if...else: Нижче наведено приклад коду, де програма перевіряє число і визначає, парне воно чи непарне:

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

int main() {
  int number;
  cout << "Введіть ціле число: ";
  cin >> number;
```

```
    if (number % 2 == 0) {
        cout << number << " є парним\n";
    } else {
        cout << number << " є непарним\n";
    }

    return 0;
}
```

У цьому коді використовується оператор % (остача від ділення) для перевірки парності числа. Якщо остача від ділення на 2 дорівнює 0, число парне, і виконується перший блок if. В іншому випадку виконується блок else.

2.2 Каскадні умови if...else if

Опис: Коли існує кілька взаємовиключних умов, використовують каскадну конструкцію if...else if...else. Першою перевіряється умова після першого if. Якщо вона хибна-перевіряється наступна умова після else if, і так далі. Якщо жодна з умов не спрацювала, можна додати заключний блок else на випадок, коли всі попередні умови хибні. Синтаксис if...else if:

```
if (умова1) {
    // код для випадку, коли умова1 істинна
} else if (умова2) {
    // код для випадку, коли умова1 хибна, а умова2 істинна
} else if (умова3) {
    // код для випадку, коли умова1 і умова2 хибні, а умова3 істинна
} else {
    // код для випадку, коли жодна з умов 1-3 не істинна
}
```

Приклад каскадних умов:

```
int grade;
cout << "Введіть вашу оцінку (0-100): ";
cin >> grade;

if (grade >= 90) {
   cout << "Ваша оцінка: A\n";
} else if (grade >= 75) {
   cout << "Ваша оцінка: B\n";
} else if (grade >= 60) {
   cout << "Ваша оцінка: C\n";
} else if (grade >= 50) {
   cout << "Ваша оцінка: D\n";
} else {
   cout << "Ваша оцінка: F (незадовільно)\n";
}
```

У цьому коді перевіряється діапазон оцінки студента і виводиться відповідна літера. Порядок умов має значення: як тільки знаходиться перша істинна умова, інші не перевіряються, і відповідний блок коду виконується.

2.3 Вкладені умовні оператори

Опис: Можна розміщувати один умовний оператор всередині іншого, утворюючи вкладені умови. Це корисно, коли рішення в одній гілці потребує додаткових перевірок. Головне-стежити за правильною розстановкою фігурних дужок та відступів для читабельності. Приклад вкладених if:

```
int a = 5, b = -3;
if (a > 0) {
   cout << "a додатнє\n";
   if (b > 0) {
```

```
    cout << "b також додатнє\n";
  } else {
    cout << "b не є додатнім\n";
  }
}
```

Спочатку перевіряється умова a > 0. Якщо a додатнє, всередині цього блоку перевіряється ще одна умова про b. Таким чином, b перевіряється лише у випадку, якщо a було додатнім.

2.4 Оператор вибору switch

Визначення: Оператор switch використовується для розгалуження логіки на основі значення певної змінної (цілого або символу, типу enum, а починаючи з C++17 і типу std::stringview). Він дозволяє порівняти значення змінної з кількома константними варіантами (мітками case) і виконати різний код залежно від того, який випадок збігся. Синтаксис switch:

```
switch (вираз) {
  case значення1:
    // код, якщо вираз == значення1
    break;
  case значення2:
    // код, якщо вираз == значення2
    break;
  // ... інші випадки ...
  default:
    // код, якщо жоден із вищенаведених випадків не підійшов
    break;
}
```

вираз-це змінна або вираз цілочислового чи символьного типу (або сумісного типу).

Ключове слово case зазначає константу, з якою порівнюється вираз. Якщо вони рівні-виконується відповідний блок.

break необхідний для виходу з switch після виконання потрібного блоку, інакше виконання "провалюється" до наступного case.

default-необов'язкова гілка, код з якої виконається, якщо не спрацював жоден case.

Приклад switch:

```
char gradeChar;
cout << "Введіть оцінку (A, B, C, D, F): ";
cin >> gradeChar;

switch (gradeChar) {
  case 'A':
    cout << "Відмінно!\n";
    break;
  case 'B':
    cout << "Дуже добре\n";
    break;
  case 'C':
    cout << "Добре\n";
    break;
  case 'D':
    cout << "Задовільно\n";
    break;
  case 'F':
    cout << "Незадовільно\n";
    break;
  default:
    cout << "Неправильна літера оцінки\n";
    break;
}
```

Цей приклад читає символ оцінки і використовує switch для вибору відповідного повідомлення. Якщо введено будь-що, крім перелічених літер, спрацює блок default.

2.5 Тернарний оператор ?:

Опис: Тернарний умовний оператор-це скорочена форма if-else, яка повертає значення на основі умови. Його структура: умова ? виразякщоtrue : виразякщо_false. Він зручний для простих умовних виразів без багатоетапного розгалуження. Приклад тернарного оператора:

```
int x = 5, y = 10;
int max = (x > y) ? x : y;
cout << "Max = " << max << endl;
```

Тут змінна max отримає значення x, якщо умова (x > y) істинна, інакше max дорівнюватиме y. У даному випадку y більше, тому max буде 10. Поради:

Завжди використовуйте фігурні дужки {} після if та else, навіть якщо виконується одна інструкція. Це покращує читабельність і запобігає логічним помилкам.

Слідкуйте за відступами: правильне форматування коду допомагає побачити, який код відноситься до якого блоку умов.

Уникайте занадто глибокого вкладення умов-це ускладнює розуміння. Краще спробувати спростити логіку або розбити її на декілька частин (наприклад, використати функції, про що йтиметься далі).

3 ОПЕРАТОРИ ЦИКЛУ (ЦИКЛІЧНІ КОНСТРУКЦІЇ)

Вступ до поняття: Цикли дозволяють багаторазово виконувати певний блок коду, поки виконується задана умова або протягом визначеної кількості ітерацій. Вони необхідні для організації повторюваних завдань, наприклад, обробки елементів масиву, многократного запиту даних від користувача тощо.

Основні види циклів у C++:

– `while` – цикл з передумовою (умова перевіряється на початку перед кожною ітерацією).

– `do ... while` – цикл з післяумовою (умова перевіряється після виконання тіла циклу, тому тіло завжди виконається хоча б один раз).

– `for` – цикл з лічильником (зручний для відомої наперед кількості ітерацій; містить ініціалізацію, умову і модифікацію лічильника в одному рядку).

3.1 Цикл while

Визначення: Цикл `while` повторює виконання блоку коду, поки деяка логічна умова є істинною. Якщо умова з самого початку хибна, тіло циклу може жодного разу не виконатись.

Синтаксис `while`:

```
while (умова) {
  // тіло циклу: код, який виконується повторно, поки умова істинна
}
```

При вході в цикл перевіряється умова. Якщо вона true, виконується тіло циклу, потім знову перевіряється умова і т.д. Коли умова стане false, цикл завершується і виконання програми продовжується після блоку while.

Приклад while: Обчислення суми чисел від 1 до N:

```
int N;
```

```
cout << "Введіть N: ";
cin >> N;

int sum = 0;
int i = 1;
while (i <= N) {
   sum += i;
   i++;
}
cout << "Сума чисел від 1 до " << N << " = " << sum << endl;
```

Тут змінна i (лічильник) починається з 1 і збільшується на 1 після кожної ітерації. Цикл виконується, поки i <= N. У тілі циклу змінна sum накопичує суму чисел. Після завершення циклу (коли i стане більшою за N) програма виводить результат.

3.2 Цикл do...while

Визначення: Цикл do...while подібний до while, але перевірка умови відбувається **після** виконання тіла циклу. Отже, тіло циклу виконається принаймні один раз, навіть якщо умова спочатку хибна.

Синтаксис do...while:

```
do {
   // тіло циклу: цей код виконається принаймні один раз
} while (умова);
```

Зверніть увагу на крапку з комою ; наприкінці конструкції do...while – вона обов'язкова.

Приклад do...while: Запитувати користувача, чи він хоче продовжити виконання дій:

```
char choice;
do {
```

```
    // ... (якась дія програми) ...
    cout << "Бажаєте продовжити? (y/n): ";
    cin >> choice;
} while (choice == 'y' || choice == 'Y');
```

У цьому прикладі програма виконає певну дію (на місці коментаря) і запитає користувача, чи повторити її. Якщо користувач вводить y або Y, цикл повториться. Навіть якщо користувач відразу введе n, тіло циклу все одно виконається один раз.

3.3 Цикл for

Визначення: Цикл for зручний тоді, коли відома конкретна кількість ітерацій або потрібно організувати цикл з лічильником. Всі елементи керування циклом (ініціалізація, умова продовження, крок зміни лічильника) описуються в заголовку циклу.

Синтаксис for:

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

ініціалізація: виконуються перед першим заходом у цикл (наприклад, оголошення і початкове значення лічильника).

умова: перевіряється перед кожною ітерацією; поки істинна – цикл виконується.

оновлення: виконується в кінці кожної ітерації (наприклад, збільшення лічильника).

Приклад for: Виведення таблиці множення для числа 7:

```
int number = 7;
for (int i = 1; i <= 10; ++i) {
    cout << number << " * " << i << " = " << number * i << endl;
}
```

Тут цикл починається зі значення i = 1 і виконується, поки i <= 10. Після кожної ітерації і збільшується на 1 (++i). В результаті буде виведено множення 7 на числа від 1 до 10.

Ще приклад for: використання кількох змінних у заголовку циклу:

```
for (int i = 0, j = 10; i < j; ++i, --j) {
  cout << "i=" << i << ", j=" << j << endl;
}
```

У цьому складнішому прикладі в заголовку for ініціалізуються дві змінні і і j, і в кожній ітерації одна збільшується, а інша зменшується. Цикл триває, поки i < j. Це демонструє, що в for можна керувати кількома змінними одночасно.

3.4 Управління циклом: break та continue

break: Оператор, що дозволяє негайно перервати виконання циклу. Коли break виконується всередині тіла циклу, програма виходить з циклу (переходить до першої інструкції після циклу).

continue: Оператор, що припиняє поточну ітерацію і переходить до наступної. У циклі for це означає негайне виконання оновлення лічильника і перевірки умови наступної ітерації; у while/do...while – негайну перевірку умови для наступної ітерації.

Приклад використання break і continue:

```
for (int i = 1; i <= 10; ++i) {
  if (i == 5) {
    continue; // пропустити залишок коду і перейти до наступної ітерації для i=6
  }
  if (i == 8) {
    break; // повністю вийти з циклу при i=8
  }
```

```
    cout << i << " ";
}
cout << "\nЦикл завершено.";
```

У цьому коді цикл for виводить числа від 1 до 10, але з особливими випадками: коли i дорівнює 5, виконується continue (число 5 не буде виведено, цикл просто переходить до наступного i); коли i дорівнює 8, виконується break (цикл повністю завершується на цьому, і числа 8,9,10 не будуть опрацьовані).

Поради при роботі з циклами:

– Слідкуйте, щоб умова циклу колись стала хибною. Якщо забути оновити лічильник або неправильно задати умову, можна отримати **нескінченний цикл**, який ніколи не завершиться.

– Якщо потрібно вийти з глибоко вкладеного циклу, можна використовувати мітки або реструктурувати код (наприклад, помістити цикл в функцію і використовувати return для виходу).

– Для перебору масивів C++11 і пізніші стандарти підтримують *range-based for loop*, що спрощує синтаксис перебору елементів контейнера (наприклад: for(int x : array) { ... }). Але на початковому етапі можна засвоїти класичний цикл for з індексами.

4 ОДНОВИМІРНІ МАСИВИ

Вступ до поняття: Одновимірний масив – це структура даних, яка зберігає послідовність елементів одного типу. Масив можна уявити як пронумерований список елементів фіксованого розміру. Масиви зручні для зберігання та обробки наборів значень, наприклад, списку оцінок студентів, набору цін товарів тощо.

4.1 Визначення та оголошення масиву

Означення: Масив – це колекція елементів, доступ до яких здійснюється за індексом (номером позиції). В C++ індекси масиву починаються з 0 (нульова позиція – перший елемент).

Оголошення масиву: Щоб оголосити масив, потрібно вказати тип елементів, ім'я масиву і кількість елементів у квадратних дужках. Наприклад:

```
тип ім'яМасиву[розмір];
```

Приклади оголошення масивів:

```
int numbers[5];      // масив з 5 цілих чисел
double measurements[10]; // масив з 10 чисел з плаваючою комою
char letters[26];    // масив з 26 символів (можна зберігати букви англ. алфавіту)
```

Після оголошення пам'ять під масив виділяється, але значення елементів поки не визначені (в них можуть бути випадкові дані). Щоб уникнути невизначеності, масиви можна ініціалізувати одразу при оголошенні.

Ініціалізація масивів

Ініціалізація під час оголошення: Можна надати початкові значення для елементів масиву, перелічивши їх у фігурних дужках {}. Кількість значень не повинна перевищувати розмір масиву.

```
int daysInMonth[12] = {31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

```
// Масив з 12 елементів, що зберігає кількість днів у кожному місяці року
```

Якщо значень менше, ніж розмір, то інші елементи автоматично заповнюються нулями:

```
int scores[5] = {100, 90, 80};
// Тут scores[0]=100, scores[1]=90, scores[2]=80, а scores[3] і scores[4] будуть 0
```

Можна також дозволити компілятору самому визначити розмір масиву за кількістю наданих значень, залишивши квадратні дужки порожніми:

```
int primes[] = {2, 3, 5, 7, 11}; // масив з 5 елементів, автоматично визначено розмір 5
```

4.2 Доступ до елементів масиву та базові операції

Індексація: Доступ до конкретного елемента масиву здійснюється за його індексом у квадратних дужках. Приклад:

```
int firstPrime = primes[0];   // перший елемент (2)
primes[2] = 13;               // присвоєння нового значення третьому елементу масиву
```

Пам’ятайте, що індекс має бути в діапазоні від 0 до розмір-1. Звернення за межі масиву (наприклад, primes[5] у масиві з 5 елементів) призводить до непередбачуваної поведінки (помилка виконання або некоректні дані).

Обхід масиву: Найчастіше масиви обробляють за допомогою циклів, звертаючись до кожного елемента за індексом. Наприклад, можна вивести всі елементи або обчислити суму:

```
int arr[5] = {10, 20, 30, 40, 50};
int sum = 0;
for (int i = 0; i < 5; ++i) {
```

```
    cout << "Елемент " << i << ": " << arr[i] << endl;
    sum += arr[i];
}
cout << "Сума елементів = " << sum << endl;
```

Тут цикл for проходить індекси від 0 до 4 (всього 5 елементів). Змінна і використовується для доступу до arr[i]. Послідовно виводяться значення елементів і паралельно обчислюється їх сума.

Зміна розміру масиву: Розмір статичного масиву (оголошеного як тип ім'я[константа]) є фіксованим і не може бути змінений під час виконання. Якщо потрібні структури динамічного розміру, використовують *динамічні масиви* (виділення пам'яті через new) або контейнер std::vector, але ці теми виходять за рамки базового конспекту. Для початку ми працюємо зі статичними масивами фіксованої довжини.

4.3 Практичні приклади з масивами

Знаходження мінімуму/максимуму: Перебрати масив, порівнюючи елементи, і знайти найменший або найбільший.

Лінійний пошук: Перевіряти кожен елемент масиву, щоб знайти задане значення.

Зворотній вивід масиву: Вивести елементи у зворотному порядку, ітеруючи від останнього індексу до першого.

Копіювання масиву: Створити новий масив і скопіювати в нього значення з іншого через цикл.

Приклад: Знаходження мінімального елемента в масиві:

```
int data[] = {42, 15, 23, 8, 16};
int n = 5;
int minVal = data[0];
for (int i = 1; i < n; ++i) {
  if (data[i] < minVal) {
    minVal = data[i];
  }
}
```

```
cout << "Найменше значення: " << minVal << endl;
```

Програма бере перший елемент як початковий мінімум і потім перевіряє кожен наступний, оновлюючи minVal, якщо знаходить менше значення.

5 МАТРИЦІ (ДВОВИМІРНІ МАСИВИ)

Вступ до поняття: Матриця в програмуванні – це двовимірний масив, тобто структура даних у вигляді таблиці, де дані розташовані по рядках і стовпцях. Матриці дозволяють зберігати, наприклад, таблицю чисел, розклад занять (де рядки – дні, стовпці – години), результати експериментів у двох вимірах тощо.

5.1 Оголошення та ініціалізація матриць

Оголошення 2D-масиву: Для оголошення двовимірного масиву потрібно вказати два розміри – кількість рядків і кількість стовпців:

```
тип ім'яМатриці[кількістьРядків][кількістьСтовпців];
```

Приклади оголошення матриць:

```
int matrix[3][4];        // матриця 3x4 з цілих чисел (3 рядки, 4 стовпці)
double table[5][2];   // матриця 5x2 з чисел з плаваючою комою
char board[3][3];       // 3x3 масив символів (наприклад, для гри "Хрестики-нулики")
```

Так само, як і з одновимірними масивами, при оголошенні пам'ять виділяється, але значення елементів не визначені за замовчуванням.

Ініціалізація матриці: Можна ініціалізувати матрицю під час оголошення, задавши значення для кожного рядка в окремих фігурних дужках:

```
int identity[3][3] = {
  {1, 0, 0},
  {0, 1, 0},
  {0, 0, 1}
};
// 3x3 одинична матриця (діагональні елементи = 1, решта = 0)
```

Кожна внутрішня дужка відповідає рядку матриці. Кількість значень у кожному рядку повинна відповідати кількості стовпців.

5.2 Доступ до елементів матриці

Для доступу до елементів використовуються два індекси: перший – номер рядка, другий – номер стовпця. Індекси теж починаються з 0.

Приклад доступу:

```
int val = matrix[1][3]; // доступ до елементу другого рядка та четвертого стовпця (індекси 1 і 3)
matrix[0][0] = 5;    // присвоєння значення 5 елементу в першому рядку і першому стовпці
```

Важливо, щоб індекси не виходили за межі: для matrix[3][4] допустимі індекси рядків 0..2 і стовпців 0..3.

Обхід двовимірного масиву (матриці)

Для проходження по всіх елементах матриці зазвичай використовуються вкладені цикли: зовнішній цикл перебирає рядки, а внутрішній – стовпці.

Приклад: Виведення всіх елементів матриці 3x4:

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

for (int i = 0; i < 3; ++i) {       // цикл по рядках
  for (int j = 0; j < 4; ++j) {    // цикл по стовпцях
    cout << matrix[i][j] << " ";
  }
  cout << endl; // перехід на новий рядок після кожного рядка матриці
}
```

Цей код пройде по всіх елементах матриці matrix і виведе їх у вигляді таблиці 3x4. Змінна i вказує номер рядка, j – номер стовпця.

5.3 Практичні приклади операцій з матрицями

Обчислення суми всіх елементів: ініціювати суму 0 і додати кожен елемент (два вкладених цикли).

Пошук максимального елемента матриці: аналогічно одновимірному масиву, але з двома індексами.

Транспонування матриці: створити нову матрицю, де рядки і стовпці поміняні місцями.

Додавання/віднімання двох матриць: поелементно додати значення з двох матриць однакового розміру.

Множення матриць: більш складна операція, яка вимагає третього вкладеного циклу (але це вже просунутий матеріал).

Приклад: Обчислення суми елементів 2D-масиву:

```
int rows = 2, cols = 3;
int matrix[2][3] = {
  {3, 5, 1},
  {7, 2, 4}
};
int totalSum = 0;
for (int i = 0; i < rows; ++i) {
  for (int j = 0; j < cols; ++j) {
    totalSum += matrix[i][j];
  }
}
cout << "Сума всіх елементів матриці = " << totalSum << endl;
```

Програма обчислює суму всіх чисел в матриці 2x3. Змінні rows і cols зберігають розміри, що полегшує зміну розміру матриці у майбутньому (щоб не правити умови циклів вручну).

Зауваження:

– В пам'яті комп'ютера двовимірний масив зберігається построчно (за замовчуванням у C++ використовується *row-major*

order). Це означає, що елементи першого рядка йдуть підряд, потім другого і т.д.

– Ви можете оголосити масив більшої розмірності (3, 4 вимірів), додаючи більше індексів, але працювати з такими структурами складніше. У практиці багатовимірні масиви понад 3 вимірів використовують рідко; за потреби складніших структур даних застосовують класи або спеціалізовані бібліотеки.

6 ПОКАЖЧИКИ, ПОСИЛАННЯ ТА ДИНАМІЧНЕ ВИДІЛЕННЯ ПАМ'ЯТІ В C++

Покажчики (англ. pointers) і посилання (англ. references) – одні з ключових понять у C++, що дозволяють маніпулювати пам'яттю та ефективніше працювати зі змінними. Розуміння цих концепцій є важливим для початківців, адже з їх допомогою можна передавати дані у функції без копіювання, динамічно виділяти пам'ять під час роботи програми тощо.

6.1 Показчики в C++

Що таке покажчик?

Покажчик – це змінна, яка містить адресу комірки пам'яті, де зберігається інша змінна або дані.

Іншими словами, значенням покажчика є адреса в пам'яті. Як і звичайні змінні, покажчики мають свій тип – тип даних, на який вони вказують. Наприклад, покажчик типу int* зберігає адресу цілої змінної int. Важливо розуміти, що сам покажчик займає певну кількість байтів (як правило, 4 байти на 32-бітній системі або 8 байтів на 64-бітній) незалежно від того, на який тип він посилається. Щоб уявити роботу з пам'яттю, згадаємо, що кожна змінна при оголошенні отримує свою адресу. Наприклад, якщо оголосити int a = 7;, то в пам'яті виділиться місце для зберігання значення 7 (для типу int це 4 байти) за певною адресою (умовно кажучи, адреса може бути, наприклад, 0x0046FCF0). Покажчик же може зберігати цю адресу. Таким чином, замість значення 7 покажчик міститиме 0x0046FCF0 – адресу, за якою лежить 7. Операція взяття адреси (&) дозволяє отримати адресу змінної в пам'яті

Наприклад, &a дає нам адресу змінної a. Операція розіменування (*) дозволяє, маючи адресу, отримати значення, що знаходиться за цією адресою.

Якщо p – покажчик, то *p означає "значення за адресою, яка зберігається в p". Після розіменування покажчика ми фактично звертаємося до тієї змінної, на яку він вказує.

Покажчики часто називають ще "вказівниками" – обидва терміни вживаються як синоніми в українській мові.

Оголошення та ініціалізація покажчика

Оголосити покажчик в C++ можна, вказавши тип даних, на який він вказуватиме, та символ * перед ім'ям змінної. Загальний синтаксис оголошення:

тип_даних *ім'я_покажчика;

Наприклад:

```
int *ptr;  // оголошено покажчик ptr, що може зберігати адресу int
double *dp; // оголошено покажчик dp, що може зберігати адресу double
```

Після оголошення покажчик містить невизначене значення, оскільки йому не присвоєно жодної адреси. Як і у випадку зі звичайними змінними, покажчики не ініціалізуються автоматично при створенні

Якщо одразу не надати покажчику значення (адресу), він буде вказувати на випадкову область пам'яті (містити "сміття"), що дуже небезпечно при доступі до нього. Тому безпечна ініціалізація покажчика полягає або в негайному присвоєнні йому коректної адреси, або в установці значення nullptr (нульового покажчика), якщо поки що немає адреси для зберігання. Розглянемо приклад оголошення та ініціалізації покажчика:

```
int a = 5;      // звичайна ціла змінна
int *p = &a;    // покажчик p отримує адресу змінної a
std::cout << a << "\n";  // виведе 5 (значення змінної a)
std::cout << &a << "\n";  // виведе адресу змінної a, наприклад 0x61ff08
std::cout << p << "\n";    // виведе ту саму адресу (значення покажчика p)
std::cout << *p << "\n"; // розіменування p, виведе 5 (значення за адресою)
```

У цьому прикладі змінна a містить значення 5. Змінна p – це покажчик типу int*, якому присвоєно адресу змінної a через вираз &a. Після цього кажемо, що "покажчик p вказує на a". Вивід демонструє,

що p містить ту саму адресу, що і &a. А розіменування *p дозволяє отримати значення за цією адресою, тобто фактично значення змінної a. Таким чином, *p еквівалентне a в цьому контексті.

При оголошенні покажчика зірочку * можна писати як поруч із типом, так і поруч із назвою змінної (int* p або int *p). Головне – дотримуватися обраного стилю послідовно. Часто рекомендують писати * безпосередньо біля імені змінної (як у прикладі вище) для кращої читабельності коду

6.2 Арифметика покажчиків

Покажчики підтримують обмежений набір арифметичних операцій. Зокрема, можна збільшувати або зменшувати покажчик (операції інкременту ++p та декременту --p), додавати або віднімати ціле число, а також обчислювати різницю між двома покажчиками одного типу. Арифметика покажчиків враховує розмір типу, на який покажчик вказує.

Це означає, що при збільшенні покажчика на 1 він починає вказувати на наступний елемент відповідного типу в пам’яті, а не просто зміщується на один байт.

Інкремент (++): збільшує адресу, що зберігається в покажчику, на величину розміру типу. Наприклад, якщо ptr має тип int* і зберігає адресу 1000, то після ptr++ його значення стане 1004 на системі, де int займає 4 байти (тобто покажчик тепер вказує на наступну комірку для int).

Аналогічно, для double* (8 байтів) інкремент збільшив би адресу на 8. Це зручно для ітерації по елементах масиву.

Декремент (--): зменшує адресу в покажчику на розмір типу. Якщо ptr вказував на другу комірку масиву (ptr = 1004 умовно), то після ptr-- він знову вказуватиме на першу (1000)

Приклад (інкремент/декремент покажчика):

```
int arr[3] = {10, 20, 30};
int *p = arr;          // еквівалентно int *p = &arr[0];
std::cout << *p << "\n"; // виведе 10 (arr[0])
p++;                 // тепер p вказує на наступний елемент (arr[1])
std::cout << *p << "\n"; // виведе 20 (arr[1])
```

```
p++;                // p вказує на arr[2]
std::cout << *p << "\n"; // виведе 30 (arr[2])
p--;                // p знову вказує на arr[1]
std::cout << *p << "\n"; // виведе 20 (arr[1])
```

В цьому прикладі покажчик p проходиться по елементах масиву arr шляхом збільшення на 1. Зауважте: після p++ покажчик переміщується до наступної комірки типу int (адреса збільшується на 4 байти). Декремент p-- повертає його на попередню адресу.

Додавання або віднімання цілих чис: можна виконувати вирази на кшталт p + n або p - n, де n – ціле число. Це еквівалентно багаторазовому інкременту або декременту. Наприклад, p + 3 – покажчик на третій елемент після того, на який зараз вказує p. Якщо p вказує на arr[0], то p + 3 буде вказувати на arr[3] (тобто перший елемент поза межами масиву arr[0..2]). Важливо, що дозволяється виходити покажчиком на одну позицію за останній елемент масиву (цей покажчик не можна розіменовувати, але його використовують для позначення кінця секвенції, наприклад, у циклах). Будь-яке інше переміщення покажчика за межі виділеної пам’яті є недопустимим і спричиняє невизначену поведінку.

Різниця покажчиків: якщо два покажчики вказують на елементи одного масиву (або один на позицію в масиві, інший на позицію в тому ж масиві чи на одразу за останнім елементом), їх різницю можна обчислити. Віднімання одного покажчика від іншого дає кількість елементів між ними у цьому масиві (ціле значення типу ptrdiff_t).

Наприклад, якщо

```
int *p1 = &arr[0];
int *p2 = &arr[2];
auto diff = p2 - p1;
```

то diff буде рівним 2, оскільки між адресами &arr[0] і &arr[2] знаходиться 2 елементи типу int. Якщо ж покажчики вказують на різні масиви, результат різниці не визначений.

Порівняння покажчиків: можна порівнювати покажчики операціями ==, !=, <, > тощо. Але семантично ці порівняння мають сенс лише для покажчиків, що вказують всередині одного блоку пам’яті

(наприклад, одного масиву). Порівняння == і != допустиме і для покажчиків з різних областей (вони просто перевіряють, чи адреси однакові).

Інші порівняння (<, > і т.д.) варто використовувати тільки для покажчиків одного масиву або якщо один з них – це кінцевий (одразу за останнім елементом).

6.3 nullptr та нульові покажчики

У покажчика може бути спеціальне «нульове» значення, яке означає, що він ні на що не вказує/

Такий покажчик називають нульовим. Історично в мовах C/C++ для позначення нульового покажчика використовували константу NULL (що зазвичай визначена як 0) або просто literal 0. Починаючи з C++11, з'явилося спеціальне ключове слово nullptr для позначення нульового покажчика.

nullptr має власний тип std::nullptr_t і неявно перетворюється на будь-який тип покажчика. Рекомендується використовувати саме nullptr для ініціалізації або присвоєння, коли покажчик не має вказувати на жодний об'єкт. Це підвищує ясність коду (на відміну від просто 0 чи NULL, які можуть трактуватися і як цілочислові значення). Приклад використання nullptr:

```
int *ptr1 = nullptr;       // ptr1 поки ні на що не вказує (нульовий покажчик)
int *ptr2 = new int(42);   // виділено пам'ять під int, ptr2 вказує на неї
if (ptr1 == nullptr) {
  std::cout << "ptr1 is null\n";     // цей блок виконається
}
if (ptr2 != nullptr) {
  std::cout << "ptr2 points to " << *ptr2 << "\n"; // виконається, *ptr2 = 42
}
```

Тут ptr1 ініціалізовано як нульовий покажчик, а ptr2 отримує адресу від оператора new (про new детальніше далі). Перевірка ptr1 == nullptr показує, що ptr1 порожній, а ptr2 != nullptr підтверджує, що пам'ять успішно виділена. Перед розіменуванням будь-якого покажчика корисно робити подібну перевірку на nullptr, щоб програма не намагалася доступитися до неіснуючої адреси. Чому це важливо? Ініціалізація покажчика значенням nullptr убезпечує від випадкового використання сміттєвого значення. Нульовий покажчик легко перевірити в умовах (if (ptr) ... або if (ptr == nullptr) ...). Якщо ж покажчик неініціалізований, він може містити довільну адресу, і перевірка if (ptr) не має сенсу. Таким чином, завжди надавайте покажчикам відоме значення при оголошенні: або реальну адресу, або nullptr.

6.4 Посилання в C++

Що таке посилання?

Посилання (англ. reference) – це альтернативне ім'я для вже існуючої змінної. Посилання працює як псевдонім об'єкта, на який вказує. Її можна уявити як друге ім'я тієї ж комірки пам'яті. На відміну від покажчика, посилання не є окремою змінною-адресою, а скоріше "прізвиськом" для іншої змінної. У C++ синтаксис оголошення посилання схожий на синтаксис покажчика, але використання зовсім інше. Щоб оголосити посилання, використовують символ амперсанда & після типу даних (в оголошенні, а не у виразі). Формат:

тип_даних &ім'я_посилання = змінна;

Посилання повинна бути проініціалізована одразу під час оголошення – їй потрібно одразу присвоїти, на що вона посилається. Приклад:

```
int value = 7;      // звичайна змінна
int &ref = value;   // посилання ref на змінну value
```

У цьому прикладі ref – це посилання на value. Після цього будь-які операції з ref фактично будуть діяти на змінну value. Важливо: у декларації int &ref = value; символ & означає саме "посилання на", а не "взяти адресу". В контексті оголошення амперсанд виступає частиною

типу (тип int& – “посилання на int”), а не оператором. Тобто тут ref – псевдонім змінної value. Натомість у виразах, поза оголошенням, & – це оператор взяття адреси, як ми розглядали раніше.

Використання посилань і відмінності від покажчиків

Після оголошення посилання можна використовувати так само, як і звичайну змінну. Продовжимо приклад:

```
std::cout << value << "\n"; // 7
std::cout << ref << "\n";  // 7 (ref відображає значення value)
ref = 10;                  // змінюємо значення через посилання
std::cout << value << "\n"; // 10 (value змінилося)
```

Спочатку і value, і ref відображають одне і те ж значення 7. Після присвоювання ref = 10; змінна value стала рівною 10. Це відбувається тому, що ref – не копія value, а саме сам value під іншим іменем. Фактично операція ref = 10; еквівалентна value = 10;. Основні відмінності посилань від покажчиків:

Посилання не займає окремої пам’яті як змінна-адреса. Вона є альтернативним іменем існуючого об’єкта. У реалізації компілятор може використати покажчик “під капотом” для посилання, але на рівні мови у нас немає доступу до цієї адреси чи можливості зсувати посилання.

Посилання має бути прив’язана до об’єкта при створенні. Її не можна залишити неоприділеною і пізніше змусити посилатися на щось – потрібно одразу ініціалізувати. Після цього вона завжди посилається на один і той самий об’єкт. На відміну від покажчика, посилання не може бути переприсвоєна на інший об’єкт. В прикладі вище не можна зробити int x = 5; ref = x; щоб ref почала посилатися на x – така операція змінить значення value, а не зв’язок посилання.

Посилання не може бути “нульовою”. Теоретично, є спосіб отримати посилання, яка не прив’язана до об’єкта (наприклад, посилання на об’єкт після його видалення), але це вже порушення правил мови і призводить до невизначеної поведінки. В стандартному C++ посилання завжди повинна щось означати. Тому немає аналога nullptr для посилань (виняток – спеціальні випадки з std::reference_wrapper, але то інше).

Для використання посилання не потрібні оператори * і &. Ви працюєте з посиланням так, ніби це сама змінна. Немає операції розіменування для посилання – вона і так вже “розіменована”. У прикладі ref поводиться як звичайний int. З іншого боку, щоб отримати адресу об’єкта, на який посилання вказує, все одно можна застосувати оператор &: в нашому прикладі &ref і &value дадуть одну й ту саму адресу.

Посилання більш безпечні та зручні у багатьох випадках, особливо для передачі параметрів у функції. Використовуючи посилання як параметр функції, ми дозволяємо функції працювати з оригінальним об’єктом без копіювання, але водночас не турбуємося про перевірку нульових значень, як із покажчиками. Наприклад, замість функції void swap(int *a, int *b) можна написати void swap(int &a, int &b) – і всередині функції працювати з a та b як зі звичайними змінними, знаючи, що це посилання на аргументи. При цьому виклик виглядає природно: swap(x, y) (без амперсандів), що легше і безпечніше для користувача.

Підсумовуючи: посилання в C++ – це зручний механізм доступу до того самого об’єкта під іншим іменем. Вони тісно пов’язані з покажчиками (насправді реалізація посилання опирається на покажчики), але приховують від програміста деталі роботи з пам’яттю, забезпечуючи більш високий рівень абстракції. У простих випадках, коли потрібен саме доступ до існуючої змінної, краще використовувати посилання. Покажчики ж необхідні, коли потрібна більша гнучкість: динамічне виділення пам’яті, можливість “ні на що не вказувати” (nullptr), змінювати, куди вказує, виконувати арифметику адрес тощо.

6.5 Динамічне виділення пам’яті

На цей момент ми розглянули покажчики і посилання, які працюють з уже існуючими змінними (адресами цих змінних). А тепер розглянемо, як у C++ можна створювати нові змінні “на льоту” під час виконання програми, використовуючи динамічне виділення пам’яті. Для цього застосовуються спеціальні оператори new (виділення) та delete (звільнення пам’яті). У стандартному C++ існує три моделі виділення пам’яті: статична (для глобальних/статичних змінних), автоматична (для локальних змінних, що створюються на стеку і

автоматично знищуються при виході з блоку) та динамічна (для об’єктів, що створюються у спеціальній області пам’яті – купа або heap). Динамічне виділення дозволяє керувати часом життя змінних вручну – ви самі вирішуєте, коли виділити пам’ять і коли її звільнити. Це особливо корисно, коли розмір або кількість необхідних об’єктів невідомі під час компіляції і визначаються у процесі роботи програми (наприклад, розмір масиву задає користувач).

6.6 Оператор new

Оператор new виділяє пам’ять в області купи (heap) під об’єкт заданого типу і повертає покажчик на цю область. Формат використання:

Тип *покажчик = new Тип;

Цей вираз запитує в операційної системи пам’ять розміром під один об’єкт заданого типу. Якщо пам’ять доступна, вона буде виділена, а оператор new поверне адресу початку виділеного блоку. Ця адреса приводиться до потрібного типу покажчика і зберігається у змінній. Наприклад:

```
int *p = new int;      // динамічно виділяється пам'ять під один int, p отримує адресу
*p = 42;               // записуємо значення 42 у виділену пам'ять
std::cout << *p << "\n"; // виведе 42
```

В цьому коді new int виділяє блок пам’яті розміром 4 байти (припустимо, розмір int = 4) і повертає адресу цього блоку. Ми зберігаємо адресу в покажчику p. Тепер p вказує на область пам’яті, яка поки що не має імені, але в якій можна зберігати ціле число. Записавши *p = 42;, ми помістили число 42 в цю область. Виведення підтверджує, що за адресою в p знаходиться значення 42. Можна одразу ініціалізувати виділену пам’ять значенням, використовуючи синтаксис конструктора: new int(42) створить цілочисельний об’єкт зі значенням 42. . У початкових програмах зазвичай опускають цю перевірку, але слід пам’ятати, що у критичних системах вона може знадобитися.

6.7 Оператор delete

Оператор delete звільняє раніше виділену пам’ять, повертаючи її назад операційній системі. Синтаксис:

```
delete покажчик;
```

де покажчик – це адреса, отримана від new (або nullptr, тоді delete нічого не робить). Після виконання delete вказаний блок пам’яті помічається як вільний і може бути повторно використаний програмою (при наступних викликах new) або іншими програмами в системі. Важливо: delete потрібно викликати тільки для тих покажчиків, які було отримано через new. Застосування delete до покажчика на статичну або автоматичну змінну призведе до помилки. Повернімося до нашого прикладу з p = new int;. Після того, як ми використали виділену пам’ять, потрібно її звільнити:

```
delete p;  // звільняємо пам'ять, яку було виділено під int
p = nullptr; // зануляємо покажчик для безпеки
```

Після операції delete p; пам’ять, на яку вказувала p, більше нам не належить. Використовувати цей покажчик далі небезпечно, доки ми не присвоїмо йому нове значення. Щоб уникнути випадкового розіменування "висячого" покажчика (який вказує на вже звільнену область), ми встановили p = nullptr;. Тепер p нікуди не вказує, і спроба розіменувати буде виявлена як помилка (або нічого не робитиме, якщо ми перевірятимемо if (p) перед цим).

Важливо: Кожен виклик new повинен мати відповідний виклик delete, коли виділена пам’ять більше не потрібна. Невивільнена пам’ять призводить до витоків пам’яті (memory leaks): програма поступово “з’їдає” доступну пам’ять, що може стати критичною проблемою для довготривалих програм. Завжди слідкуйте, щоб будь-який шлях виконання програми, де викликається new, зрештою викликав і delete для тієї ж пам’яті.

6.8 Динамічні масиви

Динамічне виділення пам’яті дозволяє створювати не тільки окремі змінні, а й цілі масиви змінних. Синтаксис трохи відрізняється: для масивів використовуються форми new[] і delete[]. Виділення динамічного масиву:

Тип *ім’я = new Тип[кількість];

Це виділить блок пам’яті під зазначену кількість елементів заданого типу і поверне покажчик на перший елемент масиву. Елементи будуть розміщені підряд в пам’яті. Наприклад:

```
int n = 5;
int *arr = new int[n];  // виділення масиву з 5 цілих
for (int i = 0; i < n; ++i) {
   arr[i] = i * 2;    // ініціалізуємо елементи: 0, 2, 4, 6, 8
}
std::cout << arr[3] << "\n"; // звернення до 4-го елемента (індекс 3), виведе 6
```

Тут ми динамічно виділили масив з 5 елементів типу int. Змінна arr отримала адресу першого елемента масиву (еквівалентно &arr[0]). Тепер з arr можна працювати як із звичайним масивом: використовувати індекси arr[i], ітеруватися по елементах і т.д. Різниця в тому, що пам’ять під цей масив була виділена в купі і не буде автоматично вивільнена після виходу з області видимості змінної arr. Ми повинні вручну звільнити її, коли масив більше не потрібен. Звільнення динамічного масиву:

delete[] ім’я;

Для масиву необхідно використовувати саме форму delete[], щоб звільнити весь блок пам’яті, на який вказує покажчик. У нашому прикладі правильним буде:

```
delete[] arr;  // звільнення пам'яті, виділеної під масив з 5 int
arr = nullptr; // зробимо покажчик нульовим
```

Після цього пам’ять під усі 5 елементів повернуто системі. Занулення arr – гарна практика: тепер arr не вказує на звільнену пам’ять. Приклад повністю (динамічний масив з розміром від користувача):

```
#include <iostream>
int main() {
  std::cout << "Введіть кількість елементів: ";
  int length;
  std::cin >> length;
  int *array = new int[length];  // new[] виділяє масив довжини length
  for (int i = 0; i < length; ++i) {
    array[i] = i + 1;        // заповнюємо масив значеннями 1,2,3,...
  }
  std::cout << "Масив: ";
  for (int i = 0; i < length; ++i) {
    std::cout << array[i] << " ";
  }
  std::cout << "\n";
  delete[] array;            // звільняємо пам'ять масиву
  array = nullptr;           // зануляємо покажчик
  return 0;
}
```

Вихід, наприклад, для введеного значення 5:

Введіть кількість елементів: 5

Масив: 1 2 3 4 5

У цьому коді користувач задає розмір масиву під час виконання (динамічна пам’ять дає таку гнучкість). Ми виділяємо відповідний масив цілих, заповнюємо його, використовуємо, а потім звільняємо за допомогою delete[]. Без new і delete ми не могли б створити масив змінного розміру всередині програми – лише масив фіксованого розміру, відомого на етапі компіляції. Примітки щодо динамічних масивів:

Ініціалізувати динамічний масив можна схожим чином, як статичний, але в межах дужок треба зазначити кількість. Наприклад: int *a = new int[3]{2,4,6}; виділить масив з 3 елементів і ініціалізує їх 2,4,6 відповідно. Якщо кількість елементів в ініціалізаторі менша, ніж розмір, інші будуть 0. Якщо більша – це помилка.

Змінну з динамічним масивом часто називають “покажчиком на перший елемент масиву”. Ви можете зберігати цей покажчик в іншій змінній, передавати у функції, виконувати арифметику (інкремент проходить по елементах), як і з покажчиками на звичайні змінні.

Якщо потрібно змінити розмір динамічного масиву, доведеться створити новий масив потрібного розміру і, за потреби, скопіювати туди елементи зі старого, після чого звільнити старий масив. Розмір виділеного масиву “на льоту” змінити не можна. (Для гнучких змінних масивів на практиці краще використовувати контейнер std::vector із стандартної бібліотеки, але це тема іншої лекції.)

Не забувайте викликати delete[] для кожного new[] коли масив більше не потрібен! Витоки пам’яті в масивах так само небезпечні, як і з окремими об’єктами.

7 ФУНКЦІЇ

Вступ до поняття: Функція в C++ – це самостійний блок коду, призначений для виконання певної логічно завершеної дії. Використання функцій дозволяє розділити програму на менші підзадачі, уникнути повторення коду (через повторне використання функцій) і зробити програму більш зрозумілою. Для початківців важливо навчитися оголошувати та викликати власні функції, а також розуміти, як передаються дані у функцію і повертаються з неї.

7.1 Оголошення та визначення функцій

Прототип (оголошення) функції: Перед тим, як викликати функцію в main або іншій функції, компілятор має знати про її існування. Прототип функції – це рядок, який оголошує ім'я функції, список параметрів і тип, який вона повертає, без реалізації тіла. Прототип зазвичай розміщують перед функцією main або в заголовочному файлі. Наприклад:

```
int add(int a, int b); // прототип функції, що приймає два int і повертає int
```

Визначення функції: Це реалізація функції, де пишеться сам код (тіло функції). Визначення включає прототип плюс реалізацію в фігурних дужках. Наприклад:

```
int add(int a, int b) {
   int result = a + b;
   return result;
}
```

Тут int add(int a, int b) – сигнатура (підпис) функції, return result; повертає значення результату. Тип, зазначений перед ім'ям (int), – це тип, який функція повертає. Якщо функція не повертає значення, використовується тип void. Виклик функції: Щоб використати функцію, достатньо написати її ім'я з відповідними аргументами.

Наприклад, int sum = add(5, 7); викличе функцію add з аргументами 5 і 7, і збереже результат (12) в змінній sum.

7.2 Параметри функцій і передача даних

Функції можуть приймати дані через параметри. При виклику функції в дужках ми передаємо аргументи, які копіюються в параметри.

Передача за значенням: За замовчуванням, коли тип параметра – простий (не посилання і не вказівник), аргумент копіюється. Тобто функція працює зі копією значення, а не з самим оригіналом. Зміни цього параметра в функції не впливають на змінну, що була передана.

Передача за посиланням: Якщо ми хочемо, щоб функція могла змінити передану змінну або щоб уникнути копіювання великих об'єктів, параметр можна оголосити як посилання (&). У цьому випадку функція оперує безпосередньо змінною, переданою їй, без створення копії.

Передача за вказівником: Альтернативно, можна передати адресу змінної (вказівник). Всередині функції тоді працюємо з вказівником або розіменовуємо його для доступу до даних. Для початківця цей підхід складніший, тому спершу зосередимося на передачі за значенням і за посиланням.

Приклади:

```
void printDouble(int x) {
  cout << "Подвоєне значення: " << x * 2 << endl;
}

void incrementByFive(int &y) {
  y = y + 5;
}
```

printDouble приймає копію змінної x і лише виводить подвоєне значення, не змінюючи оригінал.

incrementByFive приймає посилання на змінну y і збільшує значення цієї змінної на 5. Якщо викликати int a = 10; incrementByFive(a);, то після виклику значення a стане 15, бо функція оперувала оригіналом через посилання.

7.3 Значення, що повертається

Функція може повернути результат певного типу, використовуючи ключове слово return. Після виконання оператора return функція завершується, а в те місце програми, звідки її викликали, повертається вказане значення. Приклад функції, що повертає значення:

```
double square(double x) {
  return x * x;
}
```

Ця функція обчислює квадрат числа і повертає його. Виклик double y = square(4.5); призведе до того, що змінна y матиме значення 20.25. Якщо функція оголошена з типом повернення void, вона не повертає значення і оператор return може використовуватися без зазначення значення тільки для дострокового виходу з функції.

Приклад: створення та використання власних функцій

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

// Прототипи функцій
int add(int a, int b);
double average(int x, int y);
void printLine();

int main() {
  int num1 = 10, num2 = 20;
  // Виклик функції add та збереження результату
  int sum = add(num1, num2);
  cout << "Сума: " << sum << endl;

  // Виклик функції average та безпосереднє використання результату у виводі
  cout << "Середнє: " << average(num1, num2) << endl;
```

```
    // Виклик функції без повернення (void)
    printLine();

    return 0;
}

// Визначення функцій
int add(int a, int b) {
    return a + b;
}

double average(int x, int y) {
    double sum = x + y;
    return sum / 2.0;
}

void printLine() {
    cout << "--------------------------" << endl;
}
```

У цьому прикладі визначено три власні функції: add (повертає суму двох цілих), average (повертає середнє значення двох цілих як double) і printLine (виводить лінію-роздільник і нічого не повертає). У функції main ми демонструємо виклики цих функцій і використання їх результатів.

7.4 Рекурсивні функції

Рекурсія – це коли функція викликає сама себе. Рекурсивні рішення корисні для деяких задач (наприклад, обхід дерев, обчислення факторіалу, чисел Фібоначчі тощо). Проте з рекурсією треба бути обережним: обов’язково має бути умова виходу, інакше виникне нескінченний рекурсивний виклик. Приклад рекурсії (факторіал числа):

```
int factorial(int n) {
  if (n <= 1)
    return 1;
  else
    return n * factorial(n - 1);
}
```

Ця функція викликає сама себе, кожного разу зменшуючи n, поки n не стане 1. Факторіал 1 і 0 визначений як 1, що і використано як базовий випадок для виходу з рекурсії.

8 ОПЕРАЦІЇ З РЯДКОВИМИ ЗМІННИМИ

Вступ до поняття: Рядкові змінні використовуються для зберігання текстових даних – послідовності символів (літер, цифр, пробілів, розділових знаків тощо). В C++ існує декілька способів роботи з рядками:

Можна використовувати масив символів (C-рядки, "c-string"), де останнім символом є спеціальний '\0' (нульовий символ, що позначає кінець рядка).

Можна використовувати клас std::string з бібліотеки <string>, який надає зручні можливості для роботи з текстом і автоматично керує пам’яттю для рядка.

Для початківців рекомендується клас std::string, оскільки з ним простіше уникнути помилок з переповненням буфера і не потрібно вручну контролювати символ '\0'.

8.1 Використання масиву символів (C-рядків)

Оголошення: Рядок у вигляді масиву символів – це масив типу char. Треба подбати, щоб розмір масиву був на 1 більший за максимальну довжину тексту (додатковий символ для '\0').

```
char name[20]; // може містити слово довжиною до 19 символів + 1 символ '\0'
```

Присвоєння: Можна присвоїти початкове значення при оголошенні:

```
char hello[] = "Привіт";
```

В цьому випадку компілятор сам розрахує розмір (6 символів: П, р, и, в, і, т, і додасть '\0'). Введення та виведення C-рядків:

Для введення можна використовувати cin, але варто бути обережним: оператор >> зчитує в масив символів слово до першого пробілу. Для зчитування цілої стрічки (включно з пробілами) використовують функцію cin.getline(масив, розмір).

Для виведення достатньо використовувати cout << name; – при виведенні масиву char бібліотека вважатиме його рядком, якщо він коректно завершується '\0'.

Приклад з масивом char:

```
char str[50];
cout << "Введіть ваше ім'я: ";
cin.getline(str, 50);
cout << "Привіт, " << str << "!" << endl;
```

Тут ми виділили масив на 50 символів, очікуючи, що ім'я користувача не перевищить 49 символів. Використано cin.getline для зчитування повного рядка (ім'я може складатися з кількох слів). Потім програма вітає користувача по імені. Основні операції з С-рядками:

Існує бібліотека <cstring> з корисними функціями для роботи з С-рядками:

strlen(cstr) – повертає довжину рядка (без '\0').

strcpy(dest, source) – копіює рядок source в буфер dest (потрібно слідкувати, щоб dest мав достатній розмір).

strcat(dest, source) – дописує (конкатенує) рядок source в кінець dest.

strcmp(str1, str2) – порівнює два рядки (повертає 0, якщо рівні, >0 або <0, якщо різні в лексикографічному порядку).

Робота з цими функціями потребує обережності, щоб буфери мали достатній розмір. Некоректне використання може призвести до переписування пам’яті (buffer overflow). Саме тому використовувати std::string безпечніше для початківців.

8.2 Використання std::string

Підключення: Щоб використовувати тип std::string, потрібно включити заголовок <string> і (якщо ми не використовуємо using namespace std;) звертатись до нього через простір імен std::. Оголошення та присвоєння:

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

```
string s1;            // порожній рядок
string s2 = "Hello";    // ініціалізація літералом
string s3("World");     // альтернативна ініціалізація
s1 = "Привіт, ";       // присвоєння значення
string s4 = s1 + "світе!"; // конкатенація рядків через оператор +
```

std::string може зберігати рядки довільної довжини (обмежені лише обсягом пам'яті), і вам не потрібно вручну слідкувати за '\0' – клас робить це сам. Введення та виведення std::string:

Введення: оператор >> для string зчитує одне слово (до пробілу). Для зчитування повної лінії (включаючи пробіли) використовується функція std::getline(cin, змінна_string).

Виведення: cout << змінна_string; виведе весь рядок.

Основні методи і операції std::string:

s.length() або s.size() – отримати довжину рядка.

s.empty() – перевірити, чи рядок порожній.

s.push_back('A') – додати символ в кінці рядка.

s.append("text") – додати строку в кінці (конкатенація).

Оператор + або += – також для конкатенації.

Доступ до символів: s[i] або s.at(i) (другий варіант виконує перевірку виходу за межі).

s.substr(pos, len) – витягти підрядок довжини len, починаючи з позиції pos.

s.find(text) – знайти підрядок text у рядку s (повертає індекс початку або std::string::npos, якщо не знайдено).

s.replace(pos, len, new_text) – замінити частину рядка на інший текст.

Приклади роботи з std::string:

```
string text;
cout << "Введіть рядок: ";
getline(cin, text);  // читаємо повний рядок з пробілами

cout << "Довжина рядка: " << text.size() << endl;
```

```
if (!text.empty()) {
   cout << "Перший символ: " << text[0] << endl;
   cout << "Останній символ: " << text.back() << endl;
}

string added = text + " (додано)"; // конкатенація через +
cout << "Новий рядок: " << added << endl;

string sub = text.substr(0, 5);
cout << "Перші 5 символів: " << sub << endl;
```

У цьому прикладі спочатку зчитується рядок, потім виводиться його довжина. Якщо він не порожній, програма показує перший і останній символ. Далі створюється новий рядок added, який є конкатенацією введеного тексту та фрази " (додано)". Також демонструється отримання підрядка з перших 5 символів початкового тексту.

8.3 Перетворення між std::string і C-рядками

Іноді потрібно отримати C-рядок з std::string (наприклад, для сумісності з старими функціями на зразок printf). Для цього використовується метод .c_str(), який повертає const char* – вказівник на C-рядок, еквівалентний вмісту std::string.

```
string s = "Example";
const char* cstr = s.c_str();
// Тепер cstr можна передати в функцію, що очікує const char*
```

У зворотний бік (із const char* до std::string) перетворення відбувається автоматично при присвоєнні або ініціалізації:

```
const char* oldStyle = "Hello";
string newStyle = oldStyle; // newStyle == "Hello"
```

Зауваження: Для початкового рівня досить впевнено почуватися з std::string та базовими операціями. Глибше занурення передбачає розуміння кодування символів (наприклад, Unicode), використання ширших типів символів (wchar_t, char16_t, char32_t), але ці теми виходять за рамки базового курсу.

9 РОБОТА З ТЕКСТОВИМИ ФАЙЛАМИ

Вступ до поняття: Збереження даних у файл дозволяє програмі зберігати інформацію між запусками, працювати з великими обсягами даних та обмінюватися даними з іншими програмами. Текстові файли – це файли, вміст яких представлений у вигляді людськочитного тексту (послідовності символів). У C++ для роботи з текстовими файлами використовуються бібліотечні класи fstream (file stream).

Підключення бібліотеки <fstream>

Для роботи з файлами необхідно підключити заголовок:

```
#include <fstream>
```

Цей заголовок містить класи:

std::ifstream (input file stream) – для читання з файлу.

std::ofstream (output file stream) – для запису в файл.

std::fstream – універсальний клас для файлу, відкритого і на читання, і на запис (в залежності від режиму).

9.1 Відкриття файлу

Способи відкрити файл:

Через конструктор:

```
std::ifstream inputFile("data.txt");    // відкриття файлу data.txt для читання
std::ofstream outputFile("result.txt"); // відкриття файлу result.txt для запису (створиться новий, якщо не існує)
```

При такому відкритті одразу можна перевірити, чи файл успішно відкрився, звернувшись до методу is_open():

```
if (!inputFile.is_open()) {
   cerr << "Не вдалося відкрити файл data.txt\n";
}
```

Через метод .open():

```
std::ifstream inputFile;
inputFile.open("data.txt");
```

Це корисно, якщо оголошення і відкриття треба розділити (наприклад, ім'я файлу відоме тільки під час виконання).

Режими відкриття: За замовчуванням:

ifstream відкриває файл у режимі читання (помилка, якщо файл не існує).

ofstream відкриває файл у режимі запису (створить файл, якщо не існує, і очистить вміст, якщо існує).

Існують інші режими (доповнення app – дописування в кінець, ate – перехід в кінець після відкриття, binary – двійковий режим тощо), але для текстових файлів базово використовують стандартні режими.

9.2 Читання з файлу

Читання посимвольно або по словах: Використовується так само, як cin. Оператор >> зчитує з файлу, пропускаючи пробільні символи:

```
int x;
inputFile >> x; // зчитати число з файлу в змінну x
```

Читання рядків: Для зчитування цілих строк, включаючи пробіли, застосовують std::getline:

```
string line;
getline(inputFile, line);
```

Ця функція зчитує символи з файлу до символу нового рядка '\n' (або до кінця файлу) і зберігає їх у змінну line. Перевірка кінця файлу: Після читання можна перевіряти стан потоку. Класичний спосіб читання до кінця:

```
string line;
while (getline(inputFile, line)) {
```

```
    // обробка прочитаної стрічки
}
```

Цей цикл буде працювати, поки вдається зчитати рядок. Коли файл закінчиться або стан потоку стане некоректним (наприклад, помилка читання), вираз в while стане хибним, і цикл завершиться.

9.3 Запис у файл

Виведення до файлу виконується подібно до cout, але з використанням ofstream потоку:

```
outputFile << "Текст, який запишеться у файл\n";
int a = 5, b = 7;
outputFile << "Сума " << a << " + " << b << " = " << (a+b) << endl;
```

Як і у випадку з cout, маніпулятори на кшталт endl працюють і для файлових потоків (вставляють символ кінця рядка і очищують буфер).

9.4 Закриття файлу

Коли завершено роботу з файлом, його бажано закрити, викликавши метод .close():

```
inputFile.close();
outputFile.close();
```

Хоча при завершенні програми або при повторному відкритті файлу старий файл закриється автоматично (через звільнення ресурсів або виклик деструктора об’єкта ofstream/ifstream), явне закриття допомагає звільнити системні ресурси одразу і записати буферизовані дані на диск.

Приклад: Читання і запис файлу

```
#include <fstream>
#include <iostream>
```

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

int main() {
  ifstream inFile("input.txt");
  ofstream outFile("output.txt");

  if (!inFile) {
    cerr << "Помилка відкриття input.txt" << endl;
    return 1; // завершити програму з кодом помилки
  }

  string line;
  // Копіюємо вміст файлу input.txt до output.txt з нумерацією рядків
  int lineNum = 1;
  while (getline(inFile, line)) {
    outFile << lineNum << ": " << line << endl;
    lineNum++;
  }

  // Закриваємо файли
  inFile.close();
  outFile.close();

  cout << "Копіювання завершено." << endl;
  return 0;
}
```

Ця програма відкриває файл input.txt для читання і output.txt для запису. В циклі while кожен рядок з input.txt читається і записується в output.txt з додаванням номера рядка на початку. Якщо input.txt не вдалося відкрити, виводиться повідомлення про помилку та програма завершується.

Додаткові зауваження щодо роботи з файлами:

– Перевірка помилок: Крім is_open(), файлові потоки мають методи fail(), eof(), bad() для перевірки різних станів. Наприклад, inFile.eof() повертає true, якщо досягнуто кінець файлу.

– Режим додавання: Якщо потрібно не перезаписувати файл, а дописувати в кінець, можна відкривати ofstream так:

```
ofstream outFile("log.txt", ios::app);
```

– Прапорець ios::app (append) забезпечить дописування нових даних у кінець файлу, зберігаючи існуючий вміст.

– Шлях до файлу: Якщо файл не у тій самій текі, що й програма, потрібно вказати шлях (напр., "C:\\data\\input.txt" для Windows або "/home/user/data/input.txt" для Linux). В рядкових літералах обов'язково екранувати зворотні слеші подвійним слешем \\.

– Кодування файлу: Стандартний ofstream/ifstream працює з байтовими даними. Для Unicode (UTF-8) тексту зазвичай достатньо, щоб файл зберігався у UTF-8 і читався/писався без перетворень. Якщо потрібна робота з широкими символами, існує std::wifstream/std::wofstream для wchar_t та спеціальні налаштування локалі.

– Закриття файлів при виході: Якщо не викликати close(), файли закриються автоматично при виході з програми або при знищенні об'єкта ifstream/ofstream (наприклад, коли виконання виходить за межі блока, де об'єкт був оголошений). Але явне закриття – хороша практика, особливо у великих програмах.

ЛІТЕРАТУРА

1. Бублик В.В. Об’єктно-орієнтоване програмування: підручник для студентів, які навчаються за напрямами «Комп’ютерні науки», «Комп’ютерна інженерія», «Програмна інженерія», «Системна інженерія», «Інформатика», «Прикладна математика» – Київ : ІТ-книга, 2015. – 624 с.

2. Mayers S. Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14 1st Edition. – O'Reilly, 2014. – 334 p.

3. Lafore R. Object-Oriented Programming in C++. – Indianapolis : Sams Publishing, 2002. – 1012 p.

4. Уроки програмування на C++. – Режим доступу: https://acode.com.ua/uroki-po-cpp/.

5. Дорожня карта C++. - Режим доступу: https://roadmap.sh/cpp.

6. Мартін Р. Чистий код. Створення і рефакторинг за допомогою Agile. – К: Фабула, 2019. – 448 с.

7. Васильєв О.М. Програмування С++ в прикладах і задачах – К.: Ліра-К, 2019. – 382 с.

8. Алгоритми та структури даних — від «десь чув» до «ефективно застосовую». – Режим доступу: https://dou.ua/forums/topic/40645/.