КОНСПЕКТ ЛЕКЦІЙ З ДИСЦИПЛІНИ

«Комп'ютерні технології та програмування»

для здобувачів вищої освіти освітнього ступеня «бакалавр»
спеціальності 141 «Електроенергетика, електротехніка та електромеханіка»
освітньо-професійна програма «Комп'ютеризоване управління енергетичними системами»
факультет комп'ютерно-інтегрованих технологій, мехатроніки та робототехніки
кафедра робототехніки, електроенергетики та автоматизації ім. проф. Б.Б. Самотокіна

Схвалено на засіданні кафедри робототехніки, електроенергетики та автоматизації ім. проф. Б.Б. Самотокіна
27 серпня 2024 р.,
протокол № 6

Розробник: к.т.н., доцент кафедри робототехніки, електроенергетики та автоматизації ім. проф. Б.Б.Самотокіна ДОБРЖАНСЬКИЙ Олександр

Житомир
2024– 2025 н.р.

МОДУЛЬ 1. Програмування мовою С++

Змістовий модуль 1. Базис управляючих інструкцій С++.

Тема 1.1. Спеціальні символи, директиви, типи даних, змінні та елементарні операції над ними у мові програмування С++.

▒ Алфавіт та спеціальні символи мови С++

Алфавіт С++ включає базові латинські символи, цифри та широкий набір спеціальних знаків. Вони забезпечують:

- опис структури програми,
- виконання арифметичних і логічних операцій,
- роботу з літералами та директивами.

Алфавіт С++ складається з:

- Латинських літер: A–Z, a–z
- Цифр: 0–9
- Спеціальних символів: набір знаків, що використовуються для операцій, розділення та керування структурою програми.
- Пробільних символів: пробіл, табуляція, переведення рядка.

- Спеціальні символи для керування виводом: \n, \t, \a

- Арифметичні оператори

+ додавання
- віднімання
* множення
/ ділення
% остача від ділення

- Оператори порівняння

== рівність
!= нерівність
< менше
> більше

<= менше або дорівнює
>= більше або дорівнює

- Логічні оператори
&& логічне «і»
|| логічне «або»
! заперечення

- Символи для структури програми
{ } фігурні дужки – блоки коду
( ) круглі дужки – виклик функцій, вирази
[ ] квадратні дужки – масиви, індексація
; крапка з комою – завершення інструкції
, кома – розділення елементів

- Оператори присвоєння
= просте присвоєння
+=, -=, *=, /=, %= комбіновані операції

- Інші важливі символи
директиви препроцесора (наприклад, #include)
" подвійні лапки – рядкові літерали
' одинарні лапки – символьні літерали
\\ зворотна коса риска – керуючі послідовності (\n, \t)
:: оператор області видимості

▒ Директиви препроцесора (#include, #define)

Препроцесор — це спеціальна програма, яка виконується перед компіляцією коду. Він обробляє директиви, що починаються зі знака #.

- Директиви препроцесора не є частиною мови C++, але вони керують тим, як компілятор бачить код.
- Найчастіше використовуються #include для підключення бібліотек та #define для макросів.
- Вони забезпечують гнучкість і повторне використання коду.

Основні директиви:

#include

Використовується для підключення зовнішніх файлів (заголовків).
- #include <iostream> — підключення стандартної бібліотеки вводу/виводу.
- #include "myheader.h" — підключення власного файлу заголовків.

#define

Використовується для створення макросів (іменованих констант або простих замін).
- #define PI 3.14159 — замінює всі входження PI на число 3.14159.
- #define SQUARE(x) ((x)*(x)) — макрос-функція для обчислення квадрата.

Приклад:

```
#include <iostream>
#define PI 3.14159

int main() {
    double r = 5;
    double area = PI * r * r;
    std::cout << "Площа кола: " << area << std::endl;
    return 0;
}
```

▒ Типи даних: цілі, дійсні, символьні, логічні

Цілі типи (integer types)

- Використовуються для збереження цілих чисел (без дробової частини).

Варіанти цілих типів:
- int — стандартний цілий тип.
- short, long, long long — різні діапазони значень.
- unsigned — лише додатні числа (розширює верхню межу).

Приклад:

```
int a = 42;
unsigned int b = 100;
```

Дійсні типи (floating-point types)

- Використовуються для чисел з дробовою частиною.

Варіанти дійсних типів:
- float — одинарна точність.
- double — подвійна точність (частіше використовується).
- long double — розширена точність.

Приклад:

```
double pi = 3.14159;
float x = 2.5f;
```

Символьний тип (character type)
- Використовується для збереження одного символу (літери, цифри, знака).
Основний тип: char - може зберігати значення як ASCII-код символу.

Приклад:

```
char letter = 'A';
char digit = '7';
```

▒ Оголошення та ініціалізація змінних

Оголошення — це створення змінної з певним типом даних.

Синтаксис оголошення:

```
тип_даних ім’я_змінної;
```

Приклад:

```
int age;       // оголошення змінної age типу int
double salary; // оголошення змінної salary типу double
```

Ініціалізація — це присвоєння початкового значення змінній. Її можна виконати одразу при оголошенні.

Приклад:

```
int age = 25;          // ініціалізація значенням 25
double pi = 3.14159;   // ініціалізація значенням 3.14159
char letter = 'A';     // ініціалізація символом 'A'
bool flag = true;      // ініціалізація логічним значенням true
```

Способи ініціалізації:

- Класичний (через =)

```
int x = 10;
```

- Конструкторний (через дужки)

```
int y(20);
```

- Спискова (uniform initialization, через фігурні дужки)

```
int z{30};
double d{2.5};
```

Правила використання ініціалізації:

- Змінна повинна бути оголошена перед використанням.
- Ім’я змінної має бути унікальним у межах області видимості.
- Рекомендується одразу виконувати ініціалізацію, щоб уникнути помилок із «сміттєвими» значеннями.

▒ Арифметичні, логічні та порівняльні операції

Арифметичні операції

Використовуються для виконання математичних дій над числами.

1) додавання + виконує операцію a + b
2) віднімання - виконує операцію a - b
3) множення * виконує операцію a * b
4) ділення / виконує операцію a / b
5) остача від ділення % виконує операцію знаходження остачі від ділення

(виконується лише для цілих чисел) a % b

Логічні операції

Використовуються для роботи з умовами та булевими значеннями (true/false).

1) Логічне І (AND) &&
   Приклад:
   (x > 0 && y > 0)
2) Логічне АБО (OR)
   Приклад:
   (x > 0 || y > 0)
3) Заперечення !
   Приклад:
   !(x > 0)

Порівняльні операції

Використовуються для порівняння значень. Результат — булеве значення (true або false).

1) Рівність ==
   Приклад:
   a == b
2) Нерівність !=
   Приклад:
   a != b
3) Менше <
   Приклад:
   a < b
4) Більше >
   Приклад:
   a > b
5) Менше або дорівнює <=
   Прикад:
   a <= b
6) Більше або дорівнює >=
   Приклад:
   a >= b

Приклад:

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

int main() {
        int a = 10, b = 3;
        cout << "Сума: " << a + b << endl;          // арифметика
        cout << "Чи a > b? " << (a > b) << endl;    // порівняння
        cout << "Логіка: " << (a > 0 && b > 0) << endl; // логіка
        return 0;
}
```

▒ Пріоритет та асоціативність операторів

Структурована таблиця пріоритетів та асоціативності операторів у C++

Пріоритет	Оператори	Асоціативність
1 (найвищий)	`::` (scope resolution)	зліва направо
2	`++` (префікс), `--` (префікс), `()` (виклик функції), `[]` (індексація), `.` , `->`	зліва направо
3	`++` (постфікс), `--` (постфікс)	зліва направо
4	`!`, `~`, `+` (унарний), `-` (унарний), `*` (розіменування), `&` (адреса), `sizeof`, `new`, `delete`, `typeid`, `throw`	справа наліво
5	`.*`, `->*` (pointer-to-member)	зліва направо
6	`*`, `/`, `%`	зліва направо
7	`+`, `-`	зліва направо
8	`<<`, `>>`	зліва направо
9	`<`, `<=`, `>`, `>=`	зліва направо
10	`==`, `!=`	зліва направо
11	`&` (бітове AND)	зліва направо
12	`^` (бітове XOR)	зліва направо
13	`|` (бітове OR)	зліва направо
14	`&&` (логічне AND)	зліва направо
15	`||` (логічне OR)	зліва направо
16	`?:` (тернарний оператор)	справа наліво
17	`=`, `+=`, `-=`, `*=`, `/=`, `%=` , `<<=`, `>>=`, `&=`, `^=`, `|=`	справа наліво
18	`throw` (виключення)	справа наліво
19	`,` (кома)	зліва направо

- Пріоритет визначає порядок виконання операторів у виразі. Наприклад, у `a + b * c` спочатку виконується множення.
- Асоціативність визначає напрямок обчислення операторів однакового рівня. Наприклад, у `a = b = c` оператор присвоєння має праву асоціативність, тому спочатку виконується `b = c`, а потім `a = (результат)`.
Хочеш, я зроблю для тебе набір флеш-карток з прикладами виразів, щоб студенти могли швидко засвоїти порядок виконання операторів?

Тема 1.2. Потоки введення / виведення при операціях з консоллю у мові програмування С++.

▓ Стандартні бібліотеки <iostream>

Бібліотека <iostream> у C++ — це стандартний заголовковий файл, який забезпечує роботу з потоками введення та виведення. Вона є частиною стандартної бібліотеки і використовується для взаємодії програми з консоллю (клавіатура та екран).

Основні об’єкти <iostream>:
- cin — стандартний потік введення (з клавіатури).
- cout — стандартний потік виведення (на екран).
- cerr — потік виведення для повідомлень про помилки (не буферизується).
- clog — потік виведення для логування (буферизується).
- wcin, wcout, wcerr, wclog — ті ж самі потоки, але для широких символів (wchar_t).

Приклад:

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

int main() {
        int x;
        cout << "Введіть число: ";
        cin >> x;
        cout << "Ви ввели: " << x << endl;
```

```
        cerr << "Це повідомлення про помилку!" << endl;
        return 0;
    }
```

- <iostream> часто автоматично підключає інші заголовки, такі як <istream>, <ostream>, <ios>, <streambuf>.

- Потоки підтримують оператори зсуву (<< для виведення, >> для введення), що робить синтаксис простим і зрозумілим.

▓ Об’єкти cin, cout

Об’єкти cin та cout у C++ — це стандартні потоки введення та виведення, оголошені в бібліотеці <iostream>.

cin
- Є об’єктом класу istream.
- Використовується для введення даних з клавіатури.
- Працює з оператором >> (оператор вилучення).

Приклад:

```
int x;
cin >> x;   // користувач вводить число
```

cout
- Є об’єктом класу ostream.
- Використовується для виведення даних на екран.
- Працює з оператором << (оператор вставки).

Приклад:

```
cout << "Введене число: " << x << endl;
```

- Обидва є глобальними об’єктами, створеними ще до виконання main().
- Їх можна перевантажувати для роботи з користувацькими класами (через перевантаження операторів >> та <<) .
- Вони забезпечують простий і зрозумілий синтаксис для роботи з потоками.

▓ Форматування виведення (.width(), .precision())

C++ форматування виведення здійснюється через методи класу ios та маніпулятори. Найчастіше використовуються .width() та .precision() для керування виглядом числових даних.

.width()

- Встановлює мінімальну ширину поля для наступного виведення.
- Якщо значення займає менше місця, додаються пробіли (за замовчуванням зліва).
- Використовується лише для одного наступного виведення, після чого повертається до стандартного режиму.

Приклад:

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

int main() {
    cout.width(10);
    cout << 123 << endl;   // "       123" (7 пробілів + 123)
}
```

.precision()

- Визначає кількість значущих цифр для чисел з плаваючою комою.
- У поєднанні з fixed або scientific змінює значення на кількість цифр після коми.

Приклад:

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

int main() {
    double pi = 3.1415926535;

    cout.precision(4);
    cout << pi << endl;          // 3.142 (4 значущі цифри)
```

```
    cout << fixed << setprecision(4);
    cout << pi << endl;          // 3.1416 (4 цифри після коми)
}
```

▓ Читання рядків та числових даних

Читання числових даних

Використовується оператор >>:

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

int main() {
    int a;
    double b;
    cout << "Введіть ціле число: ";
    cin >> a;
    cout << "Введіть дійсне число: ";
    cin >> b;
    cout << "a = " << a << ", b = " << b << endl;
}
```

- cin >> a; зчитує ціле число.
- cin >> b; зчитує число з плаваючою комою.
- Введення завершується пробілом, табуляцією або Enter.

Читання рядків (без пробілів)

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

int main() {
    string name;
    cout << "Введіть ім’я: ";
    cin >> name;   // зчитує лише до першого пробілу
    cout << "Привіт, " << name << "!" << endl;
```

```
    }
```

- cin >> name; зчитує слово до пробілу.
- Якщо потрібно зчитати кілька слів — використовують getline().

Читання рядків з пробілами:

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

int main() {
   string fullName;
   cout << "Введіть повне ім’я: ";
   getline(cin, fullName);   // зчитує весь рядок до Enter
   cout << "Ви ввели: " << fullName << endl;
  }
```

- getline(cin, fullName); зчитує увесь рядок, включно з пробілами.

Комбіноване введення

Якщо після числового вводу потрібно зчитати рядок через getline(), слід очистити буфер:

```
int age;
string name;
cin >> age;
cin.ignore();          // ігнорує символ '\n'
getline(cin, name);
```

▓ Обробка помилок введення

У C++ обробка помилок введення здійснюється через стани потоку cin та спеціальні методи класу istream. Це дозволяє програмі реагувати на некоректні дані та уникати збоїв.

Основні стани потоку

- cin.good() — введення успішне, помилок немає.
- cin.fail() — помилка формату (наприклад, введено літери замість числа).
- cin.eof() — досягнуто кінець файлу/потоку.
- cin.bad() — серйозна помилка (наприклад, апаратний збій).

Методи для очищення та повторного введення

- cin.clear() — скидає прапори помилок (наприклад, після fail()).
- cin.ignore(n, '\n') — пропускає n символів або до символу нового рядка.
- cin.sync() — очищає буфер введення.

Приклад: перевірка введення числа

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

int main() {
   int x;
   cout << "Введіть число: ";
   cin >> x;

   if (cin.fail()) {
      cout << "Помилка! Ви ввели не число." << endl;
      cin.clear();           // скидаємо стан помилки
      cin.ignore(1000, '\n'); // очищаємо буфер
   } else {
      cout << "Ви ввели: " << x << endl;
   }
 }
```

Приклад: повторне введення до успіху

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

int main() {
   int x;
```

```
    while (true) {
        cout << "Введіть число: ";
        cin >> x;

        if (cin.fail()) {
            cout << "Помилка! Спробуйте ще раз." << endl;
            cin.clear();
            cin.ignore(1000, '\n');
        } else {
            break; // правильне введення
        }
    }
    cout << "Ви ввели: " << x << endl;
}
```

Тема 1.3. Операції розгалуження та циклічні операції при програмуванні мовою С++.

▒ Умовні оператори: if, if-else, switch

В C++ умовні оператори дозволяють керувати виконанням програми залежно від певних умов. Основні конструкції — if, if-else та switch.

if

Виконує блок коду, якщо умова істинна (true).

```
int x = 5;
if (x > 0) {
    cout << "x додатне" << endl;
}
```

if-else

Додає альтернативний блок, який виконується, якщо умова хибна (false).

```
int x = -3;
if (x > 0) {
    cout << "x додатне" << endl;
} else {
```

```
    cout << "x недодатне" << endl;
  }
```

if-else if-else

Дозволяє перевіряти кілька умов послідовно.

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

switch

Зручний для вибору між багатьма варіантами одного виразу (зазвичай цілого типу або enum).

```
int day = 3;
switch (day) {
   case 1: cout << "Понеділок"; break;
   case 2: cout << "Вівторок"; break;
   case 3: cout << "Середа"; break;
   case 4: cout << "Четвер"; break;
   case 5: cout << "П’ятниця"; break;
   case 6: cout << "Субота"; break;
   case 7: cout << "Неділя"; break;
   default: cout << "Невірний номер дня";
  }
```

▒ Тернарний оператор ?:

Тернарний оператор ?: у C++ — це компактна форма умовного вибору, яка дозволяє записати простий if-else в одному рядку.

```
умова ? вираз_якщо_true : вираз_якщо_false;
```

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

int main() {
   int x = 5;
   string result = (x > 0) ? "Додатне" : "Недодатне";
   cout << "x = " << x << " → " << result << endl;
  }
```

- Якщо x > 0 → вибирається "Додатне".
- Інакше → "Недодатне".

Особливості:
- Повертає значення, яке можна присвоїти змінній.
- Зручний для коротких умовних виразів.
- Може вкладатися, але це знижує читабельність:

```
string res = (x > 0) ? "Додатне" : (x < 0 ? "Від'ємне" : "Нуль");
```

▒ Цикли: for, while, do-while

У C++ цикли дозволяють багаторазово виконувати блок коду, поки умова істинна. Основні конструкції — for, while, do-while.

Цикл for

Використовується, коли кількість повторень відома заздалегідь.

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

int main() {
   for (int i = 0; i < 5; i++) {
      cout << "Ітерація №" << i << endl;
   }
  }
```

- Має три частини: ініціалізація, умова, оновлення.
- Виконується, поки умова істинна.

Цикл while

Виконує блок коду, поки умова істинна.

```
int i = 0;
while (i < 5) {
   cout << "Ітерація №" << i << endl;
   i++;
 }
```

- Перевірка умови відбувається перед виконанням тіла циклу.
- Якщо умова відразу хибна, тіло не виконається жодного разу.

Цикл do-while

Виконує тіло циклу хоча б один раз, навіть якщо умова хибна.

```
int i = 0;
do {
   cout << "Ітерація №" << i << endl;
   i++;
 } while (i < 5);
```

- Перевірка умови відбувається після виконання тіла.
- Гарантовано мінімум одне виконання.

▓ Оператори керування циклом (break, continue)

break

- Призначення: достроково завершує виконання циклу.
- Після break програма виходить з циклу і продовжує виконання наступних інструкцій.

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

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i == 5) break;   // цикл зупиниться на i = 5
      cout << i << " ";
   }
     // Виведе: 1 2 3 4
 }
```

continue

- Призначення: пропускає поточну ітерацію та переходить до наступної.
- Використовується, коли потрібно пропустити певні значення.

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

int main() {
   for (int i = 1; i <= 10; i++) {
      if (i % 2 == 0) continue;   // пропускає парні числа
      cout << i << " ";
   }
   // Виведе: 1 3 5 7 9
 }
```

▓ Вкладені цикли та комбіновані конструкції

Вкладені цикли

Цикл всередині іншого циклу виконується повністю для кожної ітерації зовнішнього циклу.

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

int main() {
   for (int i = 1; i <= 3; i++) {          // зовнішній цикл
      for (int j = 1; j <= 4; j++) {       // внутрішній цикл
         cout << "(" << i << "," << j << ") ";
      }
      cout << endl;
   }
}
```

Результат:

```
(1,1) (1,2) (1,3) (1,4)
(2,1) (2,2) (2,3) (2,4)
(3,1) (3,2) (3,3) (3,4)
```

Комбіновані конструкції

Цикли можна поєднувати з умовними операторами (if, switch) для більш гнучкого керування.

Приклад: таблиця множення

```
for (int i = 1; i <= 5; i++) {
    for (int j = 1; j <= 5; j++) {
        cout << i * j << "\t";
    }
    cout << endl;
  }
```

Приклад: пропуск певних значень

```
for (int i = 1; i <= 10; i++) {
    if (i % 2 == 0) continue;   // пропускаємо парні числа
    cout << i << " ";
}
```

Приклад: вихід із вкладеного циклу

```
for (int i = 1; i <= 3; i++) {
    for (int j = 1; j <= 3; j++) {
        if (i == j) break;   // вихід із внутрішнього циклу
        cout << i << "," << j << endl;
    }
}
```

Змістовий модуль 2. Робота з масивами та файлами даних в програмах на мові С++.

Тема 2.1. Масиви даних та рядки, операції над ними у мові програмування С++.

▒ Одновимірні та багатовимірні масиви

У С++ масиви — це структури даних, які дозволяють зберігати кілька

елементів одного типу під одним ім’ям. Вони бувають одновимірні та багатовимірні.

Одновимірні масиви

- Це набір елементів, розташованих послідовно в пам’яті.
- Доступ до елементів здійснюється за індексом (починається з 0).

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

int main() {
   int arr[5] = {10, 20, 30, 40, 50};

   for (int i = 0; i < 5; i++) {
      cout << "arr[" << i << "] = " << arr[i] << endl;
   }
 }
```

Результат:

```
arr[0] = 10
arr[1] = 20
arr[2] = 30
arr[3] = 40
arr[4] = 50
```

▓ Ініціалізація та доступ до елементів

Багатовимірні масиви

- Це масиви масивів (наприклад, матриці).
- Найчастіше використовуються двовимірні масиви.

```
#include <iostream>
using namespace std;
int main() {
   int matrix[3][3] = {
      {1, 2, 3},
      {4, 5, 6},
      {7, 8, 9}
   };

   for (int i = 0; i < 3; i++) {
```

```
            for (int j = 0; j < 3; j++) {
                cout << matrix[i][j] << " ";
            }
            cout << endl;
        }
    }
```

Результат:

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

▓ Рядки як масиви символів
▓ Стандартна бібліотека <string>
▓ Основні операції над рядками

У C++ рядки можна розглядати як масиви символів, оскільки вони фактично є послідовністю елементів типу char, розташованих у пам’яті один за одним.

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

int main() {
    char str[6] = {'H', 'e', 'l', 'l', 'o', '\0'}; // '\0' – завершальний символ
    cout << str << endl;   // Виведе: Hello
}
```

- Завжди завершується нуль-символом '\0', який позначає кінець рядка.
- Без '\0' виведення може бути некоректним.

Ініціалізація рядка

```
char str1[] = "Hello";   // автоматично додається '\0'
char str2[10] = "Hi";    // решта елементів заповнені '\0'
```

Введення та виведення рядків

```
char name[50];
cout << "Введіть ім’я: ";
cin >> name;   // читає до пробілу
cout << "Привіт, " << name << "!" << endl;
```

- cin >> name; зчитує лише одне слово.

Для рядків із пробілами використовують cin.getline():

```
cin.getline(name, 50);
```

Робота з рядками як масивами

```
char word[] = "Test";
for (int i = 0; word[i] != '\0'; i++) {
   cout << "Символ " << i << ": " << word[i] << endl;
 }
```

Функції для роботи з рядками (з <cstring>)

- strlen(str) — довжина рядка.
- strcpy(dest, src) — копіювання рядка.
- strcat(str1, str2) — конкатенація рядків.
- strcmp(str1, str2) — порівняння рядків.

У C++ string — це стандартний клас для роботи з рядками, оголошений у бібліотеці <string>. Він є спеціалізацією шаблону std::basic_string<char>, тобто фактично контейнером для символів типу char .

Основні характеристики std::string

- Не є вбудованим типом, але поводиться як базовий завдяки зручному інтерфейсу .
- Автоматично керує пам’яттю (на відміну від масивів char[]).
- Підтримує операції конкатенації, порівняння, пошуку, обрізання тощо.
- Має методи для роботи з рядками: .length(), .substr(), .find(), .replace(), .c_str() та інші .

```
#include <iostream>
#include <string>
using namespace std;
int main() {
   string greeting = "Hello";
   string name;
   cout << "Введіть ім’я: ";
   cin >> name;
   cout << greeting + ", " + name + "!" << endl;
 }
```

Тема 2.2. Застосування покажчиків у мові програмування С++. Динамічне виділення пам’яті.

▓ Оголошення та використання покажчиків

У С++ покажчики (pointers) — це змінні, які зберігають адресу іншої змінної в пам’яті. Вони є фундаментальним інструментом для роботи з пам’яттю, масивами, динамічними структурами даних та функціями.

Оголошення покажчика

```
int x = 10;
int* p;      // оголошення покажчика на int
 p = &x;     // p зберігає адресу змінної x
```

- int* p; — покажчик на ціле число.
- &x — оператор "адреса", повертає адресу змінної x.

Використання покажчика

```
cout << "Значення x: " << x << endl;
cout << "Адреса x: " << &x << endl;
cout << "Адреса в p: " << p << endl;
cout << "Значення через p: " << *p << endl;
```

- p — містить адресу змінної x.
- *p — оператор розіменування, дозволяє отримати значення за цією адресою.

Зміна значення через покажчик

```
*p = 20;   // змінюємо значення x через покажчик
 cout << "x = " << x << endl;   // x тепер дорівнює 20
```

▓ Арифметика покажчиків

Покажчики та масиви

Масиви автоматично перетворюються на покажчики:

```
int arr[3] = {1, 2, 3};
int* p = arr;   // p вказує на перший елемент масиву
```

```
cout << *p << endl;     // 1
cout << *(p + 1) << endl; // 2
 cout << *(p + 2) << endl; // 3
```

▓ Динамічне виділення пам’яті (new, delete)

```
int* p = new int;   // виділення пам’яті
*p = 42;
cout << *p << endl; // 42
 delete p;          // звільнення пам’яті
```

▓ Масиви та структури у динамічній пам’яті

У C++ масиви та структури можна розміщувати у динамічній пам’яті за допомогою операторів та . Це дозволяє створювати гнучкі структури даних, розмір яких визначається під час виконання програми.

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

int main() {
    int n;
    cout << "Введіть розмір масиву: ";
    cin >> n;

    int* arr = new int[n];   // виділення пам’яті для масиву

    for (int i = 0; i < n; i++) {
        arr[i] = i * 2;
    }

    for (int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }

    delete[] arr;   // звільнення пам’яті
}
```

- new int[n] — створює масив у динамічній пам’яті.
- delete[] arr — обов’язково звільняє пам’ять.

Динамічні структури

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

struct Student {
    string name;
    int age;
};

int main() {
    Student* st = new Student;   // виділення пам’яті для структури
    st->name = "Olexander";
    st->age = 20;

    cout << st->name << " (" << st->age << " років)" << endl;

    delete st;   // звільнення пам’яті
  }
```

- new Student — створює об’єкт структури у динамічній пам’яті.
- Доступ до полів здійснюється через оператор ->.

Масив структур у динамічній пам’яті

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

struct Student {
    string name;
    int age;
};

int main() {
    int n;
```

```
    cout << "Кількість студентів: ";
    cin >> n;

    Student* group = new Student[n];   // масив структур

    for (int i = 0; i < n; i++) {
       cout << "Ім’я студента " << i+1 << ": ";
       cin >> group[i].name;
       cout << "Вік: ";
       cin >> group[i].age;
    }

    cout << "\nСписок студентів:\n";
    for (int i = 0; i < n; i++) {
       cout << group[i].name << " (" << group[i].age << " років)" << endl;
    }

    delete[] group;   // звільнення пам’яті
}
```

▓ Потенційні помилки та витоки пам’яті

У C++ при роботі з динамічною пам’яттю (через new, delete, malloc, free) можуть виникати потенційні помилки та витоки пам’яті. Це критично для стабільності програм, особливо великих систем.

- Використання невиділеної пам’яті

```
int* p;
*p = 10;   // помилка: p не ініціалізований
```

- Подвійне звільнення пам’яті

```
int* p = new int(5);
delete p;
delete p;   // помилка: повторне звільнення
```

- Використання після звільнення (dangling pointer)

```
int* p = new int(5);
```

```
delete p;
cout << *p;   // помилка: доступ до вже звільненої пам’яті
```

- Вихід за межі масиву

```
int* arr = new int[5];
arr[5] = 10;   // помилка: індекс поза межами
delete[] arr;
```

- Витоки пам’яті (memory leaks)

```
void func() {
   int* p = new int(10);
   // немає delete → витік пам’яті
}
```

Як уникати помилок

- Завжди використовувати delete / delete[] для кожного new / new[].
- Після звільнення пам’яті встановлювати покажчик у nullptr:

```
delete p;
p = nullptr;
```

- Використовувати розумні покажчики (std::unique_ptr, std::shared_ptr) з бібліотеки <memory>. Вони автоматично керують життєвим циклом об’єктів.
- Перевіряти індекси масивів перед доступом.
- Використовувати інструменти для пошуку витоків пам’яті (Valgrind, AddressSanitizer).
- Найбільша небезпека при роботі з динамічною пам’яттю — це витоки та некоректний доступ. У сучасному C++ рекомендується використовувати розумні покажчики та контейнери STL (vector, string), щоб мінімізувати ризики.

Тема 2.3. Файлові потоки введення / виведення при створенні застосувань мовою програмування С++.

▓ Бібліотека <fstream>

▓ Бібліотека <fstream> у C++ використовується для роботи з файлами — читання та запису даних. Вона є частиною стандартної бібліотеки і надає класи для організації файлових потоків.

▓ Об’єкти ifstream, ofstream, fstream

Основні класи <fstream>

- ifstream — потік для читання з файлу.
- ofstream — потік для запису у файл.
- fstream — універсальний потік, який може працювати і для читання, і для запису.

▓ Запис та читання текстових файлів

Приклад: запис у файл

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

int main() {
   ofstream fout("data.txt");   // відкриття файлу для запису
   if (!fout) {
      cout << "Помилка відкриття файлу!" << endl;
      return 1;
   }

   fout << "Hello, file!" << endl;
   fout << 123 << endl;
   fout.close();   // закриття файлу
 }
```

Приклад: читання з файлу

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

int main() {
   ifstream fin("data.txt");   // відкриття файлу для читання
   if (!fin) {
      cout << "Файл не знайдено!" << endl;
      return 1;
```

```
    }

    string text;
    int number;
    fin >> text >> number;   // читає "Hello," у text і 123 у number
    cout << text << " " << number << endl;

    fin.close();
}
```

Приклад: комбіноване використання

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

int main() {
    fstream file("data.txt", ios::in | ios::out | ios::app);
    if (!file) {
        cout << "Помилка відкриття файлу!" << endl;
        return 1;
    }

    file << "Додатковий рядок" << endl;  // запис
    file.seekg(0);                       // повернення на початок
    string line;
    while (getline(file, line)) {        // читання рядків
        cout << line << endl;
    }

    file.close();
}
```

- Файли відкриваються з певними режимами доступу (ios::in, ios::out, ios::app, ios::binary).
- Завжди перевіряти, чи файл відкрився успішно (if (!fin) / if (!fout)).
- Не забувати закривати файл методом .close().

▓ Робота з бінарними файлами

У C++ робота з бінарними файлами здійснюється через бібліотеку <fstream> із використанням режиму ios::binary. Це дозволяє зберігати та читати дані у вигляді байтів, без перетворення у текстовий формат.

Запис у бінарний файл

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

int main() {
   int numbers[5] = {10, 20, 30, 40, 50};

   ofstream fout("data.bin", ios::binary);
   if (!fout) {
      cout << "Помилка відкриття файлу!" << endl;
      return 1;
   }

   fout.write(reinterpret_cast<char*>(numbers), sizeof(numbers));
   fout.close();
 }
```

- ios::binary — відкриття файлу у бінарному режимі.
- write() — запис масиву байтів у файл.
- reinterpret_cast<char*> — перетворення покажчика на int у покажчик на char.

Читання з бінарного файлу

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

int main() {
   int numbers[5];

   ifstream fin("data.bin", ios::binary);
   if (!fin) {
```

```
        cout << "Файл не знайдено!" << endl;
        return 1;
    }

    fin.read(reinterpret_cast<char*>(numbers), sizeof(numbers));
    fin.close();

    for (int i = 0; i < 5; i++) {
        cout << numbers[i] << " ";
    }
}
```

- read() — читає масив байтів із файлу.
- Дані відновлюються у тому ж форматі, в якому були записані.

Робота зі структурами

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

struct Student {
    char name[20];
    int age;
};

int main() {
    Student st1 = {"Olexander", 20};

    ofstream fout("student.bin", ios::binary);
    fout.write(reinterpret_cast<char*>(&st1), sizeof(st1));
    fout.close();

    Student st2;
    ifstream fin("student.bin", ios::binary);
    fin.read(reinterpret_cast<char*>(&st2), sizeof(st2));
    fin.close();

    cout << st2.name << " (" << st2.age << " років)" << endl;
}
```

- Структури можна зберігати та відновлювати цілком.
- Важливо, щоб структура не містила динамічних полів (std::string, покажчики), бо вони не збережуться коректно.

▓ Обробка помилок при роботі з файлами

У C++ при роботі з файлами через бібліотеку <fstream> важливо враховувати можливі помилки відкриття, читання та запису. Для цього використовуються методи перевірки стану потоку.

Перевірка відкриття файлу

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

int main() {
   ifstream fin("data.txt");
   if (!fin) {   // перевірка, чи файл відкрився
      cout << "Помилка: файл не знайдено!" << endl;
      return 1;
   }
   cout << "Файл відкрито успішно." << endl;
   fin.close();
 }
```

- if (!fin) — перевірка, чи файл відкрився коректно.
- Якщо файл не існує або немає доступу, повертається помилка.

Перевірка стану потоку
- .good() — немає помилок.
- .fail() — помилка формату або операції.
- .eof() — досягнуто кінець файлу.
- .bad() — серйозна помилка (наприклад, апаратний збій).

```
ifstream fin("data.txt");
int x;
fin >> x;
if (fin.fail()) {
```

```
        cout << "Помилка читання числа!" << endl;
    }
```

Обробка помилок при записі

```
    ofstream fout("output.txt");
    if (!fout) {
        cout << "Помилка відкриття файлу для запису!" << endl;
        return 1;
    }

    fout << "Hello, world!" << endl;
    if (fout.fail()) {
        cout << "Помилка запису у файл!" << endl;
    }
fout.close();
```

Використання винятків (try-catch)

Можна налаштувати потік так, щоб він кидав винятки при помилках:

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

    int main() {
        ifstream fin;
        fin.exceptions(ifstream::failbit | ifstream::badbit);

        try {
            fin.open("data.txt");
            int x;
            fin >> x;
            cout << "Прочитано: " << x << endl;
        }
        catch (ios_base::failure &e) {
            cout << "Виняток: " << e.what() << endl;
        }
    }
```

- .exceptions() — вмикає генерацію винятків при помилках.
- catch перехоплює виняток і дозволяє обробити його.

Змістовий модуль 3. Структуризація даних. Об'єктно-орієнтоване програмування на С++.

Тема 3.1. Застосування структур у мові програмування С++.

▓ Оголошення структур
▓ Ініціалізація та доступ до полів

Структура — це користувацький тип даних, який дозволяє об'єднувати змінні різних типів під одним іменем.
- Використовується для групування логічно пов'язаних даних (наприклад, дані про студента: ім'я, вік, середній бал).

У С++ структура оголошується за допомогою ключового слова struct.

```
struct Student {
   std::string name;   // ім'я студента
   int age;            // вік
   double grade;       // середній бал
 };
```

- Ім'я структури: Student — новий тип даних.
- Члени структури: name, age, grade — змінні різних типів.

Створення змінних структури

```
Student s1; // створення змінної типу Student
s1.name = "Олександр";
s1.age = 20;
 s1.grade = 89.5;
```

Ініціалізація при оголошенні

```
Student s2 = {"Марія", 19, 95.0};
```

Приклад:

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

struct Student {
```

```
    std::string name;
    int age;
    double grade;
};

int main() {
    Student s1 = {"Олександр", 20, 89.5};
    std::cout << "Ім’я: " << s1.name << "\n";
    std::cout << "Вік: " << s1.age << "\n";
    std::cout << "Середній бал: " << s1.grade << "\n";
    return 0;
 }
```

▒ Масиви структур

Оголошення масиву структур

```
struct Student {
    std::string name;
    int age;
    double grade;
};

 Student group[3]; // масив із 3 студентів
```

Заповнення масиву

```
group[0] = {"Олександр", 20, 89.5};
group[1] = {"Марія", 19, 95.0};
 group[2] = {"Іван", 21, 78.3};
```

Доступ до елементів масиву

```
std::cout << group[1].name << " має середній бал: " << group[1].grade << "\n";
```

Ініціалізація масиву при оголошенні

```
Student group[3] = {
   {"Олександр", 20, 89.5},
   {"Марія", 19, 95.0},
   {"Іван", 21, 78.3}
 };
```

Використання циклів

```
for (int i = 0; i < 3; i++) {
   std::cout << group[i].name << " (" << group[i].age << " років) "
          << " має бал " << group[i].grade << "\n";
    }
```

▓ Вкладені структури

- Це структури всередині структур, які дозволяють моделювати складні об’єкти.
- Використовуються для опису даних, що мають ієрархічну або багаторівневу структуру (наприклад, адреса всередині даних про студента).

```
struct Address {
   std::string city;
   std::string street;
   int houseNumber;
};

struct Student {
   std::string name;
   int age;
   double grade;
   Address address; // вкладена структура
 };

Student s1 = {
   "Олександр",
   20,
   89.5,
```

```
    {"Житомир", "Київська", 12}
  };

  std::cout << s1.name << " живе у місті " << s1.address.city << "\n";

Student group[2] = {
    {"Марія", 19, 95.0, {"Львів", "Галицька", 5}},
    {"Іван", 21, 78.3, {"Київ", "Хрещатик", 10}}
};

for (int i = 0; i < 2; i++) {
    std::cout << group[i].name << " проживає на вулиці "
              << group[i].address.street << "\n";
  }
```

Тема 3.2. Оголошення, написання та правила використання функцій у мові програмування С++.

▓ Оголошення та виклик функцій

Функція — це блок коду, який виконує певну задачу і може бути викликаний у програмі багаторазово.
- Використання функцій робить програму більш зрозумілою, модульною та легкою для супроводу.

Оголошення функції
Функція оголошується за схемою:

```
тип_повернення ім’я_функції(параметри) {
    // тіло функції
    return значення; // якщо є тип повернення
  }
```

Приклад:

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

Виклик функції

```
int result = add(5, 7); // виклик функції
 std::cout << "Результат: " << result << "\n";
```

Функції без параметрів

```
void greet() {
   std::cout << "Привіт, світе!\n";
}

 greet(); // виклик
```

Функції з параметрами за замовчуванням

```
void printMessage(std::string msg = "Hello") {
   std::cout << msg << "\n";
}

printMessage();          // виведе "Hello"
 printMessage("Привіт");  // виведе "Привіт"
```

Практичне застосування

- Модульність: розділення програми на логічні блоки.
- Повторне використання: один код — багато викликів.
- Зручність: легше тестувати та змінювати окремі частини програми.

▓ Передача параметрів (за значенням, за посиланням)

У C++ параметри можна передавати:

- За значенням — створюється копія аргументу.
- За посиланням — функція працює безпосередньо з оригінальною змінною.

Передача за посиланням дозволяє:

- уникнути зайвого копіювання (економія пам’яті та часу),
- змінювати значення змінних у функції.

```
    void updateValue(int &x) {
       x = x + 10; // змінюємо оригінальну змінну
     }
```

- int &x — параметр передається за посиланням.
- Будь-які зміни всередині функції впливають на змінну, передану при виклику.

Порівняння з передачею за значенням

```
    void byValue(int x) {
       x = x + 10; // змінюється лише копія
    }

    void byReference(int &x) {
       x = x + 10; // змінюється оригінал
    }

    int main() {
       int num = 5;
       byValue(num);
       std::cout << num << "\n"; // 5

       byReference(num);
       std::cout << num << "\n"; // 15
     }
```

Константні посилання

- Якщо потрібно передати параметр без копіювання, але заборонити зміну, використовують const &.

```
    void printValue(const std::string &text) {
       std::cout << text << "\n";
     }
```

Практичне застосування

- Оптимізація: уникнення копіювання великих об’єктів (наприклад, рядків, структур).

- Модифікація даних: функції можуть змінювати значення змінних.
- Безпечне читання: використання const & для великих об’єктів, які не потрібно змінювати.

▓ Локальні та глобальні змінні

Локальні змінні
- Оголошуються всередині функцій або блоків коду {}.
- Область видимості: доступні лише в межах тієї функції чи блоку, де оголошені.
- Життєвий цикл: створюються при вході у функцію/блок і знищуються при виході.

```
#include <iostream>
void func() {
    int localVar = 10; // локальна змінна
    std::cout << localVar << "\n";
}
```

Глобальні змінні
- Оголошуються поза будь-якими функціями (зазвичай на початку програми).
- Область видимості: доступні у всіх функціях програми.
- Життєвий цикл: існують протягом виконання всієї програми.

```
#include <iostream>
int globalVar = 100; // глобальна змінна

void func() {
    std::cout << globalVar << "\n"; // доступ до глобальної змінної
}
```

Конфлікт імен
Якщо локальна змінна має те саме ім’я, що й глобальна, то локальна змінна перекриває глобальну в межах функції.
Для доступу до глобальної змінної використовується оператор :: (scope resolution).

```
#include <iostream>
int value = 50; // глобальна змінна

void func() {
   int value = 10; // локальна змінна
   std::cout << value << "\n";   // виведе 10
   std::cout << ::value << "\n"; // доступ до глобальної змінної → 50
 }
```

Практичне застосування

- Локальні змінні: використовуються для тимчасових обчислень, щоб уникати конфліктів і зберігати чистоту коду.
- Глобальні змінні: застосовуються для даних, які мають бути доступні у багатьох функціях (наприклад, налаштування програми, константи).

▓ Рекурсивні функції

Рекурсія — це процес, коли функція викликає саму себе для розв’язання задачі.

- Використовується для задач, які можна розбити на менші підзадачі того ж типу.

Структура рекурсивної функції

Рекурсивна функція завжди має:

- Базовий випадок — умова завершення, щоб уникнути нескінченних викликів.
- Рекурсивний виклик — виклик самої себе з новими параметрами.

Приклад: факторіал числа

```
#include <iostream>

int factorial(int n) {
   if (n == 0 || n == 1) // базовий випадок
      return 1;
   else
      return n * factorial(n - 1); // рекурсивний виклик
}

int main() {
```

```
    std::cout << "Факторіал 5 = " << factorial(5);
    return 0;
}
```

Результат:

```
Факторіал 5 = 120
```

Приклад: числа Фібоначчі

```
int fibonacci(int n) {
    if (n == 0) return 0;       // базовий випадок
    if (n == 1) return 1;       // базовий випадок
    return fibonacci(n - 1) + fibonacci(n - 2); // рекурсія
  }
```

Переваги рекурсії

- Простота запису для задач із природною ієрархією (дерева, графи, математичні послідовності).
- Зручність для алгоритмів пошуку та сортування (наприклад, QuickSort, MergeSort).

Недоліки рекурсії

- Може бути менш ефективною через накладні витрати на виклики функцій.
- Ризик переповнення стеку при надто глибокій рекурсії.
- Іноді краще використовувати ітеративні алгоритми.

Практичне застосування

- Алгоритми роботи з деревами та графами (обхід вузлів).
- Математичні задачі (факторіал, Фібоначчі).
- Алгоритми сортування (QuickSort, MergeSort).

▓ Прототипи та перевантаження функцій

Функції можна оголошувати прототипом перед main(), а реалізовувати після:

```
int add(int a, int b); // прототип

int main() {
    std::cout << add(3, 4);
```

```
    return 0;
}

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

Перевантаження функцій означає, що в одній програмі можна оголосити кілька функцій з однаковим ім’ям, але з різними параметрами.
- Компилятор сам вибирає потрібну функцію залежно від кількості та типів аргументів при виклику.

```
#include <iostream>

// Функція для додавання двох цілих чисел
int add(int a, int b) {
    return a + b;
}

// Функція для додавання трьох цілих чисел
int add(int a, int b, int c) {
    return a + b + c;
}

// Функція для додавання двох чисел з плаваючою крапкою
double add(double a, double b) {
    return a + b;
}

int main() {
    std::cout << add(2, 3) << "\n";       // виклик першої функції → 5
    std::cout << add(2, 3, 4) << "\n";    // виклик другої функції → 9
    std::cout << add(2.5, 3.7) << "\n";   // виклик третьої функції → 6.2
    return 0;
}
```

Правила перевантаження
- Ім’я функції однакове.
- Список параметрів (кількість або типи) має відрізнятися.

- Тип повернення не може бути єдиною відмінністю (тобто не можна перевантажувати функції лише за типом результату).

Практичне застосування

- Зручність: одна назва для схожих дій (наприклад, print() для різних типів даних).
- Гнучкість: можна працювати з різними наборами параметрів без створення нових імен функцій.
- Читабельність: код стає більш зрозумілим і компактним.