І.М. Данилюк, Г.М. Унгурян

Програмування. Мова С

Міністерство освіти і науки України
Чернівецький національний університет
імені Юрія Федьковича

І.М. Данилюк, Г.М. Унгурян

Програмування. Мова С

Навчальний посібник

Чернівці
Чернівецький національний університет
імені Юрія Федьковича
2025

УДК 004.438(075.8)
Д 183

Рекомендовано Вченою радою
Чернівецького національного університету імені Юрія Федьковича
(протокол № 14 від 27 жовтня 2025 р.)

Рецензенти:

Дмитришин М.І., доктор фізико-математичних наук, професор, завідувач кафедри диференціальних рівнянь і прикладної математики Карпатського національного університету імені Василя Стефаника;

Журавльов В.П., доктор фізико-математичних наук, професор, завідувач кафедри вищої та прикладної математики Поліського національного університету.

Данилюк І.М., Унгурян Г.М.

Д 183 Програмування. Мова С : навч. посібник / І.М. Данилюк, Г.М. Унгурян. Чернівці : Чернівец. нац. ун-т ім. Ю. Федьковича, 2025. 268 с.

ISBN 978-617-8703-21-9

У виданні висвітлено основи програмування, необхідні в математичному моделюванні, числових методах і розробці алгоритмів.

Для студентів спеціальності "F1 – Прикладна математика" та інших спеціальностей, таких як "F2 – Інженерія програмного забезпечення", "F3 – Комп'ютерні науки", а також для початківців і професіоналів, які прагнуть поглибити розуміння низькорівневого програмування чи перейти до вивчення C++ або інших похідних мов.

УДК 004.438(075.8)

ISBN 978-617-8703-21-9

Зміст

Вступ

Мова програмування C є однією з найвпливовіших і найпоширеніших мов в історії комп'ютерних наук. Її поява стала поворотним моментом у розвитку програмування, заклавши фундамент для багатьох сучасних мов і технологій.

Мова C створена на початку 1970-х років у лабораторіях Bell Labs у США Деннісом Рітчі (Dennis Ritchie) з суттєвим внеском Кена Томпсона (Ken Thompson). Її розробка була тісно пов'язана з операційною системою UNIX, яка на той час створювалася як портативна платформа для різноманітного апаратного забезпечення. C стала еволюційним кроком від попередніх мов, таких як B (розроблена Томпсоном) та BCPL (створена Мартіном Річардсом). B, наприклад, була спрощеною мовою, яка використовувалася для ранніх версій UNIX, але мала обмеження в роботі з апаратним забезпеченням і типізацією даних. C, названа так, як наступна літера алфавіту після B, була представлена в 1972 році як мова, що поєднувала простоту асемблера з можливостями високорівневих мов.

Основною мотивацією створення C було забезпечення розробників інструментом, який би дозволяв писати ефективний, низькорівневий код для керування апаратним забезпеченням, зберігаючи при цьому читабельність і структурованість. У 1973 році значна частина UNIX була переписана на C, що стало першим прикладом використання мови для створення операційної системи – і цей крок довів її потенціал. Перша офіційна документація, "The C Programming Language" авторства Браяна Кернігана (Brian Kernighan) і Денніса Рітчі, відома як "K&R C", вийшла в 1978 році, ставши класичним підручником і де-факто стандартом [1].

C вирізняється унікальним поєднанням характеристик, які зробили її незамінною свого часу й зберігають її актуальність донині:

- **Низькорівневий доступ:** C дозволяє напряму працювати з пам'яттю через вказівники, що забезпечує високу продуктивність і контроль.

- **Простота і компактність:** мова має невеликий набір ключових слів (близько 32 у K&R C), що полегшує її освоєння та компіляцію.
- **Портативність:** код на C легко адаптується до різних платформ завдяки мінімальній залежності від апаратного забезпечення.
- **Гнучкість:** відсутність строгих обмежень дозволяє розробникам реалізовувати широкий спектр завдань – від системного програмування до прикладних програм.
- **Ручне керування пам'яттю:** розробник відповідає за виділення і звільнення пам'яті, що підвищує ефективність, але вимагає уважності.

Ці особливості зробили C ідеальним вибором для системного програмування, розробки вбудованих систем і створення компіляторів у 1970 – 80-х роках.

Розвиток C відбувався у кілька етапів. Спочатку K&R C залишався неформальним стандартом, але з ростом популярності мови виникла потреба в уніфікації. У 1983 році Американський національний інститут стандартів (ANSI) почав роботу над стандартизацією, яка завершилася в 1989 році випуском ANSI C (офіційно C89). Цей стандарт додав чіткі правила для типів даних, прототипів функцій і бібліотек, зробивши мову більш надійною. У 1990 році ANSI C був адаптований як міжнародний стандарт ISO/IEC 9899:1990, відомий як C90.

Подальші оновлення включали:

C99 (1999): Додав підтримку змінних довжини масивів (VLA), вбудованих комплексних чисел, поліпшену обробку десяткових чисел із плаваючою комою та інші можливості.

C11 (2011): Увів багатопоточність, атомарні операції та вдосконалив безпеку типів.

C17 (2018): Виправлення помилок C11 без значних нововведень.

C23 (опублікований ISO 31.10.2024): Додано нові функціональні можливості, такі як поліпшена підтримка Unicode і сучасні конструкції.

Кожен стандарт зберігав зворотну сумісність, що сприяло довголіттю мови. Паралельно розвивалися різні реалізації C, такі як Turbo C, GCC і Clang, адаптовані під конкретні платформи та компілятори.

C стала основою для багатьох сучасних мов програмування:

- **C++:** розроблена Б'ярном Страуструпом у 1980-х як розширення C з об'єктно-орієнтованим підходом.
- **Objective-C:** використовується Apple для macOS і iOS, додаючи об'єктну модель поверх C.
- **C#:** розроблена Microsoft як частина .NET, поєднує ідеї C і C++ із безпечнішим управлінням пам'яттю.
- **Java:** хоча синтаксично подібна до C, використовує віртуальну машину для портативності.

Ці мови успадкували синтаксис і логіку C, але додали нові парадигми, такі як об'єктно-орієнтоване програмування чи автоматичне управління пам'яттю.

Актуальність C:

- **На початку виникнення (1970-ті):** C була революційною завдяки своїй здатності замінити асемблер у системному програмуванні. Її використання в UNIX забезпечило швидке поширення, адже операційна система стала стандартом для академічних і комерційних систем.
- **У процесі розвитку (1980-90-ті):** Зі стандартизацією ANSI C мова стала основою для розробки операційних систем (наприклад, Windows NT), драйверів, вбудованих систем і компіляторів. Її портативність дозволила адаптувати програми до різноманітного обладнання – від мейнфреймів до мікроконтролерів.
- **У сучасних реаліях (2020-ті):** C залишається актуальною в областях, де потрібна максимальна продуктивність і контроль: операційні системи (Linux kernel), вбудовані системи (IoT, автомобільна електроніка), ігрові рушії (Unreal Engine), а також у високопродуктивних обчисленнях. Хоча високорівневі мови, такі як Python і

Java, домінують у веброзробці та аналізі даних, C зберігає свою нішу там, де важливі швидкість і ефективність. За індексом TIOBE (станом на 2025 рік), C стабільно входить у топ-5 мов програмування.

Окрім офіційних стандартів, існували й існують різні діалекти C:

- **K&R C:** Оригінальний варіант, що передував стандартизації.
- **Embedded C:** Розширення для вбудованих систем із специфічними типами даних.
- **MISRA C:** Набір правил для підвищення безпеки в автомобільній та авіаційній промисловості. Ці варіанти адаптують мову до спеціалізованих потреб, зберігаючи її ядро.

Посібник рекомендований студентам спеціальності "F1 – Прикладна математика" для освоєння основ програмування, необхідних у математичному моделюванні, числових методах і розробці алгоритмів. C ідеально підходить для цих завдань завдяки своїй швидкості та прямому доступу до апаратних ресурсів, що дозволяє ефективно розв'язувати обчислювальні задачі, такі як симуляції, оптимізація чи обробка великих даних. Водночас посібник може бути корисним для студентів інших спеціальностей, таких як "F2 – Інженерія програмного забезпечення" чи "F3 – Комп'ютерні науки", а також для початківців і професіоналів, які прагнуть поглибити розуміння низькорівневого програмування чи перейти до вивчення C++ або інших похідних мов.

Посібник структуровано так, щоб поступово вводити читача в основи C – від базових конструкцій до складних концепцій, таких як вказівники, динамічна пам'ять і рекурсія. Кожен розділ супроводжується прикладами та питаннями для самоконтролю, що сприяє практичному засвоєнню матеріалу.

1. Мова програмування С. Основи

Мова програмування С є потужним інструментом, який вимагає від розробника чіткого розуміння базових концепцій. У цій темі розглядаються основи С: що таке алгоритм і програма, як виглядає структура програми, як працювати зі змінними, операціями та введенням/виведенням. Звертається також увага на особливості С для тих, хто звик до Python, і додано детальні пояснення для початківців.

1.1. Поняття алгоритму та програми

Алгоритм – це чіткий перелік інструкцій для розв'язання задачі за скінченну кількість кроків. Наприклад, інструкція "Додати 2 до числа" – це частина алгоритму [6].

Програма – це алгоритм, записаний мовою програмування для виконання комп'ютером. У С програма завжди починається з функції **`main()`**.

Найпростіша програма на С може бути записана так:

```
void main() { }
```

Цей код визначає функцію **`main()`** без повернення значення (void). Однак у сучасному стандарті С (починаючи з ANSI C) рекомендується використовувати **`int main()`** із поверненням значення, щоб повідомити операційній системі про успішне завершення програми. Ось приклад:

```
int main() {
  return 0; // 0 означає успішне завершення
}
```

1.2. Різниця між void main() і int main():

- **`void main()`** історично використовувався в деяких старих компіляторах, але не відповідає стандарту ANSI C. Він не повертає значення, тому операційна система не отримує інформації про результат виконання.
- **`int main()`** є стандартним підходом. Значення, яке повертається (зазвичай 0 для успіху або ненульове для

помилки), передається операційній системі. У цьому посібнику використовуватимемо **`int main()`** як стандарт.

1.3. Структура програми на C

Програма на C складається з:

1. **Директив препроцесора**: Наприклад, **`#include <stdio.h>`** для введення/виведення.
2. **Головної функції**: **`int main()`**.
3. **Тіла програми** [7]: інструкції в **`{}`**.

Приклад із виведенням тексту:

```
#include <stdio.h>

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

1.4. Змінні та типи даних

Змінна – це іменована область пам'яті. У C змінні завжди оголошуються з типом [21]:

- **`int`** – ціле число (наприклад, **`-10, 0, 42`**).
- **`float`** – дійсне число (наприклад, **`3.14, -0.5`**).

Оголошення змінних:

```
int a, b; // Два цілих числа
float x; // Дійсне число
```

Правила іменування такі ж, як у Python, але C чутливий до регістру (**`A ≠ a`**):

- Ім'я складається з латинських літер, цифр і символу **`_`**.
- Перший символ не може бути цифрою.
- C розрізняє великі та малі літери (**`a`** і **`A`** – різні змінні).

1.5. Базові операції над змінними

C підтримує стандартні арифметичні операції:

- `+` (додавання), `-` (віднімання), `*` (множення), `/` (ділення).
- `%` (остача від ділення, тільки для `int`).
- `=` (присвоєння) – запис значення в змінну.

Особливості:

1. Піднесення до степеня: У C немає вбудованої операції `**` (як у Python). Для цього використовується функція `pow()` із бібліотеки `math.h`. Наприклад:

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

int main() {
  float result = pow(2, 3); // 2^3 = 8
  printf("2^3 = %.0f\n", result);
  return 0;
}
```

Потрібно підключити `math.h` і, можливо, додати опцію `-lm` під час компіляції (наприклад, `gcc program.c -lm`).

2. Ділення з цілими операндами: Якщо обидва операнди типу `int`, результат ділення – ціле число (дробова частина відкидається). Наприклад:

```
int a = 7 / 2; // a = 3, а не 3.5
```

Щоб отримати дійсний результат, одним із операндів має бути `float`:

```
float b = 7.0 / 2; // b = 3.5
```

```
#include <stdio.h>
int main() {
  int x = 10 / 3; // x = 3
  float y = 10.0 / 3; // y = 3.333...
  printf("x = %d, y = %.2f\n", x, y);
  return 0;
}
```

1.6. Інкрементні та декрементні операції

С пропонує зручні скорочення для зміни значень змінних:

- **++** – збільшення на **1**.
- **--** – зменшення на **1**.
- **+=n**, **-=n**, ***=n**, **/=n** – зміна на задане значення.

```
#include <stdio.h>

int main() {
    int i = 5;
    i++;          // i = 6
    i += 3;       // i = 9
    i *= 2;       // i = 18
    printf("i = %d\n", i);
    return 0;
}
```

1.7. Коментарі в коді

Коментарі пояснюють код і не впливають на його виконання:

```
Однорядкові: // текст.
Багаторядкові: /* текст */.
```

Приклад:

```
#include <stdio.h>

void main() {
    int x = 5; // Оголошення змінної x
    /* Виведення значення змінної
       на екран */
    printf("x = %d\n", x);
}
```

1.8. Форматований ввід/вивід

У Python введення/виведення просте: **print()** і **input()**. У С ці операції складніші через ручне форматування та роботу з пам’яттю.

Виведення з printf()

```
Синтаксис: printf("формат", змінні);
```

- **Специфікатори**: `%d` (int), `%f` (float), `%c` (символ), `%s` (рядок).
- **Модифікатори**:
  - `%4d` – ширина 4 символи.
  - `%.2f` – 2 знаки після коми.
  - `%4.2f` – комбінація.

Приклад (аналог `print(f"a = {a}")` у Python):

```
#include <stdio.h>
int main() {
  float a = 3.14159;
  printf("a = %.2f\n", a); // Виведе: a = 3.14
  return 0;
}
```

На відміну від Python, у C немає автоматичного перенесення рядка – потрібно додавати `\n`.

Введення з scanf()

```
Синтаксис: scanf("формат", &змінна1,
&змінна2, ...);
```

- `&` – оператор адреси, потрібен для запису значення в змінну.
- Пробіли у форматі ігноруються, але введені дані мають відповідати порядку специфікаторів.

Приклад (аналог `a = float(input())` у Python):

```
#include <stdio.h>
int main() {
  float a;
  printf("Введіть число: ");
  scanf("%f", &a);
  printf("Ви ввели: %.2f\n", a);
```

```
  return 0;
}
```

Відмінності від Python:

1. У Python типи визначаються автоматично, у C – явно.
2. У C потрібно вказувати адресу змінної (`&`) для `scanf()`.
3. Помилка введення (наприклад, букви замість цифр) може призвести до невизначеної поведінки – у Python це обробляється винятками.

1.9. Особливості написання коду на C

1. **Крапка з комою (`;`)**: кожен оператор (крім блоків у `{}`) закінчується `;`. У Python це необов'язково.
2. **Ручне керування пам'яттю**: немає сміттєзбирача, як у Python.
3. **Компіляція**: код потрібно компілювати (`gcc file.c -o file`), а не інтерпретувати, як у Python.
4. **Відсутність вбудованих високорівневих структур**: немає списків чи словників – їх треба реалізовувати вручну.

1.10. Практичні приклади

Приклад 1.1. Обчислення площі кола

```
Завдання. Обчислити площу кола за радіусом
(S=π·r2, π=3.14159).
```

```
#include <stdio.h>
int main() {
  float radius, area;
  const float PI = 3.14159; // Константа
  printf("Введіть радіус: ");
  scanf("%f", &radius);
  area = PI * radius * radius;
  printf("Площа кола = %.2f\n", area);
```

```
  return 0;
}
```

Приклад 1.2. Переведення секунд у години, хвилини, секунди

```
Завдання. Перевести введену кількість секунд у
формат "години:хвилини:секунди".
```

```
#include <stdio.h>
int main() {
  int total_seconds, hours, minutes, seconds;
  printf("Введіть кількість секунд: ");
  scanf("%d", &total_seconds);
  hours = total_seconds / 3600;
  minutes = (total_seconds % 3600) / 60;
  seconds = total_seconds % 60;
  printf("%d секунд = %d:%d:%d\n",
         total_seconds, hours, minutes,
         seconds);
  return 0;
}
```

Приклад 1.3. Обчислення периметра прямокутника

```
Завдання. Користувач вводить довжину і ширину
прямокутника, програма обчислює периметр.
```

```
#include <stdio.h>
int main() {
  float length, width, perimeter;
  printf("Введіть довжину і ширину прямокутника:
");
  scanf("%f %f", &length, &width);
  perimeter = 2 * (length + width);
  printf("Периметр = %.2f\n", perimeter);
  return 0;
}
```

Приклад 1.4. Переведення температури з Цельсія у Фаренгейт

```
Формула: F=C·9/5+32.
```

```
#include <stdio.h>
int main() {
  float celsius, fahrenheit;
  printf("Введіть температуру в градусах
Цельсія: ");
  scanf("%f", &celsius);
  fahrenheit = celsius * 9.0 / 5.0 + 32;
  printf("Температура у Фаренгейтах: %.2f\n",
fahrenheit);
  return 0;
}
```

Приклад 1.5. Обмін значень двох змінних

```
Завдання. Поміняти місцями значення двох змінних
за допомогою третьої.
```

```
#include <stdio.h>
int main() {
  int a, b, temp;
  printf("Введіть два цілих числа: ");
  scanf("%d %d", &a, &b);
  temp = a; // Зберігаємо a
  a = b;    // Присвоюємо a значення b
  b = temp; // Присвоюємо b збережене значення a
  printf("Після обміну: a = %d, b = %d\n",
         a, b);
  return 0;
}
```

Приклад 1.6. Обчислення суми цифр числа

```
Завдання. Знайти суму цифр введеного цілого
числа.
```

```
#include <stdio.h>
int main() {
  int number, sum = 0, digit;
  printf("Введіть ціле число: ");
  scanf("%d", &number);
  while (number > 0) {
    digit = number % 10;
    sum += digit;
    number /= 10;
  }
  printf("Сума цифр = %d\n", sum);
  return 0;
}
```

Приклад 1.7. Перевірка парності числа з остачею

```
Завдання. Визначити, чи є введене число парним.
```

```
#include <stdio.h>
int main() {
  int number;
  printf("Введіть ціле число: ");
  scanf("%d", &number);
  if (number % 2 == 0) {
    printf("%d - парне\n", number);
  }
  else {
    printf("%d - непарне\n", number);
  }
  return 0;
}
```

Приклад 1.8. Таблиця множення для числа

```
Завдання. Вивести таблицю множення для введеного
числа від 1 до 10.
```

```
#include <stdio.h>
int main() {
```

```
  int number, i;
  printf("Введіть число: ");
  scanf("%d", &number);
  for (i = 1; i <= 10; i++) {
    printf("%d * %d = %d\n", number,
           i, number * i);
  }
  return 0;
}
```

Висновки

У цій темі розглянуто основи C: структура програми, змінні, базові операції та введення/виведення. Ці знання є основою для подальшого вивчення складніших конструкцій, таких як умови, цикли та масиви. Наступна тема присвячена логічним операторам і розгалуженням, які дозволять робити програми більш гнучкими та інтерактивними.

Запитання та завдання для самоконтролю

1. Що таке алгоритм у контексті програмування?
2. Як визначається програма в мові C?
3. Яка функція є точкою входу в будь-яку програму на C?
4. У чому різниця між **`void main()`** і **`int main()`**?
5. Що означає повернення значення **`0`** у функції **`int main()`**?
6. Яка роль директив препроцесора, таких як **`#include <stdio.h>`**?
7. Що таке змінна в мові C?
8. Які основні числові типи даних використовуються в C?
9. Які правила застосовуються до імен змінних у C?
10. Як оголосити кілька змінних одного типу в одному рядку?
11. Що таке оператор присвоєння в C і як він позначається?
12. Які арифметичні операції доступні для числових змінних у C?

13. Як працює операція `%` (остача від ділення) і для яких типів вона застосовується?
14. Чи є в мові C вбудована операція піднесення до степеня? Якщо ні, як її реалізувати?
15. Як поводиться операція ділення (`/`) для двох цілих чисел у C?
16. Як отримати дійсний результат ділення для двох цілих чисел?
17. Що таке форматоване виведення і яка функція використовується для цього в C?
18. Які специфікатори формату використовуються для **`int`** і **`float`** у **`printf()`**?
19. Що означає модифікатор **`%.2f`** у функції **`printf()`**?
20. Як працює функція **`scanf()`** для введення даних?
21. Поясніть, чому перед змінними в **`scanf()`** потрібно ставити символ **`&`**.
22. Як у C виводиться символ нового рядка?
23. Яка різниця між однорядковими та багаторядковими коментарями в C?
24. Що таке інкрементна операція (**`++`**) і як вона працює?
25. Як операція **`+=`** відрізняється від простого додавання і присвоєння?
26. Як скомпілювати програму на C за допомогою компілятора **`gcc`**?
27. Чим відрізняється введення/виведення в C від аналогічних операцій у Python?
28. Що станеться, якщо ввести букви замість числа при використанні **`scanf("%d", &var)`**?
29. Як оголосити константу в мові C?
30. Яке значення повертає функція **`pow()`** із бібліотеки **`math.h`**, і як її підключити?

2. Логічні оператори і вирази. Розгалуження

У програмуванні часто виникає потреба приймати рішення залежно від певних умов або виконувати різні дії залежно від введених даних. У мові С для цього використовуються логічні оператори, вирази та структури розгалуження. У цій темі детально розглядається, як С працює з логікою, як формувати умови, використовувати умовні оператори (**`if, else`**) і умовні вирази (**`?:`**), а також як ці інструменти застосовуються в практичних задачах.

2.1. Логічні оператори та булева логіка в С

С не має спеціального булевого типу даних, як, наприклад, **`bool`** у Python чи C++. Натомість логічні значення моделюються через цілі числа:

- 0 – хибність (**`false`**).
- Будь-яке ненульове значення (зазвичай 1) – істина (**`true`**).

Це дозволяє використовувати будь-який числовий вираз як умову. Наприклад, 5 або -3 вважатимуться істинними, а 0 – хибним.

2.2. Операції відношення

Операції відношення порівнюють два значення і повертають 0 (хибність) або 1 (істину):

- **`==`** – дорівнює.
- **`!=`** – не дорівнює.
- **`<`** – менше.
- **`>`** – більше.
- **`<=`** – менше або дорівнює.
- **`>=`** – більше або дорівнює.

```
#include <stdio.h>

int main() {
  int a = 5, b = 3;
  printf("a == b: %d\n", a == b);// 0 (хибність)
```

```
  printf("a > b: %d\n", a > b);  // 1 (істина)
  return 0;
}
```

2.3. Логічні операції

Логічні операції поєднують умови:

- `&&` – кон'юнкція (і): істинна, якщо обидва операнди істинні.
- `||` – диз'юнкція (або): істинна, якщо хоча б один операнд істинний.
- `!` – заперечення (не): інвертує значення (`0` → `1`, `1` → `0`).

```
#include <stdio.h>

int main() {
  int x = 1, y = 0;
  printf("x && y: %d\n", x && y);
  // 0 (x = 1, y = 0)
  printf("x || y: %d\n", x || y);
  // 1 (хоча б x = 1)
  printf("!x: %d\n", !x);
  // 0 (інверсія 1)
  return 0;
}
```

2.4. Пріоритет операцій

Логічні оператори мають нижчий пріоритет [11], ніж арифметичні, але вищий за оператор присвоєння (`=`). Порядок:

`!` (найвищий) `-->` `&&` `-->` `||` (найнижчий).

Щоб уникнути помилок, використовуйте дужки:

```
int result = (a > b) && (c < d);
// Чітке групування умов
```

2.5. Особливості булевих виразів у С

Умови завжди визначені як цілі числа, але можуть включати арифметичні вирази. Наприклад, `5 - 5` поверне `0` (хибність).

Операції **&&** і **||** використовують "ліниве обчислення": якщо результат уже зрозумілий, другий операнд не обчислюється. Наприклад,

```
int x = 0;
if (x && (5 / x)) { // Ділення на 0 не відбудеться
    printf("Не виконається\n");
}
```

2.6. Умовний вираз (?:)

Умовний вираз (тернарний оператор) – це компактна альтернатива умовному оператору [8]. Синтаксис:

```
умова ? вираз_якщо_істина : вираз_якщо_хибність
```

Результат залежить від істинності умови.

```
int z = (x < 0 && y < 0) ? 1 : 0;
// 1, якщо обидва від'ємні, інакше 0
```

Розширений приклад.

```
#include <stdio.h>
int main() {
  int a = 10, b = 15;
  int max = (a > b) ? a : b; // max = 15
  printf("Більше число: %d\n", max);
  return 0;
}
```

Тернарний оператор зручний для простих умов, але для складних краще використовувати **if**.

2.7. Умовний оператор (if, else)

Умовний оператор дозволяє виконувати різні блоки коду залежно від умови. Синтаксис:

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

```
    // код, якщо умова хибна
}
```

Якщо блок складається з одного оператора, **{}** можна опустити.

Частина **else** необов'язкова.

Приклад (максимум двох чисел).

```
#include <stdio.h>

int main() {
    float x, y, max;
    printf("Введіть два числа: ");
    scanf("%f %f", &x, &y);
    if (x >= y)
        max = x;
    else
        max = y;
    printf("Максимум: %.2f\n", max);
    return 0;
}
```

2.8. Вкладені умови

Для складніших перевірок можна вкладати **if**:

```
#include <stdio.h>

int main() {
    int num;
    printf("Введіть число: ");
    scanf("%d", &num);
    if (num > 0) {
        printf("Число додатне\n");
    } else {
        if (num < 0) {
            printf("Число від'ємне\n");
        } else {
            printf("Число дорівнює нулю\n");
        }
```

```
    }
    return 0;
}
```

Альтернатива – використання **else if**:

```
if (num > 0) {
    printf("Число додатне\n");
} else if (num < 0) {
    printf("Число від'ємне\n");
} else {
    printf("Число дорівнює нулю\n");
}
```

2.9. Оператор switch

Оператор **switch** у мові С є конструкцією керування потоком виконання, яка дозволяє виконувати різні блоки коду залежно від значення цілочислового виразу. Він часто застосовується як альтернатива множинним операторам **if-else**, коли потрібно порівняти змінну з кількома конкретними значеннями. Використання **switch** може зробити код більш читабельним і, в деяких випадках, ефективнішим завдяки оптимізації компілятором.

Загальна форма оператора **switch** виглядає так:

```
switch (вираз) {
 case константа1:
   // код для випадку, коли вираз дорівнює
   // константа1
 break;
 case константа2:
   // код для випадку, коли вираз дорівнює
   // константа2
   break;
 ...
 default:
   // код, який виконується, якщо вираз
```

```
   // не збігається з жодною константою
}
```

- **вираз** – це значення цілочислового типу (наприклад, **int** або **char**), яке обчислюється один раз.
- **константа1**, **константа2** тощо – це унікальні цілочислові константи, з якими порівнюється **вираз**
- Оператор **break** завершує виконання блоку **switch** після відповідного випадку.
- Якщо **break** відсутній, виконання "провалюється" (fall-through) до наступного випадку.
- **default** – необов'язковий випадок, який виконується, якщо **вираз** не збігається з жодною константою.

Приклади використання

Приклад 2.1: Визначення дня тижня за числом

Цей приклад демонструє, як можна використати **switch** для відображення числа (від 1 до 7) на назву дня тижня:

```
#include <stdio.h>

int main() {
 int day;
 printf("Введіть число (1-7): ");
 scanf("%d", &day);

 switch (day) {
 case 1:
 printf("Понеділок\n");
 break;
 case 2:
 printf("Вівторок\n");
            break;
        case 3:
            printf("Середа\n");
            break;
        case 4:
```

```
            printf("Четвер\n");
            break;
        case 5:
            printf("П'ятниця\n");
            break;
        case 6:
            printf("Субота\n");
            break;
        case 7:
            printf("Неділя\n");
            break;
        default:
            printf("Некоректне введення\n");
    }
    return 0;
}
```

Пояснення: Користувач вводить число, і програма виводить відповідний день тижня. Випадок **default** обробляє некоректні значення, наприклад, числа поза діапазоном 1–7.

Приклад 2.2: Класифікація літер (із провалюванням)

Тут **switch** використовується для визначення, чи є введена літера голосною, із застосуванням механізму провалювання:

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

int main() {
 char letter;
 printf("Введіть літеру: ");
 scanf(" %c", &letter);
    letter = tolower(letter);
    // Перетворюємо на малу літеру
    switch (letter) {
        case 'a':
```

```
        case 'e':
        case 'i':
        case 'o':
        case 'u':
            printf("Голосна\n");
            break;
        default:
            if (isalpha(letter)) {
                printf("Приголосна\n");
            } else {
                printf("Не літера\n");
            }
    }
    return 0;
}
```

Пояснення: Якщо введена літера – одна з голосних (**a**, **e**, **i**, **o**, **u**), випадки "провалюються" до спільного блоку коду, який виводить "Голосна". Інакше програма перевіряє, чи є символ літерою, і класифікує його як приголосну або не літеру.

Приклад 2.3: Вибір математичної операції

Цей приклад показує, як **switch** може бути корисним у прикладній математиці для вибору операції над двома числами:

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

double add(double a, double b)
     { return a + b; }
double subtract(double a, double b)
     { return a - b; }
double multiply(double a, double b)
     { return a * b; }
double divide(double a, double b)
     { return a / b; }
double power(double a, double b)
```

```
     { return pow(a, b); }

int main() {
    int choice;
    double num1, num2;
    printf("Введіть два числа: ");
    scanf("%lf %lf", &num1, &num2);
    printf("Виберіть операцію: 1-Додавання, 2-
Віднімання, 3-Множення, 4-Ділення, 5-Степінь:
");
    scanf("%d", &choice);

    switch (choice) {
        case 1:
            printf("Результат: %.2f\n",
                    add(num1, num2));
            break;
        case 2:
            printf("Результат: %.2f\n",
                    subtract(num1, num2));
            break;
        case 3:
            printf("Результат: %.2f\n",
                    multiply(num1, num2));
            break;
        case 4:
            if (num2 != 0) {
                printf("Результат: %.2f\n",
                        divide(num1, num2));
            } else {
                printf("Помилка: Ділення на
                        нуль\n");
            }
            break;
        case 5:
            printf("Результат: %.2f\n",
                    power(num1, num2));
            break;
        default:
```

```
            printf("Некоректний вибір\n");
    }
    return 0;
}
```

Пояснення: Користувач вводить два числа та вибирає операцію. Оператор **`switch`** викликає відповідну функцію, а для ділення додано перевірку на нуль. Цей приклад демонструє практичне застосування **`switch`** у задачах, пов'язаних із обчисленнями.

Важливі зауваження

- **Типи даних:** **`switch`** працює лише з цілочисловими типами (**`int`**, **`char`** тощо), але не з **`float`** чи рядками.
- **Унікальність:** Кожна константа в **`case`** має бути унікальною.
- **Провалювання:** Без **`break`** виконання продовжиться до наступного випадку, що може бути як корисним, так і джерелом помилок.
- **Default:** Рекомендується завжди включати **`default`** для обробки неочікуваних значень.

Коли використовувати switch

- Для порівняння змінної з кількома дискретними значеннями.
- Для заміни довгих ланцюжків **`if-else`**, що покращує читабельність.
- У випадках, коли компілятор може оптимізувати код (наприклад, через таблицю переходів).

2.10. Практичні приклади

Приклад 2.4. Перевірка належності до квадранта

Завдання. Перевірити, чи належить точка (x, y) третьому координатному квадранту (x < 0, y < 0).

```
#include <stdio.h>

int main() {
    float x, y;
    printf("Введіть координати x і y: ");
    scanf("%f %f", &x, &y);
    int result = (x < 0 && y < 0);
    printf("Точка належить 3-му квадранту: %d
            (1 - так, 0 - ні)\n", result);
    return 0;
}
```

Приклад 2.5. Існування трикутника

```
Завдання. Перевірити, чи існує трикутник зі
сторонами a, b, c (сума будь-яких двох сторін
має бути більшою за третю).
```

```
#include <stdio.h>

int main() {
  float a, b, c;
  printf("Введіть сторони трикутника: ");
  scanf("%f %f %f", &a, &b, &c);
  int exists = (a + b > c) && (a + c > b) &&
               (b + c > a);
  printf("Трикутник існує: %d (1 - так, 0 – ні)
          \n", exists);
  return 0;
}
```

Приклад 2.6. Класифікація числа

```
Завдання. Визначити, чи є число парним і додат-
ним.
```

```
#include <stdio.h>

int main() {
```

```
  int num;
  printf("Введіть ціле число: ");
  scanf("%d", &num);
  if (num > 0 && num % 2 == 0) {
      printf("%d - додатне і парне\n", num);
  } else if (num > 0) {
      printf("%d - додатне і непарне\n", num);
  } else if (num < 0) {
      printf("%d - від'ємне\n", num);
  } else {
      printf("Число дорівнює 0\n");
  }
  return 0;
}
```

Приклад 2.7. Перевірка високосного року

```
Завдання. Перевірити, чи є рік високосним (ділиться на 4, але не на 100, або ділиться на 400).
```

```
#include <stdio.h>

int main() {
  int year;
  printf("Введіть рік: ");
  scanf("%d", &year);
  int is_leap = (year % 4 == 0 &&
                 year % 100 != 0) ||
                (year % 400 == 0);
  if (is_leap) {
      printf("%d - високосний рік\n", year);
  } else {
      printf("%d - не високосний рік\n", year);
  }
  return 0;
}
```

Приклад 2.8. Обчислення вартості покупки з урахуванням знижки

```
Завдання. Якщо сума покупки перевищує 1000,
застосовується знижка 10%.
```

```
#include <stdio.h>

int main() {
    float price, total;
    printf("Введіть суму покупки: ");
    scanf("%f", &price);
    total = (price > 1000) ? price * 0.9 :
            price;
    printf("До сплати: %.2f\n", total);
    return 0;
}
```

Приклад 2.9. Перевірка належності до області

```
Завдання. Перевірити, чи належить точка (x, y)
області
y≤1,y≥−x,y≥x.
```

```
#include <stdio.h>

int main() {
    float x, y;
    printf("Введіть координати x і y: ");
    scanf("%f %f", &x, &y);
    if (y <= 1 && y >= -x && y >= x) {
        printf("Точка (%.2f, %.2f)
                належить області\n", x, y);
    } else {
        printf("Точка (%.2f, %.2f)
                не належить області\n", x, y);
    }
    return 0;
}
```

Приклад 2.10. Калькулятор простих операцій

Завдання. Реалізувати калькулятор для додавання, віднімання, множення або ділення.

```
#include <stdio.h>

int main() {
    float a, b, result;
    char operation;
    printf("Введіть два числа і операцію
            (+, -, *, /): ");
    scanf("%f %f %c", &a, &b, &operation);
    if (operation == '+') {
        result = a + b;
        printf("%.2f + %.2f = %.2f\n",
                a, b, result);
    } else if (operation == '-') {
        result = a - b;
        printf("%.2f - %.2f = %.2f\n",
                a, b, result);
    } else if (operation == '*') {
        result = a * b;
        printf("%.2f * %.2f = %.2f\n",
                a, b, result);
    } else if (operation == '/') {
        if (b != 0) {
            result = a / b;
            printf("%.2f / %.2f = %.2f\n",
                    a, b, result);
        } else {
            printf("Ділення на нуль неможливе!
                    \n");
        }
    } else {
        printf("Невідома операція\n");
    }
    return 0;
}
```

Приклад 2.11. Визначення знака зодіаку

```
Завдання. За днем і місяцем визначити знак
зодіаку (спрощено для двох знаків).
```

```
#include <stdio.h>

int main() {
    int day, month;
    printf("Введіть день і
            місяць народження: ");
    scanf("%d %d", &day, &month);
    if ((month == 3 && day >= 21) ||
        (month == 4 && day <= 19)) {
        printf("Ваш знак - Овен\n");
    } else if ((month == 4 && day >= 20) ||
               (month == 5 && day <= 20)) {
        printf("Ваш знак - Телець\n");
    } else {
        printf("Інший знак (не визначено
                в прикладі)\n");
    }
    return 0;
}
```

2.11. Порівняння з іншими мовами

Python: у Python логічні оператори – **`and`**, **`or`**, **`not`**, а в C – **`&&`**, **`||`**, **`!`**. Python має тип **`bool`** (**`True`**, **`False`**), а C – ні.

C++: у C++ є тип **`bool`**, але в класичному C доводиться використовувати цілі числа.

Java: у Java умови завжди повертають **`boolean`**, а в C – **`int`**, що дозволяє використовувати арифметичні вирази як умови.

2.12. Рекомендації щодо використання

Чіткість умов: завжди перевіряйте межові значення (наприклад, **`x >= 0`**, а не просто **`x > 0`**), щоб уникнути помилок.

Ліниве обчислення: використовуйте **`&&`** і **`||`** для оптимізації, уникаючи непотрібних обчислень.

Коментарі: у складних умовах додавайте пояснення для зрозумілості.

Висновки

Логічні оператори та розгалуження – це основа для створення гнучких програм, які реагують на введені дані чи обчислені значення. В темі розглянуто операції відношення, логічні оператори, умовний вираз і оператор **`if else`**, а також застосовано їх у різноманітних задачах – від геометричних перевірок до простих калькуляторів. Наступна тема – цикли – дозволить повторювати дії, комбінуючи їх із розгалуженнями для ще складніших алгоритмів.

Запитання та завдання для самоконтролю

1. Як у мові С моделюються логічні значення (істина та хибність)?
2. Яке числове значення вважається хибним у С?
3. Які операції відношення використовуються в мові С?
4. Як працює оператор **==** і чим він відрізняється від **=**?
5. Що повертає операція **!=** при порівнянні двох однакових чисел?
6. Які логічні оператори доступні в С і що вони означають?
7. Що таке кон'юнкція і як вона позначається в С?
8. Як працює диз'юнкція в логічних виразах?
9. Поясніть що робить оператор заперечення (**!**) із числовим значенням.
10. Який пріоритет мають логічні оператори **&&**, **||** і **!**?
11. Що таке "ліниве обчислення" в контексті логічних операторів?
12. Як уникнути помилок у складних логічних виразах?
13. Що таке умовний вираз (**?:**) і як він працює?
14. У чому перевага використання тернарного оператора порівняно з **`if-else`**?

15. Як записати умовний вираз для знаходження більшого з двох чисел?
16. Що таке умовний оператор у C і який його базовий синтаксис?
17. Чи обов'язкова частина **`else`** в операторі **`if-else`**?
18. Коли можна опустити фігурні дужки **`{}`** в умовному операторі?
19. Як працює вкладений **`if`** і в яких випадках його доцільно використовувати?
20. Що таке конструкція **`else if`** і як вона спрощує код?
21. Як перевірити, чи належить точка певному координатному квадранту?
22. Які умови потрібно перевірити, щоб визначити існування трикутника за трьома сторонами?
23. Як за допомогою **`if-else`** визначити, чи є число додатним, від'ємним чи нулем?
24. Як можна реалізувати простий калькулятор за допомогою умовних операторів?
25. Що станеться, якщо в умові **`if`** використати присвоєння (**`=`**) замість порівняння (**`==`**)?
26. Як перевірити, чи є рік високосним, використовуючи логічні оператори?
27. Як комбінувати кілька умов у одному логічному виразі?
28. Чи можна використовувати арифметичні вирази як умови в **`if`**? Якщо так, як це працює?
29. Як за допомогою логічних операторів перевірити, чи належить число заданому діапазону?
30. У чому різниця між логічними операторами C (**`&&`**, **`||`**, **`!`**) і аналогічними в Python (**`and`**, **`or`**, **`not`**)?

3. Циклічні конструкції

Циклічні конструкції (або просто цикли) є ключовим елементом програмування, який дозволяє виконувати певний набір інструкцій неодноразово – доки виконується певна умова або задану кількість разів. У мові С цикли дають змогу автоматизувати повторювані операції, суттєво спрощуючи код і підвищуючи його ефективність. У цій темі детально розглядаються три типи циклів у С – **`for`**, **`while`** і **`do-while`**, їхній синтаксис, правила виконання, особливості використання, а також вкладені цикли та практичні приклади.

3.1. Поняття циклу

Цикл – це структура керування, яка забезпечує повторення певної послідовності команд. Цикли поділяються на два основні типи:

- **Цикли з лічильником:** виконуються задану кількість разів (зазвичай із використанням **`for`**).
- **Цикли за умовою:** виконуються, доки умова істинна (**`while`**, **`do-while`**).

3.2. Цикл із лічильником (for)

Цикл **`for`** ідеально підходить для ситуацій, коли кількість повторень відома заздалегідь. Синтаксис:

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

- **Ініціалізація:** встановлює початкове значення лічильника (виконується один раз).
- **Умова:** перевіряється перед кожною ітерацією; якщо хибна, цикл завершується.
- **Крок:** зміна лічильника після кожної ітерації.

```
Приклад (виведення чисел від 5 до 1).
```

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

```
int main() {
    int i;
    for (i = 5; i > 0; i--) {
        printf("%d ", i);
    }
    printf("\n");
    return 0;
}
```

```
Виведення:
5 4 3 2 1
```

Як працює цикл:

1. `i = 5` – ініціалізація.
2. Перевірка `i > 0` (`5 > 0` – істинно).
3. Виконання тіла (`printf`).
4. Зменшення `i` (`i--`).
5. Повторення з пункту `2`, доки `i > 0`.

Особливості:

- Якщо тіло циклу порожнє, можна використати `;`:

```
for (int i = 0; i < 1000000; i++); // Затримка
```

- Зміна лічильника в тілі може призвести до помилок (наприклад, нескінченний цикл):

```
for (int i = 5; i > 0; i--) {
    i++; // Нескінченний цикл!
}
```

3.3. Цикли за умовою

Цикл із передумовою (while)

Цикл **`while`** виконується, доки умова істинна [4]. Синтаксис:

```
while (умова) {
```

```
    // тіло циклу
}
```

Приклад. Виведення чисел від 1 до 5.

```
#include <stdio.h>

int main() {
    int i = 1;
    while (i <= 5) {
        printf("%d ", i);
        i++;
    }
    printf("\n");
    return 0;
}
```

```
Виведення:
1 2 3 4 5
```

Особливості:

- Умова перевіряється перед виконанням тіла, якщо вона хибна спочатку, тіло не виконається жодного разу.

Цикл із післяумовою (do-while)

Цикл **do-while** гарантує хоча б одне виконання тіла. Синтаксис:

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

Приклад. Запит числа, доки не введено додатне.

```
#include <stdio.h>

int main() {
```

```
    int num;
    do {
        printf("Введіть додатне число: ");
        scanf("%d", &num);
    } while (num <= 0);
    printf("Ви ввели: %d\n", num);
    return 0;
}
```

Відмінність від **while:**

- **while** може не виконати тіло жодного разу.
- **do-while** завжди виконає тіло хоча б раз, що корисно, коли умова залежить від дій у тілі.

3.4. Вкладені цикли

Вкладені цикли – це цикли всередині інших циклів. Вони використовуються для обробки багатовимірних структур або повторення дій у кількох вимірах.

Приклад. Виведення таблиці множення (1–5).

```
#include <stdio.h>

int main() {
    int i, j;
    for (i = 1; i <= 5; i++) {
        for (j = 1; j <= 5; j++) {
            printf("%d ", i * j);
        }
        printf("\n");
    }
    return 0;
}
```

```
Виведення:
1 2 3 4 5
2 4 6 8 10
3 6 9 12 15
4 8 12 16 20
```

```
5 10 15 20 25
```

3.5. Практичні приклади

Приклад 3.1. Добуток чисел через додавання

Завдання. Обчислити m·n через додавання.

```
#include <stdio.h>

int main() {
    int m, n, i, result = 0;
    printf("Введіть два числа (m, n): ");
    scanf("%d %d", &m, &n);
    for (i = 1; i <= m; i++) {
        result += n;
    }
    printf("%d * %d = %d\n", m, n, result);
    return 0;
}
```

Приклад 3.2. Сума многочлена (схема Горнера)

Завдання. Обчислити $x^n + x^{n-1} + \cdots + x + 1$.

```
#include <stdio.h>

int main() {
    float x, y = 1;
    int n, i;
    printf("Введіть n і x: ");
    scanf("%d %f", &n, &x);
    for (i = 1; i <= n; i++) {
        y = y * x + 1;
    }
    printf("Сума = %.2f\n", y);
    return 0;
}
```

Приклад 3.3. Максимум послідовності

```
Завдання. Знайти максимум серед n чисел.
```

```
#include <stdio.h>

int main() {
    int n, i, num, max;
    printf("Введіть кількість чисел: ");
    scanf("%d", &n);
    printf("Введіть перше число: ");
    scanf("%d", &max);
    for (i = 2; i <= n; i++) {
        printf("Введіть число %d: ", i);
        scanf("%d", &num);
        if (num > max) max = num;
    }
    printf("Максимум = %d\n", max);
    return 0;
}
```

Приклад 3.4. Подвійний факторіал

```
Завдання. Обчислити n!! (наприклад, 5!!=5·3·1).
```

```
#include <stdio.h>

int main() {
    int n, k, result = 1;
    printf("Введіть n: ");
    scanf("%d", &n);
    for (k = n; k >= 1; k -= 2) {
        result *= k;
    }
    printf("%d!! = %d\n", n, result);
    return 0;
}
```

Приклад 3.5. Сума умовної послідовності

Завдання. Обчислити суму
$z_1 + z_2 + \cdots + z_n$, де $z_i = y_i$ якщо $0 < y_i < 10$
інакше $z_i = 1$.

```
#include <stdio.h>

int main() {
    int n, i;
    float y, sum = 0;
    printf("Введіть кількість чисел: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Введіть y%d: ", i);
        scanf("%f", &y);
        if (y <= 0 || y >= 10) y = 1;
        sum += y;
    }
    printf("Сума = %.2f\n", sum);
    return 0;
}
```

Приклад 3.6. Фібоначчі до заданого числа

Завдання. Вивести числа Фібоначчі, менші за введене число.

```
#include <stdio.h>

int main() {
    int limit, a = 0, b = 1, next;
    printf("Введіть межу: ");
    scanf("%d", &limit);
    printf("Числа Фібоначчі: %d %d ", a, b);
    next = a + b;
    while (next < limit) {
        printf("%d ", next);
        a = b;
```

```
        b = next;
        next = a + b;
    }
    printf("\n");
    return 0;
}
```

Приклад 3.7. Перевірка простоти числа

```
Завдання. Перевірити, чи є число простим.
```

```
#include <stdio.h>

int main() {
    int num, i, is_prime = 1;
    printf("Введіть число: ");
    scanf("%d", &num);
    if (num <= 1) is_prime = 0;
    for (i = 2; i * i <= num; i++) {
        if (num % i == 0) {
            is_prime = 0;
            break;
        }
    }
    if (is_prime) printf("%d - просте\n", num);
    else printf("%d - не просте\n", num);
    return 0;
}
```

Приклад 3.8. Обчислення суми ряду

```
Завдання. Обчислити суму 1 + 1/2 + 1/3 + ... +
1/n.
```

```
#include <stdio.h>

int main() {
    int n, i;
    float sum = 0;
```

```
    printf("Введіть n: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        sum += 1.0 / i;
    }
    printf("Сума = %.4f\n", sum);
    return 0;
}
```

3.6. Порівняння з іншими мовами

Python: у Python цикл for працює з ітераторами (**`for i in range()`**), у C – із лічильником. Цикл **`while`** схожий, але Python не має **`do-while`**.

C++: синтаксис циклів ідентичний, але C++ додає **`range-based for`**, якого немає в C.

Java: аналогічно до C, але з додатковими перевірками безпеки.

3.7. Рекомендації

Уникайте нескінченних циклів: завжди перевіряйте умову завершення.

Оптимізація: у **`for`** використовуйте локальні змінні лічильника (**`for (int i = 0; ...)`**).

Переривання: використовуйте **`break`** для передчасного виходу з циклу, **`continue`** – для пропуску ітерації.

Висновки

Циклічні конструкції – це потужний інструмент для автоматизації повторюваних задач. В цій темі розглянуто **`for`** для лічильників, **`while`** і **`do-while`** для умов, а також вкладені цикли для складніших алгоритмів. Практичні приклади показали їхнє застосування в обчисленнях, перевірках і генерації послідовностей. Наступна тема поєднає цикли з розгалуженнями для створення ще складніших програм.

Запитання та завдання для самоконтролю

1. Що таке цикл у програмуванні?
2. Які два основні типи циклів розрізняють у мові C?
3. Назвіть основну відмінність між циклами з лічильником і циклами за умовою.
4. Який синтаксис циклу for у мові C?
5. Що означає кожна частина в заголовку циклу for (ініціалізація, умова, крок)?
6. Як працює цикл for, якщо умова хибна від самого початку?
7. Що станеться, якщо змінити лічильник усередині тіла циклу for?
8. Який цикл у C називається циклом із передумовою?
9. У чому полягає різниця між while і do-while?
10. Скільки разів виконається тіло циклу do-while, якщо умова хибна з самого початку?
11. Як можна зробити цикл нескінченним у C?
12. Що таке вкладені цикли і для чого вони використовуються?
13. Як цикл for можна замінити циклом while?
14. Як організувати виведення чисел від 1 до n за допомогою циклу while?
15. Як обчислити добуток двох чисел через додавання за допомогою циклу?
16. Що таке схема Горнера і як її застосувати в циклах?
17. Як знайти максимум серед n чисел за допомогою циклу?
18. Як обчислити подвійний факторіал (n!!) за допомогою циклу?
19. Як за допомогою циклу підрахувати суму чисел із певною умовою?
20. Як вивести таблицю множення для чисел від 1 до 5 за допомогою вкладених циклів?
21. Що таке оператор break і як він впливає на цикл?
22. Як працює оператор continue у циклах?
23. Як за допомогою циклу вивести числа Фібоначчі до певної межі?

24. Як перевірити, чи є число простим, використовуючи цикл?
25. Як обчислити суму гармонічного ряду (1 + 1/2 + 1/3 + …) за допомогою циклу?
26. Як організувати введення чисел у циклі до введення певного значення (наприклад, 0)?
27. Як вивести трикутник із зірочок за допомогою вкладених циклів?
28. Як оптимізувати цикл для перевірки простоти числа?
29. У чому різниця між циклами в C і циклами в Python (наприклад, **`for`** із **`range()`**)?
30. Як уникнути переповнення змінної в циклах при роботі з великими числами?

4. Алгоритми поєднання розгалуження та повторення. Вкладені цикли. Покрокове введення та виведення даних. Рекурентні послідовності

У програмуванні часто виникає потреба комбінувати умовні конструкції (розгалуження) із циклами, щоб розв'язувати складніші задачі. Таке поєднання дозволяє створювати алгоритми, які одночасно перевіряють умови та повторюють дії. У цій темі розглянемо, як інтегрувати оператори **`if-else`** із циклами **`for`**, **`while`** і **`do-while`**, як використовувати вкладені цикли для багаторівневих обчислень, як організувати покрокове введення та виведення даних, а також як реалізовувати рекурентні послідовності без використання рекурсії.

4.1. Поєднання розгалужень із циклами

Розгалуження всередині циклів дозволяють виконувати різні дії залежно від умов на кожній ітерації. Наприклад, можна пропускати певні значення, обчислювати лише відповідні результати або припиняти цикл за певної умови.

Основний принцип

- Цикл задає повторення.
- Умова (**`if`**) визначає, що саме виконувати на кожному кроці.

Виведення лише парних чисел від 1 до 10.

```
#include <stdio.h>

int main() {
    int i;
    for (i = 1; i <= 10; i++) {
        if (i % 2 == 0) {
            printf("%d ", i);
        }
    }
```

```
    printf("\n");
    return 0;
}
```

```
Виведення:
2 4 6 8 10
```

4.2. Вкладені цикли

Вкладені цикли – це цикли всередині інших циклів, які використовуються для обробки багатовимірних даних або виконання повторюваних дій у кількох вимірах. Вони часто комбінуються з розгалуженнями для фільтрації чи обчислень.

Виведення трикутника із зірочок.

```
#include <stdio.h>

int main() {
    int n, i, j;
    printf("Введіть висоту трикутника: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        for (j = 1; j <= i; j++) {
            printf("* ");
        }
        printf("\n");
    }
    return 0;
}
```

```
Виведення для n=4:
*
* *
* * *
* * * *
```

4.3. Поєднання з розгалуженнями

Додамо умову: виводити зірочки лише на парних рядках.

```
#include <stdio.h>

int main() {
    int n, i, j;
    printf("Введіть висоту: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            for (j = 1; j <= i; j++) {
                printf("* ");
            }
            printf("\n");
        }
    }
    return 0;
}
```

```
Виведення для n=4:
* *
* * * *
```

4.4. Покрокове введення та виведення даних

Покрокове введення/виведення дозволяє користувачу вводити дані поступово, а програмі – обробляти їх у реальному часі. Це особливо корисно для роботи з послідовностями або інтерактивними програмами.

Покрокове введення чисел і підрахунок суми додатних.

```
#include <stdio.h>

int main() {
    int n, i, num;
```

```
    float sum = 0;
    printf("Введіть кількість чисел: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Введіть число %d: ", i);
        scanf("%d", &num);
        if (num > 0) {
            sum += num;
        }
    }
    printf("Сума додатних чисел = %.0f\n", sum);
    return 0;
}
```

4.5. Використання while для невизначеної кількості даних

Якщо кількість даних невідома, можна використовувати цикл із умовою завершення (наприклад, введення 0).

```
#include <stdio.h>
int main() {
    int num;
    float sum = 0;
    printf("Вводьте числа (0 для завершення):
\n");
    do {
        scanf("%d", &num);
        if (num > 0) {
            sum += num;
        }
    } while (num != 0);
    printf("Сума додатних чисел = %.0f\n", sum);
    return 0;
}
```

4.6. Рекурентні послідовності

Рекурентні послідовності – це послідовності, де кожен елемент залежить від попередніх [13]. У С їх можна реалізувати

за допомогою циклів, уникаючи рекурсії для економії пам’яті та підвищення швидкості.

Числа Фібоначчі
Формула: $F_n = F_{n-1} + F_{n-2}, F_0 = 0, F_1 = 1$.

```
#include <stdio.h>

int main() {
    int n, i, a = 0, b = 1, next;
    printf("Введіть кількість чисел Фібоначчі:
");
    scanf("%d", &n);
    printf("Послідовність: %d %d ", a, b);
    for (i = 3; i <= n; i++) {
        next = a + b;
        printf("%d ", next);
        a = b;
        b = next;
    }
    printf("\n");
    return 0;
}
```

Виведення для n=5:
0 1 1 2 3

Факторіал через рекурентність
Формула: $n! = n \cdot (n-1)!$, $0! = 1$.

```
#include <stdio.h>

int main() {
    int n, i;
    unsigned long long fact = 1; // Для великих
чисел
    printf("Введіть n: ");
    scanf("%d", &n);
```

```
    for (i = 1; i <= n; i++) {
        fact *= i;
    }
    printf("%d! = %llu\n", n, fact);
    return 0;
}
```

4.7. Практичні приклади

Приклад 4.1. Підрахунок парних і непарних чисел

Порахувати парні та непарні числа в послідовності.

```
#include <stdio.h>

int main() {
    int n, i, num, even = 0, odd = 0;
    printf("Введіть кількість чисел: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Число %d: ", i);
        scanf("%d", &num);
        if (num % 2 == 0) {
            even++;
        } else {
            odd++;
        }
    }
    printf("Парних: %d, Непарних: %d\n", even,
odd);
    return 0;
}
```

Приклад 4.2. Сума чисел, кратних 3 або 5

Знайти суму чисел від 1 до n, кратних 3 або 5.

```
#include <stdio.h>

int main() {
    int n, i, sum = 0;
    printf("Введіть n: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        if (i % 3 == 0 || i % 5 == 0) {
            sum += i;
        }
    }
    printf("Сума = %d\n", sum);
    return 0;
}
```

Приклад 4.3. Малювання ромба

Вивести ромб із зірочок.

```
#include <stdio.h>

int main() {
    int n, i, j, spaces;
    printf("Введіть половину висоти ромба: ");
    scanf("%d", &n);
    // Верхня частина
    for (i = 1; i <= n; i++) {
        for (spaces = n - i; spaces > 0;
spaces--) {
            printf(" ");
        }
        for (j = 1; j <= 2 * i - 1; j++) {
            printf("*");
        }
        printf("\n");
    }
    // Нижня частина
    for (i = n - 1; i >= 1; i--) {
```

```
        for (spaces = n - i; spaces > 0;
spaces--) {
            printf(" ");
        }
        for (j = 1; j <= 2 * i - 1; j++) {
            printf("*");
        }
        printf("\n");
    }
    return 0;
}
```

```
Виведення для n=3:
  *
 ***
*****
 ***
  *
```

Приклад 4.4. Середнє арифметичне додатних чисел

```
Обчислити середнє арифметичне додатних чисел із
послідовності.
```

```
#include <stdio.h>

int main() {
    int n, i, num, count = 0;
    float sum = 0, avg;
    printf("Введіть кількість чисел: ");
    scanf("%d", &n);
    for (i = 1; i <= n; i++) {
        printf("Число %d: ", i);
        scanf("%d", &num);
        if (num > 0) {
            sum += num;
            count++;
        }
    }
```

```
    if (count > 0) {
        avg = sum / count;
        printf("Середнє додатних = %.2f\n",
avg);
    } else {
        printf("Додатних чисел немає\n");
    }
    return 0;
}
```

Приклад 4.5. Ряд Тейлора для e^x

Обчислити $e^x \approx 1+x+ x^2/2! + x^3/3! + ...$ для n членів.

```
#include <stdio.h>

int main() {
    float x, sum = 1, term = 1;
    int n, i;
    printf("Введіть x і кількість членів: ");
    scanf("%f %d", &x, &n);
    for (i = 1; i < n; i++) {
        term *= x / i; // Рекурентний член
        sum += term;
    }
    printf("e^%.2f ≈ %.6f\n", x, sum);
    return 0;
}
```

Приклад 4.6. Перевірка паліндрома (числового)

Перевірити, чи є число паліндромом (наприклад, 12321).

```
#include <stdio.h>

int main() {
    int num, original, reversed = 0, digit;
```

```
    printf("Введіть число: ");
    scanf("%d", &num);
    original = num;
    while (num > 0) {
        digit = num % 10;
        reversed = reversed * 10 + digit;
        num /= 10;
    }
    if (original == reversed) {
        printf("%d - паліндром\n", original);
    } else {
        printf("%d - не паліндром\n", original);
    }
    return 0;
}
```

Приклад 4.7. Генерація простих чисел

```
Вивести перші n простих чисел.
```

```
#include <stdio.h>

int main() {
    int n, count = 0, num = 2, i, is_prime;
    printf("Введіть кількість простих чисел: ");
    scanf("%d", &n);
    printf("Прості числа: ");
    while (count < n) {
        is_prime = 1;
        for (i = 2; i * i <= num; i++) {
            if (num % i == 0) {
                is_prime = 0;
                break;
            }
        }
        if (is_prime) {
            printf("%d ", num);
            count++;
        }
```

```
        num++;
    }
    printf("\n");
    return 0;
}
```

Приклад 4.8. Обчислення НСД двох чисел

Знайти найбільший спільний дільник за алгоритмом Евкліда.

```
#include <stdio.h>

int main() {
    int a, b, temp;
    printf("Введіть два числа: ");
    scanf("%d %d", &a, &b);
    while (b != 0) {
        temp = b;
        b = a % b;
        a = temp;
    }
    printf("НСД = %d\n", a);
    return 0;
}
```

Приклад 4.9. Виведення шахової дошки (новий приклад)

Вивести шахову дошку розміром n×n, де 1 – чорна клітина, 0 – біла.

```
#include <stdio.h>

int main() {
    int n, i, j;
    printf("Введіть розмір дошки: ");
    scanf("%d", &n);
    for (i = 0; i < n; i++) {
        for (j = 0; j < n; j++) {
```

```
            if ((i + j) % 2 == 0) {
                printf("0 "); // Біла клітина
            } else {
                printf("1 "); // Чорна клітина
            }
        }
        printf("\n");
    }
    return 0;
}
```

```
Виведення для n=4:
0 1 0 1
1 0 1 0
0 1 0 1
1 0 1 0
```

Приклад 4.10. Обчислення суми парного степеня (новий приклад)

Обчислити суму $1^2+2^2+3^2+\cdots+n^2$ для парних n.

```
#include <stdio.h>

int main() {
    int n, i, sum = 0;
    printf("Введіть n: ");
    scanf("%d", &n);
    if (n % 2 != 0) {
        printf("Введіть парне число!\n");
        return 1;
    }
    for (i = 1; i <= n; i++) {
        if (i % 2 == 0) {
            sum += i * i;
        }
    }
    printf("Сума квадратів парних чисел до %d =
%d\n", n, sum);
```

```
    return 0;
}
```

```
Виведення для n=4:
Сума квадратів парних чисел до 4 = 20
тобто 22+42=4+16=20)
```

4.8. Порівняння з іншими мовами

Python: у Python вкладені цикли простіші через range(), а рекурентні послідовності часто реалізовані через списки. C не має таких вбудованих засобів [14].

C++: синтаксис циклів той самий, але C++ додає ітератори та контейнери.

Java: аналогічно до C, але з більшим акцентом на безпеку.

4.9. Рекомендації

Контроль меж: перевіряйте умови, щоб уникнути переповнення.

Оптимізація: уникайте зайвих обчислень у вкладених циклах.

Читабельність: використовуйте зрозумілі імена змінних і коментарі.

Висновки

Поєднання розгалужень із циклами відкриває широкі можливості для створення складних алгоритмів. Ми розглянули вкладені цикли, покрокове введення/виведення та рекурентні послідовності, застосувавши їх у різноманітних задачах – від геометричних фігур до математичних обчислень. Наступна тема про одновимірні масиви розширить наші можливості роботи з даними.

Запитання та завдання для самоконтролю

1. Як можна комбінувати розгалуження та цикли в алгоритмах?
2. Висвітліть роль умовного оператора if усередині циклу.

3. Як за допомогою циклу та умови вивести лише парні числа з діапазону?
4. Що таке вкладені цикли і в яких випадках їх доцільно використовувати?
5. Як працює вкладений цикл у контексті виведення геометричних фігур?
6. Як за допомогою вкладених циклів вивести трикутник із зірочок лише на парних рядках?
7. Що таке покрокове введення даних і як його реалізувати в C?
8. Як організувати введення чисел у циклі з підрахунком суми додатних?
9. Як за допомогою циклу do-while реалізувати введення даних до введення певного значення?
10. Що таке рекурентна послідовність?
11. Як обчислити числа Фібоначчі за допомогою циклу без рекурсії?
12. Як реалізувати обчислення факторіала через рекурентність у циклі?
13. Як підрахувати кількість парних і непарних чисел у послідовності за допомогою циклу та умови?
14. Як знайти суму чисел, кратних 3 або 5, у заданому діапазоні?
15. Як за допомогою вкладених циклів намалювати ромб із зірочок?
16. Поясніть як обчислити середнє арифметичне додатних чисел у послідовності
17. Як реалізувати наближене обчислення e^x за рядом Тейлора в циклі?
18. Як перевірити, чи є число паліндромом, використовуючи цикл і умову?
19. Як вивести перші n простих чисел за допомогою вкладених циклів?
20. Як обчислити НСД двох чисел за алгоритмом Евкліда в циклі?

21. Як за допомогою вкладених циклів вивести шахову дошку?
22. Як обчислити суму квадратів парних чисел до n за допомогою циклу та умови?
23. Як оптимізувати вкладені цикли для зменшення кількості ітерацій?
24. Як за допомогою циклу та розгалуження вивести лише ті числа, які діляться на 7, із заданого діапазону?
25. Як реалізувати виведення прямокутника з умовою заповнення лише країв?
26. Як за допомогою циклів і умов знайти найбільший елемент послідовності, який є кратним 4?
27. Як реалізувати покрокове введення чисел із виведенням їх у зворотному порядку?
28. Як обчислити суму членів геометричної прогресії за допомогою циклу?
29. У чому різниця між реалізацією рекурентних послідовностей у C та Python?
30. Як уникнути помилок при комбінуванні складних умов і циклів у програмах?

5. Одновимірні масиви. Найпростіші алгоритми роботи з одновимірними масивами. Пошук заданого елемента, пошук мінімального/максимального елемента

У програмуванні часто виникає потреба працювати з групами однотипних даних, наприклад, списком чисел, символів чи інших елементів. У мові С для цього використовуються масиви – структури даних, які дозволяють зберігати кілька значень під одним іменем із доступом до них за допомогою індексів. У цій темі детально розглянемо одновимірні масиви: їх оголошення, ініціалізацію, базові операції, а також найпростіші алгоритми для роботи з ними, такі як пошук заданого елемента та визначення мінімального й максимального значень.

Масиви є фундаментальною концепцією, яка відкриває двері до складніших структур даних і алгоритмів. У С масиви мають свої особливості, такі як статична природа та прямий доступ до пам'яті, що робить їх потужними, але водночас вимагає обережності від розробника.

5.1. Поняття масиву

Масив – це впорядкований набір однотипних даних, об'єднаних під спільним іменем [12]. Основні характеристики масиву в С:

- **Однотипність:** усі елементи масиву належать до одного типу даних (наприклад, `int`, `float`, `char`).
- **Прямий доступ:** кожен елемент має унікальний індекс, що дозволяє швидко звертатися до нього.
- **Фіксований розмір:** кількість елементів визначається під час оголошення і не може змінюватися під час виконання програми.

Одновимірний масив – це найпростіший тип масиву, який можна уявити як лінійний список елементів. Наприклад, масив чисел {1, 2, 3, 4, 5} є одновимірним масивом із п'ятьма елементами.

5.2. Оголошення та ініціалізація масивів

Оголошення масиву

У мові С одновимірний масив оголошується за допомогою такого синтаксису [20]:

```
тип_даних ім'я_масиву[розмір];
```

- **тип_даних** – тип елементів масиву (наприклад, **int**, **float**).
- **ім'я_масиву** – ідентифікатор масиву.
- **розмір** – кількість елементів (ціле невід'ємне число).

Приклад:

```
int numbers[5];   // Масив із 5 цілих чисел
float values[10]; // Масив із 10 дійсних чисел
```

Важливо: у С індексація масивів починається з **0**. Отже, для масиву **numbers[5]** індекси будуть від **0** до **4**.

Ініціалізація масиву

Масив можна ініціалізувати під час оголошення, задаючи початкові значення елементів у фігурних дужках:

```
int numbers[5] = {1, 2, 3, 4, 5};
```

Якщо кількість заданих значень менша за розмір масиву, решта елементів автоматично заповнюється нулями:

```
int numbers[5] = {1, 2}; // numbers = {1, 2, 0,
0, 0}
```

Можна також опустити розмір масиву, якщо він визначається кількістю ініціалізованих елементів:

```
int numbers[] = {1, 2, 3, 4, 5};
// Автоматично розмір = 5
```

Без ініціалізації значення елементів будуть невизначеними (сміття з пам'яті):

```
int array[5]; // Елементи мають випадкові значення
```

Робота з пам’яттю

У С масиви є вказівниками на початкову адресу блоку пам’яті, де зберігаються їхні елементи. Елементи розташовуються послідовно, що забезпечує швидкий доступ, але також означає, що розмір масиву фіксується під час компіляції. Наприклад, для int **numbers[5]** компілятор виділяє пам’ять для 5 цілих чисел (зазвичай 4 байти на елемент, тобто 20 байтів загалом на 32-бітній системі).

5.3. Базові операції з масивами

У С немає вбудованих операцій для роботи з масивами як цілісними об’єктами (наприклад, додавання чи порівняння масивів). Усі дії виконуються поелементно за допомогою циклів.

Доступ до елементів

Доступ до елементів масиву здійснюється за допомогою індексу в квадратних дужках:

```
numbers[0] = 10; // Зміна першого елемента
int value = numbers[2]; // Читання третього елемента
```

Введення даних у масив

Для заповнення масиву даними від користувача використовуються цикли:

```
#include <stdio.h>

int main() {
    int arr[5], i;
    printf("Введіть 5 чисел:\n");
    for (i = 0; i < 5; i++) {
        scanf("%d", &arr[i]);
    }
```

```
    return 0;
}
```

Виведення елементів масиву

Виведення також виконується поелементно:

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5}, i;
    printf("Елементи масиву: ");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

```
Виведення:
Елементи масиву: 1 2 3 4 5
```

5.4. Найпростіші алгоритми роботи з одновимірними масивами

Пошук заданого елемента

Пошук елемента в масиві – це базовий алгоритм, який перевіряє, чи є задане значення серед елементів масиву. Найпростіший метод – лінійний пошук.

```
Перевірка наявності числа в масиві.
```

```
#include <stdio.h>

int main() {
    int arr[5] = {3, 7, 2, 9, 4};
    int target, i, found = 0;
    printf("Введіть число для пошуку: ");
```

```
    scanf("%d", &target);
    for (i = 0; i < 5; i++) {
        if (arr[i] == target) {
            found = 1;
            printf("Число %d знайдено за
індексом %d\n", target, i);
            break;
        }
    }
    if (!found) {
        printf("Число %d не знайдено\n",
target);
    }
    return 0;
}
```

```
Якщо target = 7, виведення: Число 7 знайдено за
індексом 1.
Якщо target = 6, виведення: Число 6 не знайдено.
```

Особливості:

- Часова складність: `O(n)`, де `n` – розмір масиву.
- Використання break дозволяє припинити пошук після першого збігу.

Пошук мінімального елемента

Для знаходження найменшого значення в масиві порівнюємо кожен елемент із поточним мінімумом.

```
#include <stdio.h>

int main() {
    int arr[5] = {8, 3, 9, 1, 6};
    int min = arr[0], i;
    for (i = 1; i < 5; i++) {
        if (arr[i] < min) {
            min = arr[i];
        }
    }
```

```
    printf("Мінімальний елемент: %d\n", min);
    return 0;
}
```

```
Виведення:
Мінімальний елемент: 1
```

Пошук максимального елемента

Аналогічно до пошуку мінімуму, але порівнюємо з поточним максимумом.

```
#include <stdio.h>

int main() {
    int arr[5] = {8, 3, 9, 1, 6};
    int max = arr[0], i;
    for (i = 1; i < 5; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    printf("Максимальний елемент: %d\n", max);
    return 0;
}
```

```
Виведення:
Максимальний елемент: 9
```

Пошук індексу мінімуму/максимуму

Часто потрібно не лише значення, а й індекс мінімального чи максимального елемента.

```
#include <stdio.h>

int main() {
    int arr[5] = {8, 3, 9, 1, 6};
    int max = arr[0], max_index = 0, i;
    for (i = 1; i < 5; i++) {
```

```
        if (arr[i] > max) {
            max = arr[i];
            max_index = i;
        }
    }
    printf("Максимальний елемент %d за індексом
%d\n", max, max_index);
    return 0;
}
```

```
Виведення:
Максимальний елемент 9 за індексом 2
```

5.5. Розширені приклади роботи з масивами

Приклад 5.1. Сума елементів масиву

```
Завдання. Обчислити суму всіх елементів масиву.
```

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int sum = 0, i;
    for (i = 0; i < 5; i++) {
        sum += arr[i];
    }
    printf("Сума елементів: %d\n", sum);
    return 0;
}
```

```
Виведення:
Сума елементів: 15
```

Приклад 5.2. Кількість елементів, більших за задане значення

```
Завдання. Підрахувати, скільки елементів масиву
перевищують задане число.
```

```
#include <stdio.h>

int main() {
    int arr[5] = {4, 7, 2, 9, 3};
    int threshold, count = 0, i;
    printf("Введіть порогове значення: ");
    scanf("%d", &threshold);
    for (i = 0; i < 5; i++) {
        if (arr[i] > threshold) {
            count++;
        }
    }
    printf("Кількість елементів, більших за %d:
%d\n", threshold, count);
    return 0;
}
```

```
Якщо threshold = 5, виведення: Кількість
елементів, більших за 5: 2 (7 і 9).
```

Приклад 5.3. Реверс масиву

```
Завдання. Перевернути порядок елементів у маси-
ві.
```

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int i, temp;
    printf("Початковий масив: ");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
```

```
    }
    printf("\n");
    // Реверс
    for (i = 0; i < 5 / 2; i++) {
        temp = arr[i];
        arr[i] = arr[4 - i];
        arr[4 - i] = temp;
    }
    printf("Перевернутий масив: ");
    for (i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

```
Виведення:
Початковий масив: 1 2 3 4 5
Перевернутий масив: 5 4 3 2 1
```

Приклад 5.4. Пошук другого максимуму

```
Завдання. Знайти друге за величиною значення в
масиві.
```

```
#include <stdio.h>

int main() {
    int arr[5] = {8, 3, 9, 1, 6};
    int max = arr[0], second_max = arr[0], i;
    // Знаходимо максимум
    for (i = 1; i < 5; i++) {
        if (arr[i] > max) {
            max = arr[i];
        }
    }
    // Знаходимо другий максимум
    for (i = 0; i < 5; i++) {
```

```
        if (arr[i] > second_max && arr[i] < max)
{
            second_max = arr[i];
        }
    }
    printf("Другий максимум: %d\n", second_max);
    return 0;
}
```

```
Виведення:
Другий максимум: 8
(оскільки максимум – 9)
```

Приклад 5.5. Підрахунок парного/непарного

```
Завдання. Підрахувати кількість парних і
непарних чисел у масиві.
```

```
#include <stdio.h>
int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    int even = 0, odd = 0, i;
    for (i = 0; i < 5; i++) {
        if (arr[i] % 2 == 0) {
            even++;
        } else {
            odd++;
        }
    }
    printf("Парних: %d, Непарних: %d\n", even,
odd);
    return 0;
}
```

```
Виведення:
Парних: 2, Непарних: 3
```

5.6. Порівняння з іншими мовами

Python: у Python масиви замінені списками (`list`), які є динамічними й дозволяють змінювати розмір. У C масиви статичні, що вимагає заздалегідь визначеного розміру.

C++: C++ успадковує масиви від C, але додає динамічні контейнери, такі як `std::vector`, тоді як у C таких можливостей немає.

Java: у Java масиви також статичні, але мають вбудовані методи для роботи з ними, на відміну від C, де все реалізується вручну.

5.7. Рекомендації щодо використання

Перевірка меж: завжди контролюйте індекси, щоб уникнути виходу за межі масиву, що може призвести до невизначеної поведінки.

Ініціалізація: ініціалізуйте масиви, щоб уникнути роботи з випадковими значеннями.

Оптимізація: для великих масивів використовуйте локальні змінні для індексів і уникайте зайвих обчислень у циклах.

Читабельність: додавайте коментарі для пояснення логіки алгоритмів.

Висновки

Одновимірні масиви є основою для роботи з групами даних у C. Ми розглянули їхнє оголошення, ініціалізацію, базові операції та прості алгоритми, такі як пошук елементів, мінімуму й максимуму. Розширені приклади показали, як масиви застосовуються для обчислень, реверсу та аналізу даних. Наступна тема – робота з файлами розширить наші можливості для збереження та обробки масивів.

Запитання та завдання для самоконтролю

1. Що таке одновимірний масив у мові C?
2. Як оголошується масив у C? Назвіть основні складові синтаксису.

3. Який індекс має перший елемент масиву в С?
4. Як ініціалізувати масив під час оголошення?
5. Що станеться, якщо під час ініціалізації масиву вказати менше значень, ніж його розмір?
6. Чи можна змінити розмір масиву після його оголошення?
7. Як звернутися до третього елемента масиву arr?
8. Як заповнити масив значеннями, введеними користувачем?
9. Як вивести всі елементи масиву на екран?
10. Що таке лінійний пошук і як він працює з масивами?
11. Як знайти задане значення в масиві за допомогою циклу?
12. Що поверне програма, якщо заданого елемента немає в масиві?
13. Як знайти мінімальний елемент у масиві?
14. Як знайти максимальний елемент у масиві?
15. Як визначити індекс максимального елемента в масиві?
16. Як обчислити суму всіх елементів масиву?
17. Як підрахувати кількість елементів, більших за задане значення?
18. Як перевернути порядок елементів у масиві?
19. Як знайти другий максимум у масиві?
20. Як підрахувати кількість парних і непарних чисел у масиві?
21. Що станеться, якщо звернутися до елемента за індексом, який виходить за межі масиву?
22. Як уникнути невизначеної поведінки при роботі з неініціалізованим масивом?
23. Як оптимізувати пошук максимального елемента в масиві?
24. Як за допомогою масиву реалізувати зберігання оцінок студентів?
25. Як перевірити, чи всі елементи масиву додатні?
26. Як знайти середнє арифметичне елементів масиву?
27. Як замінити всі від’ємні елементи масиву на нуль?

28. У чому різниця між масивами в C і списками в Python?
29. Як масиви пов’язані з пам’яттю комп’ютера в C?
30. Як за допомогою масиву знайти кількість входжень заданого числа?

6. Додаткові способи введення даних

У попередніх темах розглядали базові методи введення даних через функцію **`scanf()`**, яка дозволяє користувачу вводити значення з клавіатури в інтерактивному режимі. Проте в реальних задачах програмування часто виникає потреба отримувати дані з інших джерел, таких як файли, генерувати випадкові числа для тестування чи моделювання, або ініціалізувати масиви константами для спрощення роботи з фіксованими наборами даних. У цій темі детально розглянемо три ключові аспекти додаткових способів введення даних у мові С: роботу з файлами та потоками, генерацію випадкових чисел і методи ініціалізації масивів константами.

Ці методи фундаментальні для створення програм, які можуть працювати з великими обсягами даних, зберігати результати для подальшого використання або імітувати реальні сценарії. Звернемо увагу на особливості мови С, такі як низькорівневий доступ до файлів, відсутність вбудованої типізації файлів і необхідність ручного керування ресурсами, що відрізняє С від високорівневих мов, таких як Python чи Java.

6.1. Робота з файлами та потоками

Основи роботи з файлами

Файл – це іменований блок інформації, що зберігається на носії даних. Він характеризується:

- **Ім’ям:** унікальна послідовність символів, наприклад, **`"data.txt"`**.
- **Логічною структурою:** визначає спосіб читання та запису (текстовий чи двійковий).
- **Розміром:** кількість байтів інформації.

Файли є найменшою одиницею збереження даних на носії, на відміну від змінних, які існують лише під час виконання програми. У С файли поділяються на текстові (символьні дані, організовані в рядки) і двійкові (послідовність байтів будь-якого типу). В С файли нетипізовані, тобто програміст самостійно інтерпретує їхній вміст, що вимагає ретельного планування.

Потоки в мові С

Потік (**stream**) – це абстракція, яка представляє джерело або приймач даних [18]. У **<stdio.h>** потоки реалізуються через структуру **FILE**, а вказівник на неї (**FILE ***) є основним інструментом роботи з файлами. Наприклад:

```
FILE *fp;
```

Тут **fp** – файлова змінна, яка після зв'язування з файлом через **fopen()** дозволяє виконувати операції читання, запису чи переміщення маркера (вказівника на поточну позицію у файлі).

Операції з файлами

1. Відкриття файлу: **fopen(filename, mode)** зв'язує файл із файловою змінною. Режими:

- **"r"** – читання (файл має існувати).
- **"w"** – запис (створює новий файл або перезаписує існуючий).
- **"a"** – дописування (додає дані в кінець).
- **"rb"**, **"wb"**, **"ab"** – те ж саме для двійкових файлів.
- **"r+"**, **"w+"**, **"a+"** – читання і запис.

2. Закриття файлу: **fclose(fp)** завершує роботу з файлом, зберігаючи зміни.

3. Переміщення маркера: **fseek(fp, offset, whence)** дозволяє переміщатися у файлі:

- **SEEK_SET** – від початку.
- **SEEK_CUR** – від поточної позиції.
- **SEEK_END** – від кінця.

4. Перевірка кінця файлу: **feof(fp)** повертає ненульове значення, якщо досягнуто кінець.

Текстові файли

Текстові файли зручні для роботи із символьними даними. Основні функції [19]:

- **fgetc(fp)** – читає один символ.
- **fputc(c, fp)** – записує символ.

- **fgets(str, n, fp)** – читає рядок до n−1 символів.
- **fputs(str, fp)** – записує рядок.
- **fscanf(fp, format, ...)** і **fprintf(fp, format, ...)** – форматоване читання/запис.

Двійкові файли

Двійкові файли дозволяють працювати з даними будь-якого типу (наприклад, **int**, **float**). Основні функції:

- **fread(ptr, size, count, fp)** – читає **count** елементів розміром **size** у буфер **ptr**.
- **fwrite(ptr, size, count, fp)** – записує count елементів розміром **size** із буфера **ptr**.

6.2. Генерація випадкових чисел

Генерація випадкових чисел у C реалізується через функції **rand()** і **srand()** із бібліотеки **<stdlib.h>**. Вони корисні для тестування, моделювання або створення випадкових наборів даних.

- **rand()** – повертає псевдовипадкове число від 0 до **RAND_MAX** (зазвичай 32767).
- **srand(seed)** – встановлює початкове значення ("зерно") для генератора.

Без **srand()** послідовність завжди однакова. Зазвичай "зерно" ініціалізується поточним часом через **time(NULL)** із **<time.h>**.

6.3. Ініціалізація масивів константами

Масиви констант – це фіксовані набори даних, які задаються під час компіляції. Вони зручні для зберігання таблиць, шаблонів чи інших статичних даних. Наприклад:

```
int primes[] = {2, 3, 5, 7, 11};
```

6.4. Практичні приклади

Приклад 6.1. Запис чисел у текстовий файл

```
Записує числа від 1 до 10 у файл numbers.txt.
```

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("numbers.txt", "w");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    for (int i = 1; i <= 10; i++) {
        fprintf(fp, "%d ", i);
    }
    fclose(fp);
    printf("Дані записано у numbers.txt\n");
    return 0;
}
```

Приклад 6.2. Читання чисел із текстового файлу

```
Читає числа з numbers.txt і виводить їх.
```

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("numbers.txt", "r");
    int num;
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    printf("Числа з файлу: ");
    while (fscanf(fp, "%d", &num) != EOF) {
        printf("%d ", num);
    }
```

```
    printf("\n");
    fclose(fp);
    return 0;
}
```

Приклад 6.3. Запис двійкового файлу з масивом

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30, 40, 50};
    FILE *fp = fopen("data.bin", "wb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    fwrite(arr, sizeof(int), 5, fp);
    fclose(fp);
    printf("Масив записано у data.bin\n");
    return 0;
}
```

Приклад 6.4. Читання двійкового файлу

```
#include <stdio.h>

int main() {
    int arr[5];
    FILE *fp = fopen("data.bin", "rb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    fread(arr, sizeof(int), 5, fp);
    printf("Зчитаний масив: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
```

```
    fclose(fp);
    return 0;
}
```

Приклад 6.5. Тимчасовий файл для зберігання суми

```
#include <stdio.h>

int main() {
    FILE *tmp = tmpfile();
    if (tmp == NULL) {
        printf("Помилка створення тимчасового
файлу\n");
        return 1;
    }
    int sum = 0, num;
    printf("Введіть 5 чисел:\n");
    for (int i = 0; i < 5; i++) {
        scanf("%d", &num);
        sum += num;
        fwrite(&num, sizeof(int), 1, tmp);
    }
    fprintf(tmp, "Сума: %d", sum);
    rewind(tmp);
    printf("Дані з файлу: ");
    while (fscanf(tmp, "%d", &num) != EOF) {
        printf("%d ", num);
    }
    printf("\nСума: %d\n", sum);
    fclose(tmp);
    return 0;
}
```

Приклад 6.6. Інверсія символьного файлу

```
#include <stdio.h>

void InverseFile(char fname[]) {
    FILE *fp = fopen(fname, "r+b");
```

```
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return;
    }
    int size = 0;
    char c;
    while (fread(&c, sizeof(char), 1, fp)) {
        size++;
    }
    for (int i = 0; i < size / 2; i++) {
        char c1, c2;
        fseek(fp, i * sizeof(char), SEEK_SET);
        fread(&c1, sizeof(char), 1, fp);
        fseek(fp, (size - i - 1) * sizeof(char),
SEEK_SET);
        fread(&c2, sizeof(char), 1, fp);
        fseek(fp, i * sizeof(char), SEEK_SET);
        fwrite(&c2, sizeof(char), 1, fp);
        fseek(fp, (size - i - 1) * sizeof(char),
SEEK_SET);
        fwrite(&c1, sizeof(char), 1, fp);
    }
    fclose(fp);
}

int main() {
    InverseFile("chars.bin");
    printf("Файл інвертовано\n");
    return 0;
}
```

Приклад 6.7. Запис чисел Фібоначчі у двійковий файл

```
#include <stdio.h>

int Fib(int k) {
    int F = 1, F1 = 1, F2 = 1;
    for (int i = 2; i <= k; i++) {
        F = F1 + F2;
```

```
        F2 = F1;
        F1 = F;
    }
    return F;
}

void WriteFibToFile(char fname[], int n) {
    FILE *fp = fopen(fname, "wb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return;
    }
    int i = 0, F;
    do {
        F = Fib(i);
        if (F > n) break;
        fwrite(&F, sizeof(int), 1, fp);
        i++;
    } while (1);
    fclose(fp);
}

int main() {
    WriteFibToFile("fib.bin", 100);
    printf("Числа Фібоначчі записано у
fib.bin\n");
    return 0;
}
```

Приклад 6.8. Читання чисел Фібоначчі з двійкового файлу

```
#include <stdio.h>

void ReadFile(char fname[]) {
    FILE *fp = fopen(fname, "rb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return;
```

```
    }
    int F, b;
    printf("Числа Фібоначчі з файлу: ");
    while ((b = fread(&F, sizeof(int), 1, fp)) >
0) {
        printf("%d ", F);
    }
    printf("\n");
    fclose(fp);
}

int main() {
    ReadFile("fib.bin");
    return 0;
}
```

Приклад 6.9. Генерація випадкових чисел у масив

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

int main() {
    srand(time(NULL));
    int arr[10];
    printf("Випадкові числа: ");
    for (int i = 0; i < 10; i++) {
        arr[i] = rand() % 100; // Числа від 0 до
99
        printf("%d ", arr[i]);
    }
    printf("\n");
    return 0;
}
```

Приклад 6.10. Запис випадкових чисел у текстовий файл

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

```
#include <time.h>

int main() {
    srand(time(NULL));
    FILE *fp = fopen("random.txt", "w");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    for (int i = 0; i < 10; i++) {
        int num = rand() % 100;
        fprintf(fp, "%d\n", num);
    }
    fclose(fp);
    printf("Випадкові числа записано у
random.txt\n");
    return 0;
}
```

Приклад 6.11. Читання і підрахунок суми з текстового файлу

```
#include <stdio.h>

int main() {
    FILE *fp = fopen("random.txt", "r");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    int num, sum = 0;
    while (fscanf(fp, "%d", &num) != EOF) {
        sum += num;
    }
    printf("Сума чисел із файлу: %d\n", sum);
    fclose(fp);
    return 0;
}
```

Приклад 6.12. Ініціалізація масиву константами і запис у двійковий файл

```
#include <stdio.h>

int main() {
    int constants[] = {1, 4, 9, 16, 25}; //
Квадрати чисел
    FILE *fp = fopen("squares.bin", "wb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    fwrite(constants, sizeof(int), 5, fp);
    fclose(fp);
    printf("Константи записано у
squares.bin\n");
    return 0;
}
```

Приклад 6.13. Читання констант із двійкового файлу

```
#include <stdio.h>

int main() {
    int arr[5];
    FILE *fp = fopen("squares.bin", "rb");
    if (fp == NULL) {
        printf("Помилка відкриття файлу\n");
        return 1;
    }
    fread(arr, sizeof(int), 5, fp);
    printf("Константи з файлу: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    fclose(fp);
```

```
    return 0;
}
```

Приклад 6.14. Заміна символів у текстовому файлі

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

#define MAXLEN 256
#define MAXREPL 10

void replace(char s[], char a[], char c, char
r[]) {
    strcpy(r, "");
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] == c) {
            strcat(r, a);
        } else {
            int len = strlen(r);
            r[len] = s[i];
            r[len + 1] = '\0';
        }
    }
}

void replace_text(char fname[], char gname[],
char c, char a[]) {
    FILE *fp = fopen(fname, "r");
    FILE *gp = fopen(gname, "w");
    if (fp == NULL || gp == NULL) {
        printf("Помилка відкриття файлів\n");
        return;
    }
    char s[MAXLEN], s1[MAXLEN * MAXREPL];
    while (fgets(s, MAXLEN, fp)) {
        replace(s, a, c, s1);
        fputs(s1, gp);
    }
    fclose(fp);
```

```
    fclose(gp);
}

int main() {
    char fname[81] = "input.txt";
    char gname[81] = "output.txt";
    char c = 'a', a[] = "xyz";
    replace_text(fname, gname, c, a);
    printf("Заміна завершена\n");
    return 0;
}
```

Приклад 6.15. Видалення файлу

```
#include <stdio.h>

int main() {
    if (remove("temp.txt") == 0) {
        printf("Файл temp.txt видалено\n");
    } else {
        printf("Помилка видалення файлу\n");
    }
    return 0;
}
```

6.5. Порівняння з іншими мовами

Python: у Python робота з файлами спрощена через **`open()`** із контекстним менеджером (**`with`**), а генерація випадкових чисел – через модуль random. У C ці операції потребують ручного керування.

C++: C++ успадковує функції C, але додає об'єктно-орієнтовані потоки (**`fstream`**), які зручніші у використанні.

Java: у Java робота з файлами здійснюється через класи **`File`**, **`FileReader`**, **`FileWriter`**, із вбудованою обробкою винятків.

6.6. Рекомендації

Перевірка помилок: завжди перевіряйте результат `fopen()` на `NULL`.

Закриття файлів: не забувайте викликати `fclose()` для уникнення витоків ресурсів.

Тип даних: враховуйте розмір типів при роботі з двійковими файлами.

Ініціалізація генератора: `srand(time(NULL))` використовуйте лише раз у програмі.

Висновки

Ця тема розширила наші знання про введення даних у С, охопивши роботу з файлами (текстовими та двійковими), генерацію випадкових чисел і ініціалізацію масивів константами. Розширені приклади показали практичне застосування цих методів у різноманітних задачах. Наступна тема – двовимірні масиви – дозволить нам працювати з більш складними структурами даних.

Запитання та завдання для самоконтролю

1. Що таке файл у контексті програмування на мові С?
2. Які основні ознаки файлу ви можете назвати?
3. Чим відрізняється файл від змінної в програмі?
4. Вкажіть два основні типи файлів, які розрізняють у мові С.
5. Чому в С файли вважаються нетипізованими?
6. Яка бібліотека в С використовується для роботи з файлами?
7. Як оголошується файлова змінна в мові С?
8. Що таке маркер у файлі і як він працює?
9. Яка функція відкриває файл у С, і які режими вона підтримує?
10. Що означає режим `"rb"` при відкритті файлу?
11. Як закрити файл після роботи з ним?
12. Що робить функція `remove()` і що вона повертає при успішному виконанні?

13. Як створити і відкрити тимчасовий файл у програмі?
14. Яка різниця між функціями **`fread()`** і **`fwrite()`** для двійкових файлів?
15. Як працює функція **`fseek()`** і які значення може приймати аргумент **`whence`**?
16. Як перевірити, чи досягнуто кінець файлу?
17. Які функції використовуються для читання і запису символів у текстових файлах?
18. Як зчитати рядок із текстового файлу за допомогою **`fgets()`**?
19. У чому відмінність між **`fscanf()`** і **`scanf()`**?
20. Як записати форматовані дані у файл за допомогою **`fprintf()`**?
21. Що таке генерація випадкових чисел і для чого вона потрібна в програмах?
22. Які функції в C використовуються для генерації випадкових чисел?
23. Чому важливо ініціалізувати генератор випадкових чисел за допомогою **`srand()`**?
24. Як отримати випадкове число в заданому діапазоні (наприклад, від **`1`** до **`100`**)?
25. Як ініціалізувати масив константами під час оголошення?
26. Як записати масив чисел у двійковий файл?
27. Як зчитати дані з двійкового файлу в масив?
28. Як інвертувати вміст символьного файлу за допомогою функцій роботи з файлами?
29. Як замінити всі входження символу в текстовому файлі на рядок?
30. У чому різниця між роботою з файлами в C і в Python?

7. Двовимірні масиви. Базові алгоритми для обробки елементів двовимірних масивів

У програмуванні часто виникає потреба працювати з даними, які природно подаються у вигляді таблиць – наприклад, матриці чисел, шахові дошки чи пікселі зображень. У мові С такі структури реалізуються через двовимірні масиви – прямокутні таблиці однотипних елементів, що мають два виміри: рядки та стовпчики [20]. Визначимо двовимірний масив як прямокутну таблицю розміром m×n, яку можна розглядати як масив із m рядків, кожен із яких містить n елементів. Ця тема присвячена детальному вивченню двовимірних масивів: їхньому оголошенню, ініціалізації, принципам зберігання в пам'яті, введенню/виведенню та базовим алгоритмам обробки – від пошуку елементів до обчислення сум і аналізу за критеріями.

Двовимірні масиви є кроком до складніших структур даних, таких як тривимірні масиви чи динамічні масиви. Їхнє розуміння ключове для роботи з багатовимірними наборами даних.

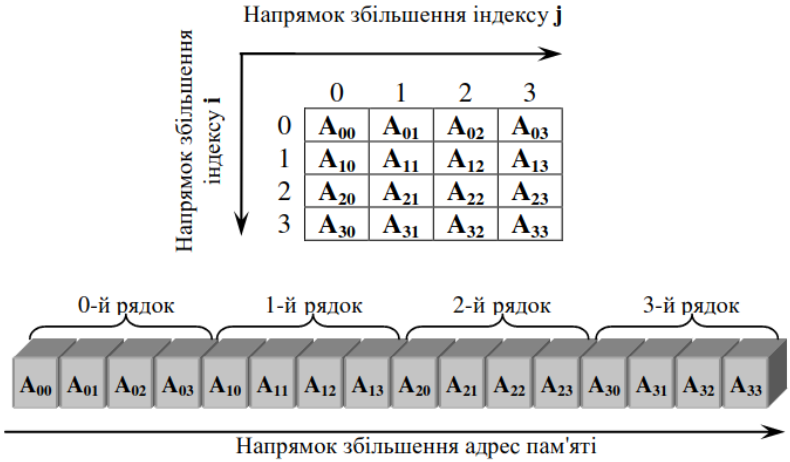
7.1. Поняття двовимірного масиву

Двовимірний масив – це структура даних, яка являє собою таблицю з **`m`** рядків і **`n`** стовпчиків, де кожен елемент має два індекси: перший позначає рядок **`(i)`**, другий – стовпчик **`(j)`**. У мові С двовимірний масив можна уявити як масив масивів: зовнішній масив містить **`m`** елементів, кожен із яких є одновимірним масивом із **`n`** елементів. Наприклад, масив **`int a2[2][3]`** – це таблиця з **`2`** рядками і **`3`** стовпчиками, що зберігає **`6`** цілих чисел.

У пам'яті елементи двовимірного масиву розташовуються послідовно "рядками": спочатку всі елементи першого рядка (**`a2[0][0]`**, **`a2[0][1]`**, **`a2[0][2]`**), потім другого (**`a2[1][0]`**, **`a2[1][1]`**, **`a2[1][2]`**). Це називається рядковим порядком зберігання (**`row-major order`**), що є стандартом для С і С++.

Наприклад, для **`int a2[2][3]`** пам’ять виділяється як один суцільний блок розміром **`2×3×sizeof(int)`** байтів (зазвичай 24 байти на 32-бітній системі).

Тривимірні та багатовимірні масиви узагальнюють цей принцип. Наприклад, **`int a3[5][2][3]`** – це 5 таблиць розміром 2×3, загалом 30 елементів.



7.2. Оголошення та ініціалізація двовимірних масивів

Оголошення

У С двовимірний масив оголошується так:

```
тип_даних ім’я[к-сть_рядків][к-сть_стовпчиків];
```

Приклади:

```
int matrix[3][4];       // Масив 3x4 цілих чисел
float table[5][2];      // Масив 5x2 дійсних
чисел
char grid[6][9];        // Масив 6x9 символів
```

- **Кількість рядків і стовпчиків** – це константи, визначені під час компіляції.

- **Індекси** починаються з `0`: для `matrix[3][4]` рядки – від `0` до `2`, стовпчики – від `0` до `3`.

Ініціалізація

Ініціалізація можлива кількома способами:

1. Повна ініціалізація з указанням усіх елементів:

```
int matrix[2][3] = {1, 2, 3, 4, 5, 6};
```

Елементи заповнюються послідовно: `matrix[0][0] = 1`, `matrix[0][1] = 2`, `...`, `matrix[1][2] = 6`.

2. Групування за рядками:

```
int matrix[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

Фігурні дужки підвищують читабельність, чітко вказуючи рядки.

3. Часткова ініціалізація:

```
int matrix[2][3] = {{1, 2}, {4}};
```

Неініціалізовані елементи автоматично заповнюються нулями: `matrix[0][2] = 0`, `matrix[1][1] = 0`, `matrix[1][2] = 0`.

4. "Безрозмірна" ініціалізація (опускається лише розмір рядків):

```
int matrix[][3] = {{1, 2, 3}, {4, 5, 6}};
```

Кількість рядків визначається автоматично (тут 2), але кількість стовпчиків має бути вказана.

Важливо: опускати розмір стовпчиків не можна, інакше компілятор видасть помилку.

Особливості зберігання в пам’яті

Елементи двовимірного масиву зберігаються в пам’яті лінійно, у порядку зростання індексів. Наприклад:

```
int m[2][3] = {{1, 2, 3}, {4, 5, 6}};
```

У пам'яті: `[1, 2, 3, 4, 5, 6]`. Це означає, що доступ до елементів швидкий, але розмір масиву фіксований під час компіляції.

7.3. Уведення та виведення двовимірних масивів

Уведення даних

Для введення елементів двовимірного масиву використовуються вкладені цикли:

```
#include <stdio.h>
#define N 3
#define M 4

int main() {
    int matrix[N][M];
    printf("Введіть %d x %d елементів:\n", N,
M);
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            scanf("%d", &matrix[i][j]);
        }
    }
    return 0;
}
```

Зовнішній цикл проходить по рядках, внутрішній – по стовпчиках.

Виведення даних

Для виведення масиву у вигляді таблиці:

```
#include <stdio.h>
#define N 3
#define M 4

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

```
    printf("Масив:\n");
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
    return 0;
}
```

```
Виведення:
Масив:
1    2    3    4
5    6    7    8
9    10   11   12
```

7.4. Базові алгоритми обробки двовимірних масивів

7.4.1. Пошук елементів за критерієм

Приклад 7.1. Заміна від'ємних елементів на нуль

```
void replaceNegatives(int array[][M], int rows)
{
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < M; j++) {
            if (array[i][j] < 0) {
                array[i][j] = 0;
            }
        }
    }
}
```

Приклад 7.2. Пошук першого додатного елемента

```
#include <stdio.h>
#define N 3
#define M 4
```

```
int main() {
    int matrix[N][M] = {{0, -2, 3, -4}, {-5, 6,
-7, 8}, {-9, -10, 11, -12}};
    int found = 0;
    for (int i = 0; i < N && !found; i++) {
        for (int j = 0; j < M && !found; j++) {
            if (matrix[i][j] > 0) {
                printf("Перший додатний: %d,
індекси [%d][%d]\n", matrix[i][j], i, j);
                found = 1;
            }
        }
    }
    return 0;
}
```

```
Виведення:
Перший додатний: 3, індекси [0][2]
```

7.4.2. Обчислення суми елементів

Приклад 7.3. Сума всіх елементів

```
int matrixSum(int array[][M], int rows) {
    int sum = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < M; j++) {
            sum += array[i][j];
        }
    }
    return sum;
}
```

Приклад 7.4. Сума елементів головного діагоналі

```
#include <stdio.h>
#define N 3

int diagonalSum(int array[][N]) {
```

```
    int sum = 0;
    for (int i = 0; i < N; i++) {
        sum += array[i][i];
    }
    return sum;
}

int main() {
    int matrix[N][N] = {{1, 2, 3}, {4, 5, 6},
{7, 8, 9}};
    printf("Сума діагоналі: %d\n",
diagonalSum(matrix));
    return 0;
}
```

```
Виведення:
Сума діагоналі: 15 (1 + 5 + 9)
```

7.4.3. Підрахунок елементів за критерієм

Приклад 7.5. Кількість парних чисел у стовпцях

```
void countEvenNumbers(int array[][M], int rows)
{
    for (int j = 0; j < M; j++) {
        int count = 0;
        for (int i = 0; i < rows; i++) {
            if (array[i][j] % 2 == 0) {
                count++;
            }
        }
        printf("Стовпець %d: %d парних\n", j +
1, count);
    }
}
```

Приклад 7.6. Кількість нулів у матриці

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

```
#define N 3
#define M 4

int countZeros(int array[][M]) {
    int count = 0;
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < M; j++) {
            if (array[i][j] == 0) {
                count++;
            }
        }
    }
    return count;
}

int main() {
    int matrix[N][M] = {{0, 1, 0, 2},
                        {3, 0, 4, 5},
                        {6, 7, 0, 8}};
    printf("Кількість нулів: %d\n",
           countZeros(matrix));
    return 0;
}
```

```
Виведення:
Кількість нулів: 3
```

7.4.4. Пошук максимального/мінімального елементів

Приклад 7.7. Максимальний елемент та його індекси

```
void maxElement(int array[][M], int rows) {
    int max = array[0][0], max_i = 0, max_j = 0;
    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < M; j++) {
            if (array[i][j] > max) {
                max = array[i][j];
                max_i = i;
```

```
                max_j = j;
            }
        }
    }
    printf("Максимум: %d, індекси [%d][%d]\n",
           max, max_i, max_j);
}
```

Приклад 7.8. Мінімум у кожному рядку

```
void minElement(int array[][M], int rows) {
    for (int i = 0; i < rows; i++) {
        int min = array[i][0], min_j = 0;
        for (int j = 1; j < M; j++) {
            if (array[i][j] < min) {
                min = array[i][j];
                min_j = j;
            }
        }
        printf("Рядок %d: мінімум %d, стовпець
%d\n",
               i + 1, min, min_j + 1);
    }
}
```

Приклад 7.9. Максимум у кожному стовпчику

```
#include <stdio.h>
#define N 3
#define M 4

void maxInColumns(int array[][M]) {
    for (int j = 0; j < M; j++) {
        int max = array[0][j];
        for (int i = 1; i < N; i++) {
            if (array[i][j] > max) {
                max = array[i][j];
            }
        }
```

```
        printf("Стовпець %d: максимум %d\n",
               j + 1, max);
    }
}

int main() {
    int matrix[N][M] = {{1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}};
    maxInColumns(matrix);
    return 0;
}
```

```
Виведення:
Стовпець 1: максимум 9
Стовпець 2: максимум 10
Стовпець 3: максимум 11
Стовпець 4: максимум 12
```

7.4.5. Формування одновимірних масивів

Приклад 7.10. Сума елементів кожного рядка

```
void rowSums(int array[][M], int mas[], int
rows) {
    for (int i = 0; i < rows; i++) {
        int sum = 0;
        for (int j = 0; j < M; j++) {
            sum += array[i][j];
        }
        mas[i] = sum;
        printf("Рядок %d: сума %d\n", i + 1,
mas[i]);
    }
}
```

Приклад 7.11. Середнє арифметичне кожного стовпчика

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

```
#define N 3
#define M 4

void columnMeans(int array[][M], float means[])
{
    for (int j = 0; j < M; j++) {
        int sum = 0;
        for (int i = 0; i < N; i++) {
            sum += array[i][j];
        }
        means[j] = (float)sum / N;
        printf("Стовпець %d: середнє %.2f\n",
               j + 1, means[j]);
    }
}

int main() {
    int matrix[N][M] = {{1, 2, 3, 4},
                        {5, 6, 7, 8},
                        {9, 10, 11, 12}};
    float means[M];
    columnMeans(matrix, means);
    return 0;
}
```

```
Виведення:
Стовпець 1: середнє 5.00
Стовпець 2: середнє 6.00
Стовпець 3: середнє 7.00
Стовпець 4: середнє 8.00
```

7.4.6. Перетворення матриці

Приклад 7.12. Транспонування матриці

```
#include <stdio.h>
#define N 3
```

```
void transpose(int matrix[][N], int result[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[j][i] = matrix[i][j];
        }
    }
}

void printMatrix(int matrix[][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            printf("%d\t", matrix[i][j]);
        }
        printf("\n");
    }
}

int main() {
    int matrix[N][N] = {{1, 2, 3},
                        {4, 5, 6},
                        {7, 8, 9}};
    int result[N][N];
    printf("Початкова матриця:\n");
    printMatrix(matrix);
    transpose(matrix, result);
    printf("Транспонована матриця:\n");
    printMatrix(result);
    return 0;
}
```

```
Виведення:
Початкова матриця:
1    2    3
4    5    6
7    8    9
Транспонована матриця:
1    4    7
2    5    8
```

```
3    6    9
```

Приклад 7.13. Обертання матриці на 90° за годинниковою стрілкою

```
#include <stdio.h>
#define N 3

void rotate90(int matrix[][N], int result[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            result[j][N - 1 - i] = matrix[i][j];
        }
    }
}

int main() {
    int matrix[N][N] = {{1, 2, 3}, {4, 5, 6},
{7, 8, 9}};
    int result[N][N];
    printf("Початкова матриця:\n");
    printMatrix(matrix);
    rotate90(matrix, result);
    printf("Матриця після обертання на 90°:\n");
    printMatrix(result);
    return 0;
}
```

```
Виведення:
Початкова матриця:
1    2    3
4    5    6
7    8    9
Матриця після обертання на 90°:
7    4    1
8    5    2
9    6    3
```

7.4.7. Аналіз симетрії

Приклад 7.14. Перевірка симетрії відносно головної діагоналі

```
#include <stdio.h>
#define N 3

int isSymmetric(int matrix[][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < i; j++) {
            if (matrix[i][j] != matrix[j][i]) {
                return 0;
            }
        }
    }
    return 1;
}

int main() {
    int matrix[N][N] = {{1, 2, 3},
                        {2, 5, 6},
                        {3, 6, 9}};
    if (isSymmetric(matrix)) {
        printf("Матриця симетрична\n");
    } else {
        printf("Матриця не симетрична\n");
    }
    return 0;
}
```

```
Виведення:
Матриця симетрична
```

7.4.8. Робота з підматрицями

Приклад 7.15. Витягнення підматриці

```
#include <stdio.h>
#define N 4
#define M 5
```

```
void extractSubmatrix(int matrix[][M], int sub[]
[2], int start_i, int start_j) {
    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 2; j++) {
            sub[i][j] = matrix[start_i + i]
[start_j + j];
        }
    }
}

int main() {
    int matrix[N][M] = {{1, 2, 3, 4, 5},
                        {6, 7, 8, 9, 10},
                        {11, 12, 13, 14, 15},
                        {16, 17, 18, 19, 20}};
    int sub[2][2];
    extractSubmatrix(matrix, sub, 1, 2);
    printf("Підматриця 2x2:\n");
    printMatrix(sub, 2, 2);
    return 0;
}
```

```
Виведення:
Підматриця 2x2:
8    9
13   14
```

7.4.9. Генерація спеціальних матриць

Приклад 7.16. Генерація одиничної матриці

```
#include <stdio.h>
#define N 3

void identityMatrix(int matrix[][N]) {
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            matrix[i][j] = (i == j) ? 1 : 0;
```

```
        }
    }
}

int main() {
    int matrix[N][N];
    identityMatrix(matrix);
    printf("Одинична матриця:\n");
    printMatrix(matrix);
    return 0;
}
```

```
Виведення:
Одинична матриця:
1    0    0
0    1    0
0    0    1
```

7.5. Порівняння з іншими мовами

Python: у Python двовимірні масиви реалізуються як списки списків (`[[1, 2], [3, 4]]`), що дозволяє динамічний розмір, але ускладнює доступ до пам'яті. У C масиви статичні й швидші.

C++: C++ підтримує масиви C, але додає **`std::vector`** для динамічних двовимірних структур.

Java: у Java двовимірні масиви (**`int[][]`**) можуть бути нерівномірними, на відміну від C, де розміри фіксовані.

7.6. Рекомендації

Перевірка меж: уникайте виходу за межі масиву, щоб запобігти невизначеній поведінці.

Ініціалізація: завжди ініціалізуйте масив, якщо значення не вводяться користувачем.

Оптимізація: для великих матриць використовуйте локальні змінні для індексів і кешуйте результати обчислень.

Читабельність: використовуйте вкладені дужки для ініціалізації та коментарі для складних алгоритмів.

Висновки

Двовимірні масиви є потужним інструментом для роботи з табличними даними в С. У цьому розділі розглянули їх оголошення, ініціалізацію, принципи зберігання, введення/виведення та базові алгоритми – від пошуку елементів до перетворень і аналізу. Приклади охопили широкий спектр задач, демонструючи практичне застосування. Наступна тема – рядки – розширить можливості для роботи з текстовими даними.

Запитання та завдання для самоконтролю

1. Що таке двовимірний масив у мові С?
2. Як двовимірний масив можна уявити як масив масивів?
3. Який порядок зберігання елементів двовимірного масиву в пам'яті С?
4. Як оголосити двовимірний масив розміром `4×5` цілих чисел?
5. Який індекс має перший елемент двовимірного масиву?
6. Як ініціалізувати двовимірний масив `2×3` конкретними значеннями?
7. Що станеться, якщо при ініціалізації двовимірного масиву вказати менше елементів, ніж його розмір?
8. Чи можна опустити розмір стовпчиків при "безрозмірній" ініціалізації? Чому?
9. Як групувати елементи за рядками при ініціалізації двовимірного масиву?
10. Як ввести елементи двовимірного масиву `3×4` з клавіатури?
11. Як вивести двовимірний масив у вигляді таблиці?
12. Як обійти двовимірний масив по стовпчиках за допомогою вкладених циклів?
13. Як знайти суму всіх елементів двовимірного масиву?
14. Як підрахувати кількість від'ємних елементів у двовимірному масиві?
15. Як замінити всі додатні елементи двовимірного масиву на 1?

16. Як знайти максимальний елемент у двовимірному масиві та його індекси?
17. Як визначити мінімальний елемент у кожному рядку матриці?
18. Як обчислити суму елементів головної діагоналі квадратної матриці?
19. Як підрахувати кількість парних чисел у кожному стовпчику?
20. Як утворити одновимірний масив із середніх арифметичних значень рядків двовимірного масиву?
21. Як транспонувати квадратну матрицю **`N×N`**?
22. Як перевірити, чи матриця симетрична відносно головної діагоналі?
23. Як обернути матрицю на 90° за годинниковою стрілкою?
24. Як витягти підматрицю розміром **`2×2`** із заданого місця у двовимірному масиві?
25. Як згенерувати одиничну матрицю розміром **`3×3`**?
26. Що станеться, якщо звернутися до елемента поза межами двовимірного масиву?
27. Як оптимізувати обхід двовимірного масиву для великих розмірів?
28. У чому різниця між двовимірними масивами в C і списками списків у Python?
29. Як ініціалізувати двовимірний масив нулями без явного переліку значень?
30. Як знайти кількість елементів, що дорівнюють заданому значенню, у двовимірному масиві?

8. Рядки і символьні масиви. Стандартні функції для роботи з рядками

Робота з текстовими даними є невід'ємною частиною програмування, чи то обробка введення користувача, парсинг файлів, чи створення інтерфейсів. У мові С рядок – це послідовність символів, яка обмежується подвійними лапками (наприклад, "string"). Однак С не має вбудованого типу даних "рядок" – замість цього рядки реалізуються через символьні масиви типу char. Ця тема присвячена детальному розгляду рядків як символьних масивів, їхньому оголошенню, зберіганню в пам'яті, а також стандартним функціям бібліотеки <string.h> для роботи з ними.

Рядки в С є основою для текстової обробки, і їхнє розуміння відкриває шлях до складніших алгоритмів, розглянутих у наступній темі.

8.1. Поняття рядка в мові С

Рядок – це послідовність символів, яка завершується спеціальним нульовим символом **`\0`** (**`ASCII`**-код **`0`**), що позначає кінець рядка. Рядок визначається як "довільна послідовність символів", відмежована подвійними лапками [15]. Наприклад:

```
char s[] = "hello";
// Рядок "hello" із завершенням '\0'
```

- У пам'яті: **`'h', 'e', 'l', 'l', 'o', '\0'`** (6 байтів).

Рядки в С реалізуються через масиви типу **`char`**, де:

- Кожен елемент масиву – це символ (1 байт у стандартному кодуванні ASCII).
- Нульовий термінатор **`\0`** додається автоматично при ініціалізації рядковим літералом.

Оголошення символьних масивів

1. Із явним розміром:

```
char s[10] = "hello";
```

```
// Масив на 10 символів, заповнений "hello\0" і нулями
```

- Невикористані елементи заповнюються `\0`.

2. Без указання розміру:

```
char s[] = "hello";
// Розмір автоматично = 6 (5 символів + '\0')
```

3. Як указівник:

```
char *s = "hello";
// Указівник на константний рядок у пам'яті
```

Важливо: константні рядки не можна модифікувати через указівник.

Зберігання в пам'яті

Рядок у С – це суцільний блок пам'яті. Наприклад:

- Для `char s[] = "abc"` пам'ять виглядає як: `[97, 98, 99, 0]` (ASCII-коди).
- Доступ до символів: `s[0] = 'a'`, `s[1] = 'b'`, `s[2] = 'c'`, `s[3] = '\0'`.

8.2. Основні операції з рядками

Звернення до символів

- Доступ до і-го символу: `s[i]`.
- Зміна символу:

```
char s[] = "abc";
s[1] = 'x'; // s стає "axc"
```

Отримання підрядка

- Операція `&s[i]` повертає вказівник на підрядок із `i`-го символу до кінця:

```
char s[] = "abcdef";
char *sub = &s[2]; // sub вказує на "cdef"
```

Через `<stdio.h>`:

- **`gets(s)`** – читає рядок із **`stdin`** до символу нового рядка (застаріла, небезпечна).
- **`fgets(s, n, stdin)`** – безпечніша альтернатива, читає до **`n-1`** символів.
- **`puts(s)`** – виводить рядок із додаванням **`\n`**.

```
#include <stdio.h>
#define N 20

int main() {
    char s[N];
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
    puts("Ваш рядок:");
    puts(s);
    return 0;
}
```

8.3. Стандартні функції бібліотеки <string.h>

Бібліотека **`<string.h>`** надає набір функцій для роботи з рядками. Ось основні з них із розширеними поясненнями:

1. `strlen(s)` – повертає довжину рядка **`s`** (без урахування **`\0`**).

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

int main() {
    char s[] = "hello";
    printf("Довжина: %zu\n", strlen(s));
    // Виведе: 5
    return 0;
}
```

2. `strcmp(s1, s2)` – порівнює рядки лексикографічно:

- **`< 0: s1<s2`**.

- `= 0: s1==s2`.
- `> 0: s1>s2`.

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

int main() {
    char s1[] = "apple";
    char s2[] = "banana";
    int result = strcmp(s1, s2);
    printf("Результат: %d\n", result); // < 0,
бо "apple" < "banana"
    return 0;
}
```

3. `strcpy(s, s1)` – копіює рядок `s1` у `s` разом із `\0`.

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

int main() {
    char s[20];
    char s1[] = "hello";
    strcpy(s, s1);
    printf("Скопійований рядок: %s\n", s); //
"hello"
    return 0;
}
```

4. `strncpy(s, s1, n)` – копіює перші `n` символів із `s1` у `s`, але не гарантує `\0`.

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

int main() {
    char s[20] = "xxxxxxxxxx";
    char s1[] = "hello";
    strncpy(s, s1, 3);
    s[3] = '\0'; // Додаємо термінатор вручну
```

```
    printf("Частково скопійований рядок: %s\n",
s); // "hel"
    return 0;
}
```

Зауваження: без явного додавання `\0` можливі помилки.

5. `memmove(s, s1, n)` – копіює `n` байтів із `s1` у `s`, безпечна для перекриття областей пам'яті.

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

int main() {
    char s[] = "abcdef";
    memmove(s + 1, s, 3);
    printf("Результат: %s\n", s); // "aaabcf"
    return 0;
}
```

6. `strcat(s, s1)` – дописує `s1` до кінця `s`.

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

int main() {
    char s[20] = "hello";
    char s1[] = " world";
    strcat(s, s1);
    printf("Об'єднаний рядок: %s\n", s); //
"hello world"
    return 0;
}
```

7. `strncat(s, s1, n)` – дописує перші `n` символів із `s1` до `s`, додаючи `\0`.

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

int main() {
    char s[20] = "hello";
```

```
    char s1[] = " world";
    strncat(s, s1, 3);
    printf("Частково об'єднаний рядок: %s\n",
s); // "hello wo"
    return 0;
}
```

8. **`strchr(s, c)`** – повертає вказівник на перше входження символу **`c`** у **`s`** або **`NULL`**.

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

int main() {
    char s[] = "hello";
    char *p = strchr(s, 'l');
    if (p) {
        printf("Знайдено 'l' за адресою: %ld\n",
p - s); // 2
    }
    return 0;
}
```

9. **`strrchr(s, c)`** – повертає вказівник на останнє входження **`c`** у **`s`**.

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

int main() {
    char s[] = "hello";
    char *p = strrchr(s, 'l');
    if (p) {
        printf("Останнє 'l' за адресою: %ld\n",
p - s); // 3
    }
    return 0;
}
```

10. **`strstr(s1, s2)`** – шукає перше входження підрядка **`s2`** у **`s1`**.

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

int main() {
    char s1[] = "hello world";
    char s2[] = "wor";
    char *p = strstr(s1, s2);
    if (p) {
        printf("Підрядок знайдено: %s\n", p); //
"world"
    }
    return 0;
}
```

8.4. Практичні приклади

Приклад 8.1. Довжина рядка

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

int main() {
    char s[] = "Programming";
    printf("Довжина рядка: %zu\n", strlen(s));
    return 0;
}
```

```
Виведення:
Довжина рядка: 11
```

Приклад 8.2. Порівняння рядків

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

int main() {
    char s1[] = "cat";
    char s2[] = "dog";
```

```
    if (strcmp(s1, s2) < 0) {
        printf("%s < %s\n", s1, s2);
    }
    return 0;
}
```

```
Виведення:
cat < dog
```

Приклад 8.3. Копіювання рядка

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

int main() {
    char src[] = "Source";
    char dest[20];
    strcpy(dest, src);
    printf("Скопійовано: %s\n", dest);
    return 0;
}
```

Приклад 8.4. Часткове копіювання

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

int main() {
    char src[] = "Example";
    char dest[20] = "Initial";
    strncpy(dest, src, 3);
    dest[3] = '\0';
    printf("Результат: %s\n", dest);
    return 0;
}
```

```
Виведення:
Результат: Exa
```

Приклад 8.5. Дописування символу

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

int main() {
    char s[20] = "abc";
    char c = 'D';
    int n = strlen(s);
    s[n] = c;
    s[n + 1] = '\0';
    printf("Новий рядок: %s\n", s);
    return 0;
}
```

```
Виведення:
Новий рядок: abcD
```

Приклад 8.6. Пошук символу

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

int main() {
    char s[] = "banana";
    char *p = strchr(s, 'n');
    if (p) {
        printf("Перший 'n' на позиції: %ld\n", p
- s);
    }
    return 0;
}
```

```
Виведення:
Перший 'n' на позиції: 2
```

Приклад 8.7. Останнє входження символу

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

int main() {
    char s[] = "banana";
    char *p = strrchr(s, 'a');
    if (p) {
        printf("Останній 'a' на позиції: %ld\n",
p - s);
    }
    return 0;
}
```

```
Виведення:
Останній 'a' на позиції: 5
```

Приклад 8.8. Пошук підрядка

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

int main() {
    char s[] = "This is a test";
    char *p = strstr(s, "is");
    if (p) {
        printf("Підрядок: %s\n", p);
    }
    return 0;
}
```

```
Виведення:
Підрядок: is a test
```

Приклад 8.9. Об’єднання рядків

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

```c
#include <stdio.h>

int main() {
    char s[20] = "Hello";
    char s1[] = " C!";
    strcat(s, s1);
    printf("Результат: %s\n", s);
    return 0;
}
```

```
Виведення:
Результат: Hello C!
```

Приклад 8.10. Введення рядка

```c
#include <stdio.h>
#define N 50

int main() {
    char s[N];
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
    printf("Ви ввели: %s", s);
    return 0;
}
```

8.5. Порівняння з іншими мовами

Python: рядки – це об'єкти типу **`str`**, незмінні, із вбудованими методами (**`len()`**, **`find()`**).

C++: додає клас **`std::string`**, який спрощує роботу порівняно з масивами **`char`**.

Java: рядки – об'єкти класу **`String`**, незмінні, із широким набором методів.

8.6. Рекомендації

Перевірка розміру: завжди враховуйте місце для **`\0`** у масиві.

Безпечність: уникайте **`gets()`**, використовуйте **`fgets()`**.

Термінатор: переконайтеся, що рядок завершується **`\0`** після модифікацій.

Висновки

Рядки в C – це символьні масиви, які потребують уважного керування пам'яттю та завершенням. Бібліотека **`<string.h>`** забезпечує потужний набір функцій для їхньої обробки. Розширені приклади показали базові операції, готуючи нас до алгоритмів у наступній темі.

Запитання та завдання для самоконтролю

1. Що таке рядок у мові C?
2. Чим рядок відрізняється від ідентифікатора чи ключового слова в коді?
3. Як рядки реалізуються в мові C?
4. Що таке нульовий термінатор і яку роль він відіграє в рядках?
5. Як оголосити символьний масив для зберігання рядка з максимум 50 символами?
6. Чим відрізняється оголошення **`char s[] = "hello"`** від **`char *s = "hello"`**?
7. Як отримати доступ до **`i`**-го символу рядка?
8. Що повертає операція **`&s[i]`** для рядка **`s`**?
9. Як змінити другий символ рядка на **`'X'`**?
10. Яка функція повертає довжину рядка без урахування нульового термінатора?
11. Поясніть як працює функція **`strcmp()`** і які значення вона може повертати.
12. Як скопіювати рядок **`s1`** у рядок **`s`** за допомогою стандартної функції?
13. У чому різниця між **`strcpy()`** і **`strncpy()`**?
14. Чому **`strncpy()`** може призвести до проблем із нульовим термінатором?
15. Як працює функція **`memmove()`** при копіюванні частини рядка?

16. Як додати рядок **s1** до кінця рядка **s** за допомогою стандартної функції?
17. Що робить функція **strncat()** і чим вона відрізняється від **strcat()**?
18. Як знайти перше входження символу в рядку за допомогою **<string.h>**?
19. Як знайти останнє входження символу в рядку?
20. Як знайти підрядок у рядку за допомогою стандартної функції?
21. Як безпечно ввести рядок із клавіатури, уникаючи переповнення буфера?
22. Чим небезпечна функція **gets()** і яка її альтернатива?
23. Як вивести рядок на екран із новим рядком за допомогою **<stdio.h>**?
24. Як вручну додати символ до кінця рядка, якщо стандартні функції не використовуються?
25. Що станеться, якщо рядок не завершується символом **\0**?
26. Як визначити скільки пам'яті займає рядок **"abc"**?
27. У чому різниця між рядками в C і об'єктами типу **string** у C++?
28. Як уникнути помилок при копіюванні рядка, якщо розмір цільового масиву менший за джерело?
29. Як працює порівняння рядків у лексикографічному порядку?
30. Як об'єднати два рядки із використанням циклу без функції **strcat()**?

9.Найпростіші алгоритми роботи з рядками

Обробка тексту – це одна з ключових задач програмування, яка включає пошук, заміну, інверсію та інші операції з рядками. У мові С, де рядки реалізовані як символьні масиви, ці задачі вимагають ручного маніпулювання елементами. Ця тема присвячена найпростішим алгоритмам роботи з рядками: пошуку символів і підрядків, заміни елементів, інверсії та побудови нових рядків.

9.1. Основи алгоритмів із рядками

Пошук символу

- **Алгоритм:** лінійний перебір символів рядка.
- **Складність:** `O(n)`, де `n` – довжина рядка.

Заміна символів

- **Алгоритм:** перебір з умовною заміною.
- **Складність:** `O(n)`.

Інверсія рядка

- **Алгоритм:** перестановка символів або побудова нового рядка.
- **Складність:** `O(n)`.

9.2. Практичні приклади

Приклад 9.1. Перевірка входження символу

```
#include <stdio.h>
#include <string.h>
#define N 255

int main() {
    char s[N];
    char c;
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
```

```
    printf("Введіть символ: ");
    scanf(" %c", &c);
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] == c) {
            printf("Символ '%c' знайдено на
позиції %d\n", c, i);
            return 0;
        }
    }
    printf("Символ '%c' не знайдено\n", c);
    return 0;
}
```

Приклад 9.2. Виведення великих літер

```
#include <stdio.h>
#include <string.h>
#define N 255

int main() {
    char s[N];
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
    printf("Великі літери: ");
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            printf("%c ", s[i]);
        }
    }
    printf("\n");
    return 0;
}
```

Приклад 9.3. Заміна '0' на '1' і навпаки

```
#include <stdio.h>
#include <string.h>
#define N 255
```

```
int main() {
    char s[N];
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] == '0') s[i] = '1';
        else if (s[i] == '1') s[i] = '0';
    }
    printf("Змінений рядок: %s", s);
    return 0;
}
```

Приклад 9.4. Заміна крапки на три крапки

```
#include <stdio.h>
#include <string.h>
#define N 255

int main() {
    char s[N], s1[N * 3] = "";
    printf("Введіть рядок: ");
    fgets(s, N, stdin);
    for (int i = 0; i < strlen(s); i++) {
        char temp[2] = {s[i], '\0'};
        strcat(s1, temp);
        if (s[i] == '.') {
            strcat(s1, "..");
        }
    }
    printf("Результат: %s", s1);
    return 0;
}
```

Приклад 9.5. Інверсія рядка (спосіб 1)

```
#include <stdio.h>
#include <string.h>
#define N 255
```

```
void Inverse(char s[]) {
    for (int i = 0; i < strlen(s) / 2; i++) {
        char temp = s[i];
        s[i] = s[strlen(s) - 1 - i];
        s[strlen(s) - 1 - i] = temp;
    }
}

int main() {
    char s[N] = "hello";
    Inverse(s);
    printf("Інверсія: %s\n", s);
    return 0;
}
```

```
Виведення:
Інверсія: olleh
```

Приклад 9.6. Інверсія через допоміжний рядок (спосіб 2)

```
#include <stdio.h>
#include <string.h>
#define N 255

void Inverse(char s[]) {
    char s1[N] = "";
    for (int i = strlen(s) - 1; i >= 0; i--) {
        char temp[2] = {s[i], '\0'};
        strcat(s1, temp);
    }
    strcpy(s, s1);
}

int main() {
    char s[N] = "world";
    Inverse(s);
    printf("Інверсія: %s\n", s);
    return 0;
```

```
}
```

Приклад 9.7. Рекурсивна інверсія (спосіб 3)

```
#include <stdio.h>
#include <string.h>
#define N 255

void Inverse(char s[], int l, int r) {
    if (l < r) {
        char temp = s[l];
        s[l] = s[r];
        s[r] = temp;
        Inverse(s, l + 1, r - 1);
    }
}

void Inv(char s[]) {
    Inverse(s, 0, strlen(s) - 1);
}

int main() {
    char s[N] = "test";
    Inv(s);
    printf("Інверсія: %s\n", s);
    return 0;
}
```

```
Виведення:
Інверсія: tset.
```

Приклад 9.8. Підрахунок голосних

```
#include <stdio.h>
#include <string.h>
#define N 255

int countVowels(char s[]) {
```

```
    int count = 0;
    for (int i = 0; i < strlen(s); i++) {
        char c = s[i];
        if (c == 'a' || c == 'e' || c == 'i' ||
c == 'o' || c == 'u' ||
            c == 'A' || c == 'E' || c == 'I' ||
c == 'O' || c == 'U') {
            count++;
        }
    }
    return count;
}

int main() {
    char s[N] = "Hello World";
    printf("Кількість голосних: %d\n",
countVowels(s));
    return 0;
}
```

```
Виведення:
Кількість голосних: 3
```

Приклад 9.9. Перетворення на верхній регістр

```
#include <stdio.h>
#include <string.h>
#define N 255

void toUpper(char s[]) {
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] >= 'a' && s[i] <= 'z') {
            s[i] -= 32; // Від 'a' до 'A' у
ASCII
        }
    }
}

int main() {
```

```
    char s[N] = "hello";
    toUpper(s);
    printf("Верхній регістр: %s\n", s);
    return 0;
}
```

```
Виведення:
Верхній регістр: HELLO
```

Приклад 9.10. Видалення пробілів

```
#include <stdio.h>
#include <string.h>
#define N 255

void removeSpaces(char s[]) {
    int j = 0;
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] != ' ') {
            s[j++] = s[i];
        }
    }
    s[j] = '\0';
}

int main() {
    char s[N] = "Hello World";
    removeSpaces(s);
    printf("Без пробілів: %s\n", s);
    return 0;
}
```

```
Виведення:
Без пробілів: HelloWorld
```

9.3. Рекомендації

Розмір буфера: переконайтеся, що допоміжні масиви достатньо великі для результатів.

Стабільність: завжди додавайте `\0` після модифікацій.

Ефективність: уникайте повторних викликів `strlen()` у циклах.

Висновки

Найпростіші алгоритми роботи з рядками у С включають пошук, заміну та інверсію, демонструючи гнучкість символьних масивів. Розглянуті приклади закладають основу для складніших текстових операцій у подальших темах.

Запитання та завдання для самоконтролю

1. Що таке алгоритм роботи з рядками у контексті мови С?
2. Яка часова складність лінійного пошуку символу в рядку?
3. Як перевірити, чи входить заданий символ до рядка?
4. Як вивести всі великі латинські літери з рядка?
5. Як замінити всі символи `'0'` на `'1'` і навпаки в рядку?
6. Як замінити кожну крапку в рядку на три крапки?
7. Що таке інверсія рядка і як її реалізувати?
8. Як інвертувати рядок, використовуючи заміну символів на місці?
9. Як інвертувати рядок за допомогою допоміжного масиву?
10. Як реалізувати рекурсивну інверсію рядка без додаткових параметрів?
11. Як працює рекурсивна інверсія рядка із зазначенням меж?
12. Як підрахувати кількість голосних літер у рядку?
13. Поясніть як перетворити всі символи рядка на верхній регістр.
14. Як видалити всі пробіли з рядка?
15. Як знайти кількість входжень заданого символу в рядок?
16. Як замінити всі малі літери на великі без використання стандартних функцій?
17. Як перевірити, чи є рядок паліндромом?

18. Як обчислити кількість слів у рядку, розділених пробілами?
19. Як обрізати рядок до першого пробілу?
20. Як замінити всі цифри в рядку на символ `'*'`?
21. Як знайти найдовше слово в рядку?
22. Як перевірити, чи складається рядок лише з цифр?
23. Як видалити всі голосні з рядка?
24. Як додати символ на початок рядка?
25. Як розбити рядок на два за певним символом?
26. Як порівняти два рядки без використання **`strcmp()`**?
27. Як визначити, чи є рядок порожнім (лише **`\0`**)?
28. Як замінити кожен другий символ рядка на пробіл?
29. Як обчислити суму ASCII-кодів символів рядка?
30. Як знайти позицію першого входження підрядка без **`strstr()`**?

10. Вказівники. Операції з вказівниками. Практичне застосування вказівників

Вказівники є одним із найпотужніших і водночас складних інструментів у мові С. Вони дозволяють програмісту працювати безпосередньо з пам'яттю, забезпечуючи гнучкість у маніпуляціях даними. Поняття вказівника вводять через оператори адреси (**&**) і розіменування (*****), пояснюючи їх як змінні, що зберігають адреси в пам'яті. У цій темі детально розглянемо концепцію вказівників, операції з ними, їхнє оголошення, ініціалізацію, використання для динамічного виділення пам'яті та практичні приклади застосування.

Вказівники є основою для розуміння багатьох концепцій у С, таких як масиви, динамічна пам'ять і передача параметрів у функції, що робить цю тему ключовою для подальшого вивчення програмування.

10.1. Оператор адреси (&)

Основи роботи з адресами

Коли змінна оголошується (наприклад, **int b;**), компілятор автоматично виділяє для неї місце в оперативній пам'яті та присвоює унікальну адресу. Ця адреса – числовий ідентифікатор комірки пам'яті, наприклад 150. Програміст не мусить знати конкретні адреси – компілятор і операційна система (ОС) беруть це на себе. Однак оператор адреси **&** дозволяє отримати цю адресу для подальшого використання.

```
#include <stdio.h>

int main() {
    int b = 10;
    printf("Значення: %d\n", b);         // 10
    printf("Адреса: %p\n", (void*)&b);  //
Наприклад, 0x7fff5fbff83c
    return 0;
}
```

- **`&b`** повертає адресу змінної **`b`** у пам’яті.
- **`%p`** у **`printf`** виводить адресу в шістнадцятковому форматі.

Відмінність від побітового І

`&` також є оператором побітового І, але контекст дозволяє їх розрізнити:

- **Унарний `&`** (наприклад, **`&b`**) – оператор адреси.
- **Бінарний `&`** (наприклад, **`a & b`**) – побітове І.

10.2. Оператор розіменування (*)

Отримання значення за адресою

Оператор **`*`** дозволяє отримати або змінити значення за адресою, на яку вказує вказівник. Ваш приклад демонструє це:

```
#include <stdio.h>

int main() {
    int a = 7;
    printf("Значення: %d\n", a);         // 7
    printf("Адреса: %p\n", (void*)&a);  //
Наприклад, 0x7fff5fbff83c
    printf("Розіменування: %d\n", *&a); // 7
    return 0;
}
```

- **`*&a`** еквівалентно **`a`**, бо **`&a`** повертає адресу, **`a *`** зчитує значення за цією адресою.

Відмінність від множення

Як і з **`&`**, зірочка **`*`** також є оператором множення, але:

- **Унарний `*`** (наприклад, **`*ptr`**) – розіменування.
- **Бінарний `*`** (наприклад, **`a * b`**) – множення.

10.3. Вказівники: оголошення та основи

Що таке вказівник?

Вказівник – це змінна, яка зберігає адресу іншої змінної в пам’яті [2]. Ваш документ визначає його як "змінну, значенням якої є адреса комірки в пам’яті". Оголошення вказівника виглядає так:

```
int *iPtr;      // Вказівник на int
double *dPtr;   // Вказівник на double
```

Синтаксис оголошення

Є кілька варіантів розміщення зірочки:

- `int *iPtr;` – рекомендований стиль (зірочка біля імені).
- `int* iPtr;` – допустимо, але може вводити в оману при оголошенні кількох змінних:

```
int* ptr1, ptr2; // ptr1 – вказівник, ptr2 –
int!
int *ptr1, *ptr2; // Обидва – вказівники
```

Неініціалізовані вказівники

Без ініціалізації вказівник містить випадкове значення (сміття), що може призвести до помилок при розіменуванні:

```
int *ptr; // Сміття
// printf("%d\n", *ptr); // Невизначена
поведінка!
```

10.4. Присвоювання значень вказівнику

Ініціалізація адресою

Вказівник отримує адресу через оператор `&`:

```
#include <stdio.h>

int main() {
    int value = 5;
```

```
    int *ptr = &value;
    printf("Адреса: %p\n", (void*)ptr);    //
Наприклад, 0x7fff5fbff83c
    printf("Значення: %d\n", *ptr);        // 5
    return 0;
}
```

Типова відповідність

Тип вказівника має відповідати типу змінної:

```
int i = 5;
double d = 3.14;
int *iPtr = &i;    // OK
double *dPtr = &d; // OK
// int *wrong = &d; // Помилка компіляції
```

Недопустимі присвоювання

Присвоювати цілі числа чи літерали адрес не можна:

```
int *ptr = 7; // Помилка: 7 – не адреса
```

10.5. Розіменування вказівників

Доступ до значення

Розіменування дозволяє читати або змінювати значення:

```
#include <stdio.h>

int main() {
    int value = 5;
    int *ptr = &value;
    printf("Перед: %d\n", *ptr); // 5
    *ptr = 10;
    printf("Після: %d\n", value); // 10
    return 0;
}
```

Зміна вказівника

Вказівник може вказувати на різні адреси:

```
#include <stdio.h>

int main() {
    int v1 = 5, v2 = 7;
    int *ptr = &v1;
    printf("v1: %d\n", *ptr); // 5
    ptr = &v2;
    printf("v2: %d\n", *ptr); // 7
    return 0;
}
```

10.6. Розіменування некоректних вказівників

Небезпека сміттєвих значень

Проілюструємо проблему неініціалізованих вказівників:

```
#include <stdio.h>

int main() {
    int *ptr; // Сміття
    // printf("%d\n", *ptr); // Збій програми
    return 0;
}
```

ОС блокує доступ до невиділеної пам’яті, викликаючи збій.

Нульові вказівники

Для уникнення проблем рекомендується ініціалізувати вказівники як **NULL**:

```
int *ptr = NULL;
// *ptr; // Збій, але легше відстежити
```

10.7. Розмір вказівників

Залежність від архітектури

Розмір вказівника залежить від архітектури:

- 32 біти (4 байти) на 32-бітних системах.
- 64 біти (8 байтів) на 64-бітних системах.

```
#include <stdio.h>

int main() {
    char *cPtr;
    int *iPtr;
    printf("Розмір char*: %zu\n", sizeof(cPtr));
    // 8 (на 64-бітній системі)
    printf("Розмір int*: %zu\n", sizeof(iPtr));
    // 8
    return 0;
}
```

10.8. Практичне застосування вказівників

Переваги вказівників

Можна навести такі 6 випадків використання:

1. **Масиви:** ітерація через вказівники.
2. **Динамічна пам'ять:** виділення через **`malloc`** або **`new`**.
3. **Передача великих даних:** без копіювання.
4. **Передача функцій:** як параметри.
5. **Поліморфізм:** у спадкуванні.
6. **Структури:** зв'язування об'єктів.

Приклад 10.1: Ітерація по масиву

```
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4};
    int *ptr = arr;
    for (int i = 0; i < 4; i++) {
        printf("%d ", *(ptr + i));
    }
```

```
    printf("\n");
    return 0;
}
```

```
Виведення:
1 2 3 4
```

Приклад 10.2. Передача у функцію

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    printf("x: %d, y: %d\n", x, y);
    return 0;
}
```

```
Виведення:
x: 10, y: 5
```

Приклад 10.3. Динамічне виділення

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

int main() {
    int *ptr = (int*)malloc(sizeof(int));
    *ptr = 42;
    printf("Значення: %d\n", *ptr);
    free(ptr);
    return 0;
```

```
}
```

10.9. Виділення та звільнення пам’яті

10.9.1. malloc

Виділяє пам’ять заданого розміру:

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

int main() {
    int n = 5;
    int *arr = (int*)malloc(n * sizeof(int));
    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

```
Виведення:
1 2 3 4 5
```

10.9.2. calloc

Виділяє та обнуляє:

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

int main() {
    int *arr = (int*)calloc(3, sizeof(int));
    for (int i = 0; i < 3; i++) {
        printf("%d ", arr[i]); // 0 0 0
    }
    free(arr);
    return 0;
}
```

10.9.3. realloc

Змінює розмір:

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

int main() {
    int *arr = (int*)malloc(2 * sizeof(int));
    arr[0] = 1; arr[1] = 2;
    arr = (int*)realloc(arr, 4 * sizeof(int));
    arr[2] = 3; arr[3] = 4;
    for (int i = 0; i < 4; i++) {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

```
Виведення:
1 2 3 4
```

10.9.4. free

Звільняє пам’ять:

```
int *ptr = (int*)malloc(sizeof(int));
free(ptr);
```

10.10. Додаткові приклади

Приклад 10.4. Арифметика вказівників

```
#include <stdio.h>

int main() {
    int arr[] = {10, 20, 30};
    int *ptr = arr;
    printf("%d\n", *(ptr + 1)); // 20
    return 0;
```

```
}
```

Приклад 10.5. Вказівник на вказівник

```
#include <stdio.h>

int main() {
    int x = 5;
    int *p = &x;
    int **pp = &p;
    printf("%d\n", **pp); // 5
    return 0;
}
```

10.11. Порівняння з іншими мовами

Python: немає явних вказівників, пам'ять керується автоматично.

Java: використовує посилання, але без прямого доступу до адрес.

10.12. Рекомендації

Ініціалізація: завжди ініціалізуйте вказівники (**`NULL`** або конкретною адресою).

Звільнення: не забувайте викликати **`free`** чи **`delete`**.

Перевірка: перевіряйте результат **`malloc`** на **`NULL`**.

Висновки

Вказівники в C надають потужний інструмент для роботи з пам'яттю, від базових операцій до динамічного виділення. Розширені приклади показали їхню універсальність, готуючи до складніших застосувань.

Запитання та завдання для самоконтролю

1. Поясніть що таке вказівник у мові С.
2. Як оператор адреси **&** пов'язаний із вказівниками?
3. Що повертає оператор **&**, коли застосовується до змінної?
4. Як відрізнити оператор адреси **&** від побітового І?
5. Що робить оператор розіменування *****?
6. Як відрізнити оператор розіменування ***** від оператора множення?
7. Як оголосити вказівник на змінну типу **int**?
8. Чому при оголошенні кількох вказівників у рядку зірочка потрібна біля кожного імені?
9. Який стиль оголошення вказівника рекомендований і чому?
10. Що містить неініціалізований вказівник?
11. Як присвоїти вказівнику адресу змінної?
12. Чому тип вказівника має відповідати типу змінної, на яку він вказує?
13. Чи можна присвоїти вказівнику ціле число, наприклад 7? Чому?
14. Як отримати значення за адресою, на яку вказує вказівник?
15. Як змінити значення змінної через вказівник?
16. Що станеться при розіменуванні неініціалізованого вказівника?
17. Як уникнути проблем із неініціалізованими вказівниками?
18. Який розмір має вказівник на 64-бітній системі?
19. Чому розмір вказівника не залежить від типу даних, на який він указує?
20. Як використати вказівник для ітерації по масиву?
21. Як передати велику структуру у функцію через вказівник?
22. Як працює функція **malloc()** і що вона повертає?
23. У чому різниця між **malloc()** і **calloc()**?
24. Як змінити розмір виділеної пам'яті за допомогою **realloc()**?

25. Як звільнити пам’ять, виділену через **`malloc()`**?
26. Як працює оператор new у C++ для виділення пам’яті?
27. Як звільнити пам’ять, виділену через **`new`**, у C++?
28. Що станеться, якщо спробувати звільнити пам’ять, не виділену динамічно?
29. Як перевірити, чи вдалося виділити пам’ять за допомогою **`malloc()`**?
30. Як передати вказівник у функцію для зміни значення змінної?

11. Використання вказівників для роботи з масивами. Динамічні масиви

Масиви є однією з основних структур даних у програмуванні, а в мові С їхня тісна взаємодія з вказівниками робить їх надзвичайно гнучкими. У цій темі детально дослідимо, як вказівники використовуються для роботи з масивами, включаючи адресну арифметику, індексацію, передачу масивів у функції та створення динамічних масивів у мові С.

11.1. Вказівники і масиви: схожість і відмінності

Схожість між вказівниками і масивами

У мові С ім'я масиву є вказівником на його перший елемент. При оголошенні фіксованого масиву, наприклад:

```
int array[4] = {5, 8, 6, 4};
```

змінна array зберігає адресу першого елемента (**`&array[0]`**). Це дозволяє використовувати array як вказівник типу **`int*`**.

```
#include <stdio.h>

int main() {
    int array[4] = {5, 8, 6, 4};
    printf("Адреса масиву: %p\n", (void*)array);
    printf("Адреса першого елемента: %p\n",
(void*)&array[0]);
    return 0;
}
```

```
Виведення (наприклад):
Адреса масиву: 0x7fff5fbff830
Адреса першого елемента: 0x7fff5fbff830
```

Це демонструє, що **`array`** і **`&array[0]`** мають однакове значення – адресу початку масиву в пам'яті. Більше того, ми можемо розіменувати **`array`**, щоб отримати перший елемент:

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

```
int main() {
    int array[4] = {5, 8, 6, 4};
    printf("Перший елемент: %d\n", *array); // 5
    return 0;
}
```

Розпад масиву у вказівник

Масив "розпадається" (decays) у вказівник на свій перший елемент у багатьох контекстах, наприклад, при передачі у функцію чи присвоюванні вказівнику:

```
#include <stdio.h>

int main() {
    int array[4] = {5, 8, 6, 4};
    int *ptr = array;
    // array автоматично стає &array[0]
    printf("Перший елемент через ptr: %d\n",
*ptr);
    // 5
    return 0;
}
```

Цей "розпад" є неявним перетворенням типу **`int[4]`** у **`int*`**, що дозволяє використовувати вказівники для доступу до елементів масиву.

Відмінності між масивами і вказівниками

Хоча **`array`** і **`ptr`** вказують на ту саму адресу, їхня природа різна:

1. Тип даних:

- **array** має тип **`int[4]`** – масив із 4 елементів.
- **ptr** має тип **`int*`** – вказівник на ціле число.

2. Оператор sizeof:

- Для масиву **`sizeof(array)`** повертає розмір усього масиву (кількість елементів × розмір типу).

- Для вказівника **sizeof(ptr)** повертає розмір адреси (4 або 8 байтів залежно від архітектури).

```
#include <stdio.h>

int main() {
    int array[4] = {5, 8, 6, 4};
    int *ptr = array;
    printf("Розмір масиву: %zu байтів\n",
sizeof(array)); // 16 (4 * 4)
    printf("Розмір вказівника: %zu байтів\n",
sizeof(ptr)); // 4 або 8
    return 0;
}
```

3. Оператор &:

- **&array** повертає вказівник на весь масив (тип **int (*)[4]**), хоча чисельно дорівнює **&array[0]**.
- **&ptr** повертає адресу самого вказівника в пам'яті.

```
#include <stdio.h>

int main() {
    int array[4] = {5, 8, 6, 4};
    int *ptr = array;
    printf("Адреса масиву: %p\n",
(void*)&array);
    printf("Адреса вказівника: %p\n",
(void*)&ptr);
    return 0;
}
```

Чому це важливо?

Розуміння відмінностей між масивами та вказівниками дозволяє уникнути помилок, таких як спроба змінити розмір масиву через вказівник або неправильне використання **sizeof** у функціях [9].

11.2. Передача масивів у функції

Розпад масиву при передачі

У С масиви передаються у функції через вказівники, а не копіюються. Це економить пам’ять, але означає втрату інформації про розмір:

```
#include <stdio.h>

void printSize(int *array) {
    printf("Розмір у функції: %zu\n",
sizeof(array)); // Розмір вказівника
}

int main() {
    int array[] = {1, 2, 3, 4, 5};
    printf("Розмір у main: %zu\n",
sizeof(array)); // 20 (5 * 4)
    printSize(array);
    return 0;
}
```

```
Виведення:
Розмір у main: 20
Розмір у функції: 4
```

Синтаксис передачі

Оголошення функції може використовувати як **`int array[]`**, так і **`int *array`** – це еквівалентно:

```
void func1(int array[]);
void func2(int *array);
```

Обидва варіанти інтерпретуються як **`int*`**, але **`int array[]`** читається краще, натякаючи на масив.

Передача розміру

Оскільки вказівник не знає розміру масиву, його треба передавати окремо:

```
#include <stdio.h>

void printArray(int *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int array[] = {1, 2, 3, 4};
    printArray(array, 4);
    return 0;
}
```

Зміна масиву через вказівник

Оскільки масив передається як вказівник, функція може змінювати його елементи:

```
#include <stdio.h>

void changeArray(int *array) {
    array[0] = 10;
}

int main() {
    int array[] = {1, 2, 3};
    printf("До: %d\n", array[0]);
    changeArray(array);
    printf("Після: %d\n", array[0]);
    return 0;
}
```

```
Виведення:
До: 1
Після: 10
```

11.3. Адресна арифметика й індексація масивів

Адресна арифметика

Додавання до вказівника враховує розмір типу даних:

- Для **int*** (4 байти) **ptr + 1** зсуває адресу на 4 байти.
- Для **char*** (1 байт) **ptr + 1** зсуває на 1 байт.

```
#include <stdio.h>

int main() {
    int value = 8;
    int *ptr = &value;
    printf("Адреса: %p\n", (void*)ptr);
    printf("Наступна адреса: %p\n", (void*)(ptr
+ 1));
    return 0;
}
```

```
Виведення (приклад):
Адреса: 0x7fff5fbff83c
Наступна адреса: 0x7fff5fbff840
```

Послідовність елементів масиву

Елементи масиву розміщені в пам’яті послідовно:

```
#include <stdio.h>

int main() {
    int array[] = {7, 8, 2, 4};
    for (int i = 0; i < 4; i++) {
        printf("Елемент %d: %p\n", i,
               (void*)&array[i]);
    }
    return 0;
}
```

```
Виведення:
Елемент 0: 0x7fff5fbff830
```

```
Елемент 1: 0x7fff5fbff834
Елемент 2: 0x7fff5fbff838
Елемент 3: 0x7fff5fbff83c
```

Індексація через вказівники

Оператор `[]` – спеціальна синтаксична конструкція для адресної арифметики: `array[i]` еквівалентно `*(array + i)`:

```
#include <stdio.h>

int main() {
    int array[] = {5, 8, 6};
    printf("array[1]: %d\n", array[1]);
    printf("*(array + 1): %d\n", *(array + 1));
    return 0;
}
```

```
Виведення:
array[1]: 8
*(array + 1): 8
```

Ітерація через вказівник

```
#include <stdio.h>

int main() {
    char name[] = "Jonathan";
    int numVowels = 0;
    for (char *ptr = name; *ptr != '\0'; ptr++)
{
        char c = *ptr;
        if (c == 'a' || c == 'e' || c == 'i' ||
c == 'o' || c == 'u' ||
            c == 'A' || c == 'E' || c == 'I' ||
c == 'O' || c == 'U') {
            numVowels++;
        }
    }
```

```
    printf("%s має %d голосних\n", name,
numVowels);
    return 0;
}
```

```
Виведення:
Jonathan має 3 голосних
```

11.4. Динамічні масиви

Обмеження фіксованих масивів

Існують три проблеми фіксованих масивів:

1. **Втрата пам'яті**: якщо масив більший за потрібне.
2. **Обмеження стеку**: локальні масиви займають стек (зазвичай 1 МБ).
3. **Фіксований розмір**: неможливість адаптації до введення.

Динамічне виділення в C

У C динамічні масиви створюються через функції **malloc**, **calloc**, **realloc** і звільняються через **free.**

Приклад 11.1. Виділення через malloc

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

int main() {
    int size;
    printf("Введіть розмір масиву: ");
    scanf("%d", &size);
    int *array = (int*)malloc(size *
sizeof(int));
    if (array == NULL) {
        printf("Помилка виділення пам'яті\n");
        return 1;
    }
    for (int i = 0; i < size; i++) {
```

```
        array[i] = i + 1;
        printf("%d ", array[i]);
    }
    free(array);
    return 0;
}
```

Приклад 11.2. Виділення через calloc

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

int main() {
    int size = 5;
    int *array = (int*)calloc(size,
sizeof(int));
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]); // 0 0 0 0 0
    }
    free(array);
    return 0;
}
```

Приклад 11.3. Зміна розміру через realloc

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

int main() {
    int *array = (int*)malloc(2 * sizeof(int));
    array[0] = 1; array[1] = 2;
    array = (int*)realloc(array, 4 *
sizeof(int));
    array[2] = 3; array[3] = 4;
    for (int i = 0; i < 4; i++) {
        printf("%d ", array[i]);
    }
    free(array);
    return 0;
```

```
}
```

```
Виведення:
1 2 3 4
```

Звільнення пам’яті

```
int *array = (int*)malloc(10 * sizeof(int));
free(array);
```

Переваги динамічних масивів

- **Гнучкість**: розмір визначається під час виконання.
- **Використання купи**: більший обсяг пам’яті порівняно зі стеком.

Недоліки

- **Ручне керування**: потрібно явно звільняти пам’ять.
- **Фрагментація**: часте виділення/звільнення може ускладнити доступ до пам’яті.

11.5. Практичні приклади

Приклад 11.4. Сортування масиву через вказівники

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

void bubbleSort(int *arr, int size) {
    for (int i = 0; i < size - 1; i++) {
        for (int j = 0; j < size - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
```

```
}

int main() {
    int arr[] = {5, 3, 8, 4, 2};
    int size = sizeof(arr) / sizeof(arr[0]);
    bubbleSort(arr, size);
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    return 0;
}
```

```
Виведення:
2 3 4 5 8
```

Приклад 11.5. Динамічний масив із введенням

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

int main() {
    int size;
    printf("Розмір масиву: ");
    scanf("%d", &size);
    int *arr = (int*)malloc(size * sizeof(int));
    printf("Введіть %d елементів:\n", size);
    for (int i = 0; i < size; i++) {
        scanf("%d", &arr[i]);
    }
    printf("Масив: ");
    for (int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

11.6. Рекомендації

Перевірка NULL: завжди перевіряйте результат **`malloc/calloc`**.

Звільнення пам'яті: не забувайте викликати **`free`** для уникнення витоків.

Розмір масиву: передавайте розмір у функції явно.

Висновки

Вказівники дозволяють ефективно працювати з масивами в С, від індексації до динамічного виділення. Розширені приклади демонструють їхню універсальність, готуючи до складніших структур даних.

Запитання та завдання для самоконтролю

1. Як пов'язані вказівники і масиви в мові С?
2. Що зберігає ім'я масиву, коли його оголошено як **`int array[5]`**?
3. Як отримати адресу першого елемента масиву?
4. Що означає "розпад масиву у вказівник" і в яких випадках це відбувається?
5. Як розіменувати ім'я масиву, щоб отримати його перший елемент?
6. У чому різниця між типом масиву **`int[4]`** і вказівником **`int*`**?
7. Як працює оператор **`sizeof`** із масивом і вказівником?
8. Що повертає &array для масиву **`int array[5]`**?
9. Чому масив не копіюється при передачі у функцію?
10. Як передати масив у функцію, щоб функція знала його розмір?
11. У чому різниця між оголошеннями функцій **`void func(int array[])`** і **`void func(int *array)`**?
12. Як змінити елемент масиву всередині функції через вказівник?
13. Поясніть що таке адресна арифметика у контексті вказівників.
14. Як додавання 1 до вказівника залежить від типу даних?

15. Як визначити, чи розташовані елементи масиву послідовно в пам'яті?
16. Як працює вираз `*(array + i)` для масиву?
17. Чому `array[i]` еквівалентно `*(array + i)`?
18. Як використати вказівник для ітерації по масиву без індексів?
19. Які обмеження мають фіксовані масиви в C?
20. Що таке динамічне виділення пам'яті і чому воно корисне для масивів?
21. Як виділити динамічний масив за допомогою `malloc()`?
22. У чому різниця між `malloc()` і `calloc()` при створенні динамічного масиву?
23. Як змінити розмір динамічного масиву за допомогою `realloc()`?
24. Як звільнити пам'ять, виділену для динамічного масиву?
25. Як перевірити, чи вдалося виділити пам'ять для масиву?
26. Що станеться, якщо не звільнити пам'ять динамічного масиву?
27. Як передати динамічний масив у функцію?
28. Як ініціалізувати всі елементи динамічного масиву нулями без `calloc()`?
29. Які переваги динамічних масивів над фіксованими?
30. Як уникнути витоків пам'яті при роботі з динамічними масивами?

12. Функції користувача

Функції є основою модульного програмування в мові С, дозволяючи розбивати код на логічні блоки, які можна повторно використовувати. У цій темі ми детально розглянемо створення функцій користувача в С, їхню структуру, способи передачі аргументів, повернення значень і практичне застосування.

12.1. Основи функцій: параметри та аргументи

Параметри vs. Аргументи

У С виділяють параметри (формальні параметри) та аргументи (фактичні параметри) [17]:

- **Параметр** – це змінна, оголошена у прототипі або визначенні функції.
- **Аргумент** – це значення, передане у функцію під час її виклику.

Приклад оголошення та виклику:

```
#include <stdio.h>

void boo(int x); // Прототип: x – параметр

void boo(int x) { // Визначення: x – параметр
    printf("x = %d\n", x);
}

int main() {
    boo(7);          // 7 – аргумент
    int y = 5;
    boo(y + 1);      // y + 1 – аргумент
    return 0;
}
```

```
Виведення:
x = 7
x = 6
```

При виклику функції:

1. Створюються локальні копії параметрів (**x** у цьому випадку).
2. Значення аргументів копіюються в ці параметри.
3. Після завершення функції параметри знищуються.

Порядок ініціалізації параметрів

Порядок обробки параметрів залежить від компілятора. У C це не визначено стандартом, тому при передачі кількох аргументів (особливо виразів) результат може бути непередбачуваним.

```
#include <stdio.h>

int getX() {
    printf("getX\n");
    return 1;
}

int getY() {
    printf("getY\n");
    return 2;
}

void printAll(int a, int b) {
    printf("a = %d, b = %d\n", a, b);
}

int main() {
    printAll(getX(), getY());
    return 0;
}
```

```
Можливе виведення:
getY
getX
a = 1, b = 2
```

Рекомендується уникати передачі викликів функцій як аргументів, якщо порядок виконання критичний [3].

Область видимості параметрів

Параметри мають локальну область видимості всередині функції:

```
void boo(int x) { // x створюється тут
    x = 10;
} // x знищується тут
```

12.2. Способи передачі аргументів

У мові С є два основні способи передачі аргументів: за значенням і за адресою.

Передача за значенням

За замовчуванням аргументи в С передаються за значенням – копіюється значення аргументу в параметр функції.

Приклад 12.1. Передача за значенням

```
#include <stdio.h>

void boo(int y) {
    printf("y = %d\n", y);
    y = 8;
    printf("y після зміни = %d\n", y);
}

int main() {
    int x = 7;
    printf("x до = %d\n", x);
    boo(x);
    printf("x після = %d\n", x);
    return 0;
}
```

```
Виведення:
x до = 7
y = 7
```

```
y після зміни = 8
x після = 7
```

x не змінюється, бо функція працює з копією.

Переваги:

- **Безпека:** вихідний аргумент не змінюється.
- **Гнучкість:** можна передавати літерали, змінні, вирази.

Недоліки:

- **Витрати пам'яті** при копіюванні великих структур.

Приклад 12.2. Передача структури за значенням

```
#include <stdio.h>

struct Point {
    int x, y;
};

void printPoint(struct Point p) {
    printf("Точка: (%d, %d)\n", p.x, p.y);
}

int main() {
    struct Point pt = {3, 4};
    printPoint(pt);
    return 0;
}
```

Копіювання структури може бути неефективним для великих об'єктів.

Передача за адресою

Передача за адресою використовує вказівники для доступу до оригінальних даних.

Приклад 12.3. Зміна значення через вказівник

```
#include <stdio.h>

void boo(int *ptr) {
```

```
    *ptr = 7;
}

int main() {
    int value = 4;
    printf("value до = %d\n", value);
    boo(&value);
    printf("value після = %d\n", value);
    return 0;
}
```

```
Виведення:
value до = 4
value після = 7
```

Приклад 12.4. Передача масиву

```
#include <stdio.h>

void printArray(int *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int arr[] = {9, 8, 6, 4};
    printArray(arr, 4);
    return 0;
}
```

Масиви автоматично "розпадаються" у вказівник на перший елемент.

Переваги:

- **Ефективність:** копіюється лише адреса.
- **Зміна даних:** дозволяє функції модифікувати аргументи.

Недоліки:

- **Небезпека:** розіменування `NULL` призводить до збою.

- **Складність:** потрібна перевірка вказівників.

Приклад 12.5. Перевірка на NULL

```
#include <stdio.h>

void printArray(int *array, int size) {
    if (array == NULL) {
        printf("Масив невизначений\n");
        return;
    }
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}

int main() {
    int *arr = NULL;
    printArray(arr, 0);
    return 0;
}
```

Передача за константною адресою

Для захисту даних використовуємо **const**:

```
#include <stdio.h>

void printArray(const int *array, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", array[i]);
        // array[i] = 0; // Помилка: const забороняє зміну
    }
    printf("\n");
}

int main() {
    int arr[] = {1, 2, 3};
    printArray(arr, 3);
```

```
    return 0;
}
```

12.3. Повернення значень із функцій

Повернення за значенням

Найпростіший спосіб – копія значення повертається в викликаючий код.

Приклад 12.6. Повернення за значенням

```
#include <stdio.h>

int doubleValue(int a) {
    int result = a * 2;
    return result;
}

int main() {
    int x = doubleValue(5);
    printf("x = %d\n", x);
    return 0;
}
```

```
Виведення:
x = 10
```

Перевагою є безпека: локальні змінні знищуються, але значення копіюється. **Недоліком є неефективність** для великих структур.

Повернення за адресою

Повертається вказівник на дані.

Приклад 12.7. Повернення динамічного масиву

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

```
int* allocateArray(int size) {
    int *arr = (int*)malloc(size * sizeof(int));
    for (int i = 0; i < size; i++) {
        arr[i] = i + 1;
    }
    return arr;
}

int main() {
    int *arr = allocateArray(3);
    for (int i = 0; i < 3; i++) {
        printf("%d ", arr[i]);
    }
    free(arr);
    return 0;
}
```

```
Виведення:
1 2 3
```

Помилка: повернення локальної змінної

```
int* badReturn() {
    int x = 5;
    return &x; // Помилка: x знищується
}
```

Використовуйте для динамічної пам’яті або аргументів функції.

12.4. Практичні приклади

Приклад 12.8. Обмін значень через вказівники

```
#include <stdio.h>

void swap(int *a, int *b) {
    int temp = *a;
    *a = *b;
    *b = temp;
```

```
}

int main() {
    int x = 5, y = 10;
    swap(&x, &y);
    printf("x = %d, y = %d\n", x, y);
    return 0;
}
```

```
Виведення:
x = 10, y = 5
```

Приклад 12.9. Обчислення суми масиву

```
#include <stdio.h>

int arraySum(int *arr, int size) {
    int sum = 0;
    for (int i = 0; i < size; i++) {
        sum += arr[i];
    }
    return sum;
}

int main() {
    int arr[] = {1, 2, 3, 4};
    printf("Сума: %d\n", arraySum(arr, 4));
    return 0;
}
```

```
Виведення:
Сума: 10
```

Приклад 12.10. Рекурсія (факторіал)

```
#include <stdio.h>

int factorial(int n) {
```

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

int main() {
    printf("Факторіал 5 = %d\n", factorial(5));
    return 0;
}
```

```
Виведення:
Факторіал 5 = 120
```

Висновки

Функції користувача у C дозволяють створювати модульний, повторно використовуваний код. Розширені приклади демонструють гнучкість передачі та повернення даних, готуючи до складніших задач програмування.

Запитання та завдання для самоконтролю

1. Що таке функція користувача в мові C?
2. У чому різниця між параметром і аргументом функції?
3. Як оголосити прототип функції з двома параметрами типу `int`?
4. Поясніть, що відбувається з параметрами функції при її виклику.
5. Чому порядок ініціалізації параметрів у функції може бути непередбачуваним?
6. Яка область видимості параметрів функції?
7. Які основні способи передачі аргументів у функцію в мові C?
8. Як працює передача аргументів за значенням?
9. Чи може функція змінити вихідний аргумент при передачі за значенням?
10. Які переваги передачі за значенням?
11. Які недоліки має передача за значенням для великих структур?

12. Як передати аргумент за адресою в мові С?
13. Як функція може змінити значення змінної через передачу за адресою?
14. Як захистити аргумент від змін при передачі за адресою?
15. Що станеться при розіменуванні нульового вказівника у функції?
16. Як передати масив у функцію і чому потрібен окремий параметр розміру?
17. Як функція може повернути значення за значенням?
18. Які переваги повернення за значенням?
19. Як повернути динамічний масив із функції?
20. Чому не можна повертати адресу локальної змінної з функції?
21. Як повернути кілька значень із функції через вказівники?
22. Як перевірити, чи вказівник, переданий у функцію, дійсний?
23. Як передати структуру в функцію за значенням та за адресою?
24. Як обчислити факторіал числа за допомогою рекурсивної функції?
25. Які ризики має змішування передачі вхідних і вихідних параметрів?
26. Як уникнути витоків пам’яті при поверненні динамічної пам’яті з функції?
27. Як оптимізувати передачу великих даних у функцію?
28. Як використовувати const для параметрів, переданих за адресою?
29. Які типи даних найкраще передавати за значенням, а які – за адресою?
30. Як коментарі можуть допомогти при роботі з функціями, що змінюють аргументи?

13. Рекурсивні алгоритми

Рекурсія – це потужний інструмент у програмуванні, який дозволяє функції викликати саму себе для розв'язання задачі шляхом розбиття її на менші підзадачі. У цій темі ми детально розглянемо організацію пам'яті в С (стек і купу), механізм роботи рекурсії, її переваги та недоліки, а також практичні приклади реалізації рекурсивних алгоритмів.

13.1. Стек і купа: організація пам'яті в С

Сегменти пам'яті

Є п'ять основних сегментів пам'яті, які використовуються програмами в С:

1. **Сегмент коду** – містить скомпільований код програми (тільки для читання).
2. **Сегмент BSS** – зберігає неініціалізовані глобальні та статичні змінні (заповнені нулями).
3. **Сегмент даних** – зберігає ініціалізовані глобальні та статичні змінні.
4. **Купа** – для динамічного виділення пам'яті [22].
5. **Стек викликів** – для локальних змінних, параметрів і керування викликами функцій.

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

int globalVar;           // BSS
int initializedVar = 5; // Сегмент даних

int main() {
    int localVar = 10;  // Стек
    int *dynamicVar = (int*)malloc(sizeof(int));
// Купа
    *dynamicVar = 20;
    printf("localVar: %d, dynamicVar: %d\n",
localVar, *dynamicVar);
    free(dynamicVar);
```

```
    return 0;            // Сегмент коду
}
```

Купа

Купа – це область пам'яті для динамічного виділення через функції **malloc**, **calloc**, **realloc** і звільнення через **free**. Адреси, виділені з купи, не обов'язково послідовні.

Приклад 13.1. Виділення пам'яті з купи

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

int main() {
    int *ptr1 = (int*)malloc(sizeof(int));
    int *ptr2 = (int*)malloc(sizeof(int));
    *ptr1 = 1;
    *ptr2 = 2;
    printf("ptr1: %p, ptr2: %p\n", (void*)ptr1,
(void*)ptr2);
    free(ptr1);
    free(ptr2);
    return 0;
}
```

```
Виведення (приклад):
ptr1: 0x5623e4c0f260, ptr2: 0x5623e4c0f270
```

- Адреси можуть бути непослідовними через фрагментацію купи.

Стек викликів

Стек викликів – це структура типу **LIFO** (**Last In, First Out**), яка керує виконанням функцій. Можна порівняти стек із стосом тарілок або поштових скриньок, над якими можна робити три операції:

- **Додавання (push)** – розміщення нового фрейму.
- **Видалення (pop)** – вилучення фрейму.

- **Перегляд вершини (peek)** – доступ до поточного фрейму.

Приклад 13.2. Ілюстрація стеку

```
#include <stdio.h>

void func(int x) {
    printf("func: %d\n", x);
}

int main() {
    printf("main початок\n");
    func(5);
    printf("main кінець\n");
    return 0;
}
```

```
Виведення:
main початок
func: 5
main кінець
```

Стек викликів:

- `main` додається.
- `func` додається поверх `main`.
- `func` завершується і видаляється.
- `main` завершується.

Механізм роботи стеку

Кожен виклик функції створює фрейм стеку, який містить:

- **Зворотну адресу** (куди повернутися після завершення).
- **Параметри функції**.
- **Локальні змінні**.
- **Збережені регістри**.

При завершенні функції фрейм видаляється, а вказівник стеку (stack pointer) зсувається вниз.

Приклад 13.3. Перегляд стеку викликів

```
#include <stdio.h>

void level2(int z) {
    printf("Рівень 2: %d\n", z);
}

void level1(int y) {
    printf("Рівень 1: %d\n", y);
    level2(y + 1);
}

int main() {
    printf("main\n");
    level1(5);
    return 0;
}
```

```
Виведення:
main
Рівень 1: 5
Рівень 2: 6
```

Стек:
main → level1 → level2 (push).
level2 → level1 → main (pop).

Переповнення стеку

Стек має фіксований розмір (зазвичай 1 МБ у Windows), і надмірне використання може спричинити переповнення стеку.

Приклад 13.4. Переповнення через великий масив

Програма може завершитися з помилкою через переповнення.

```
#include <stdio.h>

int main() {
```

```
    int stack[1000000]; // Занадто великий масив
для стеку
    printf("Масив створено\n");
    return 0;
}
```

Приклад 13.5. Переповнення через нескінченну рекурсію

Наступна програма викликає переповнення стеку.

```
#include <stdio.h>

void infinite() {
    infinite(); // Нескінченний виклик
}

int main() {
    infinite();
    return 0;
}
```

13.2. Рекурсія: основи та принципи

Що таке рекурсія?

Рекурсія – це коли функція викликає сама себе для розв'язання задачі, розбиваючи її на менші підзадачі.

Умова завершення (базовий випадок)

Кожен рекурсивний алгоритм потребує базового випадку – умови, за якої рекурсія припиняється.

Приклад 13.6. Рекурсія з умовою завершення

```
#include <stdio.h>

void countout(int count) {
    printf("push %d\n", count);
    if (count > 1) {
        countout(count - 1);
```

```
    }
    printf("pop %d\n", count);
}

int main() {
    countout(3);
    return 0;
}
```

```
Виведення:
push 3
push 2
push 1
pop 1
pop 2
pop 3
```

Базовий випадок: **`count <= 1`**.

Механізм роботи

Кожен виклик додає фрейм у стек.

При досягненні базового випадку фрейми починають видалятися.

Порядок операцій залежить від розміщення рекурсивного виклику в коді.

Приклад 13.7. Сума чисел

```
#include <stdio.h>

int sumCount(int value) {
    if (value <= 0) return 0;  // Базовий
випадок
    if (value == 1) return 1;  // Базовий
випадок
    return sumCount(value - 1) + value;
}

int main() {
```

```
    printf("Сума до 4: %d\n", sumCount(4));
    return 0;
}
```

```
Виведення:
Сума до 4: 10 (1 + 2 + 3 + 4).
```

Стек:

- sumCount(4) → sumCount(3) + 4.
- sumCount(3) → sumCount(2) + 3.
- sumCount(2) → sumCount(1) + 2.
- sumCount(1) → 1.

13.3. Рекурсивні алгоритми: приклади

Приклад 13.8. Обчислення чисел Фібоначчі

Числа Фібоначчі: **F(n)=F(n−1)+F(n−2), де F(0)=0, F(1)=1.**

```
#include <stdio.h>

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);
}

int main() {
    for (int i = 0; i < 10; i++) {
        printf("%d ", fibonacci(i));
    }
    printf("\n");
    return 0;
}
```

```
Виведення:
0 1 1 2 3 5 8 13 21 34
```

Приклад 13.9. Рекурсивний факторіал

$n!=n\times(n-1)\times\ldots\times1$, де $0!=1$.

```
#include <stdio.h>

int factorial(int n) {
    if (n <= 1) return 1;  // Базовий випадок
    return n * factorial(n - 1);
}

int main() {
    for (int i = 0; i < 8; i++) {
        printf("%d! = %d\n", i, factorial(i));
    }
    return 0;
}
```

```
Виведення:
0! = 1
1! = 1
2! = 2
3! = 6
4! = 24
5! = 120
6! = 720
7! = 5040
```

Приклад 13.10. Сума цифр

Сума цифр числа розраховується шляхом виділення цифр через ділення та остачу.

```
#include <stdio.h>

int sumDigits(int x) {
    if (x < 10) return x;  // Базовий випадок
    return (x % 10) + sumDigits(x / 10);
}

int main() {
    int num = 83569;
```

```
    printf("Сума цифр %d = %d\n", num,
sumDigits(num));
    return 0;
}
```

```
Виведення:
Сума цифр 83569 = 31.
```

Приклад 13.11. Бінарне число

Переведення числа в бінарний вигляд через рекурсію.

```
#include <stdio.h>

void printBinary(int x) {
    if (x <= 0) return;  // Базовий випадок
    printBinary(x / 2);
    printf("%d", x % 2);
}

int main() {
    int x;
    printf("Введіть число: ");
    scanf("%d", &x);
    if (x <= 0) printf("0 або від'ємне
число\n");
    else {
        printBinary(x);
        printf("\n");
    }
    return 0;
}
```

```
Введення:
13
Виведення:
1101
```

Приклад 13.12. Бінарне число з обробкою

Розширення попереднього завдання для нуля та від'ємних чисел.

```
#include <stdio.h>

void printBinary(int x) {
    if (x == 0) {
        printf("0");
        return;
    }
    if (x < 0) {
        printf("-");
        printBinary(-x);
        return;
    }
    printBinary(x / 2);
    printf("%d", x % 2);
}

int main() {
    int x;
    printf("Введіть число: ");
    scanf("%d", &x);
    printBinary(x);
    printf("\n");
    return 0;
}
```

```
Введення:
-5
Виведення:
-101
```

13.4. Рекурсія vs Ітерація

Порівняння

Будь-яку рекурсивну задачу можна розв'язати ітеративно і це буде швидше працювати, але рекурсія часто простіша.

Приклад 13.13. Ітеративний факторіал

```
#include <stdio.h>

int factorialIter(int n) {
    int result = 1;
    for (int i = 1; i <= n; i++) {
        result *= i;
    }
    return result;
}

int main() {
    printf("5! = %d\n", factorialIter(5));
    return 0;
}
```

```
Виведення:
5! = 120
```

Переваги рекурсії:

- **Простота коду** для складних задач (наприклад, обхід дерев).
- **Читабельність**.

Недоліки:

- **Витрати на стек** (кожен виклик додає фрейм).
- **Ризик переповнення**.

Приклад 13.14. Ітеративна сума цифр

```
#include <stdio.h>

int sumDigitsIter(int x) {
    int sum = 0;
    while (x > 0) {
        sum += x % 10;
        x /= 10;
```

```
    }
    return sum;
}

int main() {
    printf("Сума цифр 83569 = %d\n",
sumDigitsIter(83569));
    return 0;
}
```

```
Виведення:
Сума цифр 83569 = 31
```

Коли використовувати рекурсію?

- Задача природно розбивається на підзадачі.
- Глибина рекурсії обмежена.
- Продуктивність некритична.

13.5. Додаткові приклади

Приклад 13.15. Обхід масиву рекурсивно

```
#include <stdio.h>

void printArray(int *arr, int size, int index) {
    if (index >= size) return;  // Базовий
випадок
    printf("%d ", arr[index]);
    printArray(arr, size, index + 1);
}

int main() {
    int arr[] = {1, 2, 3, 4, 5};
    printArray(arr, 5, 0);
    printf("\n");
    return 0;
}
```

```
Виведення:
1 2 3 4 5
```

13.6. Рекомендації

Базовий випадок: завжди визначайте умову завершення.

Оптимізація: для великих `n` розглядайте ітеративний підхід.

Документація: коментуйте рекурсивні функції для зрозумілості.

Тестування: перевіряйте граничні випадки (0, від'ємні числа).

Висновки

Рекурсія у C спирається на стек викликів, дозволяючи елегантно розв'язувати складні задачі. Розширені приклади демонструють її універсальність, а аналіз стеку та купи підкреслює важливість управління пам'яттю.

Запитання та завдання для самоконтролю

1. Що таке рекурсивний алгоритм у мові C?
2. Які основні сегменти пам'яті використовуються програмами в C?
3. Що зберігається в сегменті коду програми?
4. Яка роль сегменту BSS у пам'яті програми?
5. Як сегмент даних відрізняється від сегменту BSS?
6. Що таке купа і для чого вона використовується?
7. Як працює динамічне виділення пам'яті з купи в C?
8. Чому послідовні запити до купи не завжди дають послідовні адреси?
9. Що таке стек викликів і яка його основна функція?
10. Які операції можна виконувати зі стеком (push, pop, peek)?
11. Поясніть що означає принцип LIFO у контексті стеку.
12. Як стек викликів відстежує виконання функцій?

13. Що входить до складу фрейму стеку при виклику функції?
14. Що таке зворотна адреса у фреймі стеку?
15. Як процесор використовує стек викликів при завершенні функції?
16. Що таке переповнення стеку і які його причини?
17. Як великий масив може спричинити переповнення стеку?
18. Як нескінченна рекурсія призводить до переповнення стеку?
19. Які переваги має стек порівняно з купою?
20. Які недоліки стеку обмежують його використання?
21. Що таке базовий випадок у рекурсивному алгоритмі?
22. Чому умова завершення необхідна для рекурсії?
23. Як працює рекурсивна функція для обчислення факторіалу?
24. Як рекурсивно знайти суму цифр числа?
25. Як рекурсивно вивести бінарне представлення числа?
26. Як обчислити числа Фібоначчі за допомогою рекурсії?
27. У чому різниця між рекурсивним та ітеративним підходами?
28. Чому ітеративні функції зазвичай ефективніші за рекурсивні?
29. У яких випадках рекурсія є кращим вибором, ніж ітерація?
30. Як уникнути переповнення стеку при використанні рекурсії?

14. Структури

Структури в мові С є ключовим інструментом для групування змінних різних типів у єдину логічну одиницю. У цій темі зосередимося на оголошенні, визначенні, ініціалізації структур, доступі до їхніх членів, вкладених структурах та особливостях роботи з пам'яттю, таких як вирівнювання.

14.1. Навіщо потрібні структури?

Проблема окремих змінних

Представлення складного об'єкта через набір окремих змінних створює труднощі в управлінні даними:

```
char myName[50];
int myBirthDay;
int myBirthMonth;
int myBirthYear;
int myHeight;
int myWeight;
```

- Потрібно створювати нові набори змінних для кожного об'єкта.
- Код стає громіздким і важким для підтримки.

Рішення проблеми – структури

Структури дозволяють об'єднати пов'язані дані в один тип:

```
struct Person {
    char name[50];
    int birthDay;
    int birthMonth;
    int birthYear;
    int height;
    int weight;
};
```

Одна змінна типу **`struct Person`** замінює шість окремих змінних.

Приклад 14.1: Організація даних через структури

```
#include <stdio.h>

struct Person {
    char name[50];
    int birthDay;
    int birthMonth;
    int birthYear;
    int height;
    int weight;
};

int main() {
    struct Person john = {"John Doe", 15, 6,
1995, 175, 70};
    struct Person alice = {"Alice Smith", 22, 3,
1998, 165, 55};
    printf("Ім'я: %s, Зріст: %d\n", john.name,
john.height);
    printf("Ім'я: %s, Зріст: %d\n", alice.name,
alice.height);
    return 0;
}
```

```
Виведення:
Ім'я: John Doe, Зріст: 175
Ім'я: Alice Smith, Зріст: 165
```

Структури підвищують організацію даних.

Порівняння з масивами

- Масиви: групують дані одного типу (наприклад, `int numbers[5]`).
- **Структури:** групують дані різних типів у єдину сутність.

14.2. Оголошення та визначення структур

Оголошення структури

Оголошення задає шаблон структури за допомогою ключового слова **struct**:

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

- **Employee** – ім'я типу (з великої літери за конвенцією).
- **id**, **age**, **salary** – члени (поля) структури.
- Оголошення структури не виділяє пам'ять.

Важливо:

- Крапка з комою (;) після фігурних дужок обов'язкова, інакше компілятор видасть помилку.

Приклад 14.2. Помилка без крапки з комою

```
struct Test {
    int x
} // Помилка: відсутня ;
struct Test t;
```

Визначення змінних структури

Для створення екземпляра структури оголошуємо змінну:

```
struct Employee john;
struct Employee james;
```

- У C необхідно використовувати struct перед ім'ям типу.

Приклад 14.3: Оголошення кількох змінних

```
#include <stdio.h>

struct Employee {
    short id;
```

```
    int age;
    double salary;
};

int main() {
    struct Employee john, james;
    printf("Розмір john: %zu, james: %zu\n",
sizeof(john), sizeof(james));
    return 0;
}
```

```
Виведення (приклад):
Розмір john: 16, james: 16
```

14.3. Доступ до членів структур

Оператор вибору члена (.)

Доступ до полів здійснюється через крапку (.):

```
john.id = 8;
john.age = 27;
john.salary = 32.17;
```

Приклад 14.4. Ініціалізація та операції

```
#include <stdio.h>

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

int main() {
    struct Employee john;
    john.id = 1;
    john.age = 25;
    john.salary = 30000.0;
```

```
    john.salary += 5000.0; // Підвищення зарпла-
ти
    john.age++;            // День народження

    printf("ID: %d, Вік: %d, Зарплата: %.2f\n",
john.id, john.age, john.salary);
    return 0;
}
```

```
Виведення:
ID: 1, Вік: 26, Зарплата: 35000.00
```

Неініціалізовані члени

Члени структури без ініціалізації містять випадкові значення.

Приклад 14.5. Сміття в структурі

```
#include <stdio.h>

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

int main() {
    struct Employee john;
    printf("Неініціалізовані: ID: %d, Вік: %d,
Зарплата: %f\n",
           john.id, john.age, john.salary);
    return 0;
}
```

```
Виведення (приклад):
```

```
Неініціалізовані: ID: 32767, Вік: -858993460,
Зарплата: 0.000000
```

14.4. Ініціалізація структур

Ручна ініціалізація

Присвоювання значень кожному члену вручну:

```
struct Employee john;
john.id = 5;
john.age = 27;
john.salary = 45000.0;
```

Список ініціалізації

У C можна ініціалізувати структуру під час оголошення:

```
struct Employee john = {5, 27, 45000.0};
```

Неініціалізовані члени отримують значення 0.

Приклад 14.6. Часткова ініціалізація

```
#include <stdio.h>

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

int main() {
    struct Employee john = {1, 25}; // salary =
0.0
    printf("ID: %d, Вік: %d, Зарплата: %.2f\n",
john.id, john.age, john.salary);
    return 0;
}
```

```
Виведення:
```

```
ID: 1, Вік: 25, Зарплата: 0.00
```

14.5. Вкладені структури

Оголошення вкладених структур

Структури можуть містити інші структури як поля:

```
struct Date {
    int day;
    int month;
    int year;
};

struct Person {
    char name[50];
    struct Date birth;
};
```

Ініціалізація вкладених структур

Використовуємо вкладені списки ініціалізації:

```
struct Person john = {"John Doe", {15, 6,
1995}};
```

Приклад 14.7. Робота з вкладеною структурою

```
#include <stdio.h>

struct Date {
    int day;
    int month;
    int year;
};

struct Person {
    char name[50];
    struct Date birth;
};
```

```
int main() {
    struct Person john = {"John Doe", {15, 6,
1995}};
    printf("Ім'я: %s, Народження: %d/%d/%d\n",
           john.name, john.birth.day,
john.birth.month, john.birth.year);
    return 0;
}
```

```
Виведення:
Ім'я: John Doe, Народження: 15/6/1995
```

14.6. Розмір структур і вирівнювання

Очікуваний розмір

Розмір структури – це сума розмірів її членів, але може бути скоригований через вирівнювання.

Приклад 14.8: Розмір структури

```
#include <stdio.h>

struct Employee {
    short id;    // 2 байти
    int age;     // 4 байти
    double salary; // 8 байтів
};

int main() {
    printf("Розмір Employee: %zu байтів\n",
sizeof(struct Employee));
    return 0;
}
```

```
Виведення (приклад):
Розмір Employee: 16 байтів
(замість 14 через вирівнювання)
```

Вирівнювання пам’яті

Компілятор додає "пробіли" для оптимізації доступу:

- `short id` (2 байти) + 2 байти padding = 4 байти.
- `int age` (4 байти).
- `double salary` (8 байтів).
- **Загалом**: 16 байтів.

14.7. Рекомендації

Ініціалізація: завжди ініціалізуйте члени структур, щоб уникнути сміттєвих значень.

Імена: використовуйте великі літери для типів структур, малі – для змінних.

Організація: групуйте пов’язані дані в структури для зрозумілості коду.

Висновки

Структури в С дозволяють ефективно групувати дані, спрощуючи їхнє використання в програмах. Ця тема закладає основу для подальшого вивчення роботи зі структурами через функції та вказівники.

Запитання та завдання для самоконтролю

1. Що таке структура в мові С?
2. Для чого потрібні структури в програмуванні?
3. Чому використання окремих змінних менш ефективне, ніж структура?
4. Як оголосити структуру в мові С?
5. Що таке члени структури і як їх називають інакше?
6. Чому крапка з комою в кінці оголошення структури є обов’язкова?
7. Як визначити змінну типу структури в С?
8. Як відрізнити ім’я структури від імені змінної за конвенцією?
9. Як отримати доступ до члена структури за допомогою оператора вибору?
10. Що станеться, якщо не ініціалізувати члени структури?

11. Як ініціалізувати структуру за допомогою списку ініціалізації?
12. Поясніть що відбувається з членами структури, які не вказані в списку ініціалізації.
13. Як вручну присвоїти значення членам структури після її оголошення?
14. У чому різниця між структурами та масивами в С?
15. Як порівняти два екземпляри структури за одним із полів?
16. Що таке вкладена структура і як її оголосити?
17. Як ініціалізувати вкладену структуру за допомогою списку ініціалізації?
18. Як отримати доступ до члена вкладеної структури?
19. Чому розмір структури може бути більшим за суму розмірів її членів?
20. Що таке вирівнювання пам’яті і як воно впливає на структури?
21. Як дізнатися розмір структури в С?
22. Як уникнути ініціалізації сміттєвих значень у структурі?
23. Як створити масив структур і заповнити його вручну?
24. Як обчислити середнє значення поля для кількох структур без функцій?
25. Які переваги дає групування даних у структури?
26. Як перевірити, чи всі члени структури ініціалізовані коректно?
27. Як визначити порядок членів у структурі для оптимального розміру?
28. Як створити структуру для зберігання даних про книгу (назва, автор, рік)?
29. Як об’єднати дві структури в одну без використання функцій?
30. Як скопіювати одну структуру в іншу вручну без функцій?

15. Використання структур для роботи з функціями. Використання вказівників для роботи зі структурами

Структури в мові С є потужним інструментом для групування даних, а їхнє поєднання з функціями та вказівниками значно розширює можливості програмування. У цій темі розглянемо, як передавати структури у функції, повертати їх із функцій, а також як використовувати вказівники для ефективної роботи зі структурами.

Мета теми – показати, як структури інтегруються з функціями для обробки складних даних і як вказівники дозволяють оптимізувати доступ до структур, економити пам'ять і змінювати їх вміст у функціях. Розглянемо далі передачу за значенням та за адресою, повернення структур, а також типові сценарії використання вказівників зі структурами.

15.1. Передача структур у функції

Передача за значенням

У С структури можна передавати у функції за значенням, коли копія структури передається в параметр функції. Це зручно для невеликих структур, але може бути неефективним для великих через копіювання даних.

Приклад 15.1. Передача структури за значенням

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void printPoint(struct Point p) {
    printf("Точка: (%d, %d)\n", p.x, p.y);
}
```

```
int main() {
    struct Point p1 = {3, 4};
    printPoint(p1);
    return 0;
}
```

```
Виведення:
Точка: (3, 4)
```

- Копія **p1** передається в **printPoint**, оригінал не змінюється.

Переваги:

- **Безпека:** функція не може змінити оригінальну структуру.
- **Простота:** не потрібні вказівники.

Недоліки:

- Копіювання великих структур **займає час і пам'ять**.

Передача за адресою

Для великих структур або коли потрібно змінити оригінал, передаємо вказівник на структуру. Оператор **->** використовується для доступу до членів через вказівник.

Приклад 15.2. Передача структури за адресою

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void movePoint(struct Point *p, int dx, int dy)
{
    p->x += dx;
    p->y += dy;
}

int main() {
```

```
    struct Point p1 = {3, 4};
    printf("До: (%d, %d)\n", p1.x, p1.y);
    movePoint(&p1, 2, -1);
    printf("Після: (%d, %d)\n", p1.x, p1.y);
    return 0;
}
```

```
Виведення:
До: (3, 4)
Після: (5, 3)
```

- Вказівник дозволяє змінити оригінальну структуру.

Переваги:

- **Ефективність:** передається лише адреса (4 або 8 байтів).
- **Зміна:** функція може модифікувати дані.

Недоліки:

- **Потрібна перевірка на NULL** для безпеки.

Приклад 15.3. Перевірка на NULL

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

void printPointSafe(const struct Point *p) {
    if (p == NULL) {
        printf("Помилка: NULL\n");
        return;
    }
    printf("Точка: (%d, %d)\n", p->x, p->y);
}

int main() {
    struct Point p1 = {1, 2};
    printPointSafe(&p1);
```

```
    printPointSafe(NULL);
    return 0;
}
Виведення:
Точка: (1, 2)
Помилка: NULL
```

- **const** захищає структуру від змін.

15.2. Повернення структур із функцій

Повернення за значенням

Функція може повертати структуру, що дозволяє передавати кілька значень як єдине ціле.

Приклад 15.4. Повернення структури

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

struct Point createPoint(int x, int y) {
    struct Point p;
    p.x = x;
    p.y = y;
    return p;
}

int main() {
    struct Point p = createPoint(5, 7);
    printf("Точка: (%d, %d)\n", p.x, p.y);
    return 0;
}
```

```
Виведення:
Точка: (5, 7)
```

Приклад 15.5. Обчислення середини відрізка

```
#include <stdio.h>

struct Point {
    int x;
    int y;
};

struct Point midpoint(struct Point p1, struct
Point p2) {
    struct Point mid;
    mid.x = (p1.x + p2.x) / 2;
    mid.y = (p1.y + p2.y) / 2;
    return mid;
}

int main() {
    struct Point p1 = {2, 4}, p2 = {6, 8};
    struct Point mid = midpoint(p1, p2);
    printf("Середина: (%d, %d)\n", mid.x,
mid.y);
    return 0;
}
```

```
Виведення:
Середина: (4, 6)
```

Повернення вказівника на структуру

Для динамічних структур повертаємо вказівник, але важливо уникати повернення адрес локальних змінних.

Приклад 15.6. Повернення динамічної структури

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

struct Point {
    int x;
```

```
    int y;
};

struct Point* newPoint(int x, int y) {
    struct Point *p = (struct
Point*)malloc(sizeof(struct Point));
    if (p == NULL) return NULL;
    p->x = x;
    p->y = y;
    return p;
}

int main() {
    struct Point *p = newPoint(3, 5);
    if (p != NULL) {
        printf("Точка: (%d, %d)\n", p->x, p->y);
        free(p);
    }
    return 0;
}
```

```
Виведення:
Точка: (3, 5)
```

Помилка: повернення локальної структури

```
struct Point* badReturn() {
    struct Point p = {1, 2};
    return &p; // Помилка: p знищується після
завершення
}
```

15.3. Використання вказівників зі структурами

Доступ через вказівники

Вказівники дозволяють ефективно працювати зі структурами, особливо в масивах або динамічній пам’яті.

Приклад 15.7. Масив структур із вказівниками

```
#include <stdio.h>

struct Employee {
    int id;
    double salary;
};

void increaseSalary(struct Employee *emp, double
amount) {
    emp->salary += amount;
}

int main() {
    struct Employee staff[] = {{1, 30000.0}, {2,
35000.0}};
    increaseSalary(&staff[0], 5000.0);
    printf("ID: %d, Зарплата: %.2f\n",
staff[0].id, staff[0].salary);
    return 0;
}
```

```
Виведення:
ID: 1, Зарплата: 35000.00
```

Модифікація через вказівники

Вказівники дозволяють функціям змінювати оригінальні структури.

Приклад 15.8. Оновлення структури

```
#include <stdio.h>

struct Student {
    char name[50];
    int grade;
};
```

```
void updateGrade(struct Student *s, int
newGrade) {
    s->grade = newGrade;
}

int main() {
    struct Student alice = {"Alice", 85};
    printf("До: %s, Оцінка: %d\n", alice.name,
alice.grade);
    updateGrade(&alice, 90);
    printf("Після: %s, Оцінка: %d\n",
alice.name, alice.grade);
    return 0;
}
```

```
Виведення:
До: Alice, Оцінка: 85
Після: Alice, Оцінка: 90
```

Передача масиву структур

Масиви структур передаються як вказівники на перший елемент.

Приклад 15.9. Обробка масиву структур

```
#include <stdio.h>

struct Employee {
    int id;
    double salary;
};

void printStaff(struct Employee *staff, int
size) {
    for (int i = 0; i < size; i++) {
        printf("ID: %d, Зарплата: %.2f\n",
staff[i].id, staff[i].salary);
    }
}
```

```
int main() {
    struct Employee staff[] = {{1, 30000.0}, {2,
35000.0}};
    printStaff(staff, 2);
    return 0;
}
```

```
Виведення:
ID: 1, Зарплата: 30000.00
ID: 2, Зарплата: 35000.00
```

15.4. Вкладені структури та вказівники

Доступ до вкладених структур

Вказівники дозволяють зручно працювати з вкладеними структурами.

Приклад 15.10. Вкладена структура

```
#include <stdio.h>

struct Date {
    int day, month, year;
};

struct Person {
    char name[50];
    struct Date birth;
};

void printPerson(struct Person *p) {
    printf("%s, Народження: %d/%d/%d\n", p-
>name, p->birth.day,
           p->birth.month, p->birth.year);
}

int main() {
```

```
    struct Person john = {"John", {15, 6,
1995}};
    printPerson(&john);
    return 0;
}
```

```
Виведення:
John, Народження: 15/6/1995
```

Зміна вкладених полів

```
#include <stdio.h>

struct Date {
    int day, month, year;
};

struct Person {
    char name[50];
    struct Date birth;
};

void setBirthYear(struct Person *p, int year) {
    p->birth.year = year;
}

int main() {
    struct Person john = {"John", {15, 6,
1990}};
    setBirthYear(&john, 1995);
    printf("Новий рік: %d\n", john.birth.year);
    return 0;
}
```

```
Виведення:
Новий рік: 1995
```

15.5. Практичні приклади

Приклад 15.11. Управління списком студентів

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

struct Student {
    char name[50];
    int grade;
};

void addStudent(struct Student *list, int
*count, const char *name, int grade) {
    strcpy(list[*count].name, name);
    list[*count].grade = grade;
    (*count)++;
}

int main() {
    struct Student students[10];
    int count = 0;
    addStudent(students, &count, "Alice", 85);
    addStudent(students, &count, "Bob", 90);
    for (int i = 0; i < count; i++) {
        printf("%s: %d\n", students[i].name,
students[i].grade);
    }
    return 0;
}
```

```
Виведення:
Alice: 85
Bob: 90
```

Приклад 15.12. Обчислення відстані між точками

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

```
struct Point {
    double x, y;
};

double distance(const struct Point *p1, const
struct Point *p2) {
    double dx = p1->x - p2->x;
    double dy = p1->y - p2->y;
    return sqrt(dx * dx + dy * dy);
}

int main() {
    struct Point p1 = {0.0, 0.0}, p2 = {3.0,
4.0};
    printf("Відстань: %.2f\n", distance(&p1,
&p2));
    return 0;
}
```

```
Виведення:
Відстань: 5.00
```

15.6. Рекомендації

Передача: використовуйте вказівники для великих структур, щоб уникнути копіювання.

Безпека: перевіряйте вказівники на **`NULL`** перед розіменуванням.

Читабельність: ініціалізуйте структури під час оголошення, якщо можливо.

Пам'ять: звільняйте динамічно виділені структури через **`free`**.

Висновки

Структури в поєднанні з функціями та вказівниками забезпечують ефективний спосіб роботи зі складними даними в C. Приклади демонструють передачу, повернення та модифікацію

структур, підкреслюючи переваги вказівників для оптимізації та гнучкості.

Запитання та завдання для самоконтролю

1. Як передати структуру у функцію за значенням в мові C?
2. Які переваги має передача структури в функцію за значенням?
3. Вкажіть недоліки передачі структури в функцію за значенням
4. Як передати структуру в функцію за адресою?
5. У чому різниця між операторами `.` і `->` при роботі зі структурами?
6. Як функція може змінити оригінальну структуру через вказівник?
7. Як захистити структуру від змін у функції при передачі за адресою?
8. Що станеться, якщо передати в функцію нульовий вказівник на структуру?
9. Як перевірити вказівник на `NULL` перед використанням у функції?
10. Як повернути структуру з функції за значенням?
11. Назвіть переваги повернення структури з функції за значенням.
12. Як повернути вказівник на структуру з функції?
13. Чому не можна повертати вказівник на локальну структуру з функції?
14. Як правильно повернути динамічно виділену структуру з функції?
15. Як передати масив структур у функцію?
16. Як обробити масив структур у функції за допомогою вказівника?
17. Як змінити поле вкладеної структури через вказівник у функції?
18. Як передати вкладену структуру в функцію за адресою?
19. Як повернути вкладену структуру з функції?

20. Як обчислити відстань між двома точками, переданими в функцію як структури?
21. Як функція може додати елемент до масиву структур через вказівник?
22. Як оптимізувати передачу великих структур у функцію?
23. Як ініціалізувати структуру всередині функції та повернути її?
24. Як передати кілька структур у функцію для порівняння їхніх полів?
25. Як модифікувати масив структур у функції без копіювання?
26. Як уникнути витоків пам’яті при поверненні динамічної структури з функції?
27. Як функція може обчислити середню зарплату працівників із масиву структур?
28. Як передати вказівник на структуру в іншу функцію для подальшої обробки?
29. Як реалізувати сортування масиву структур за певним полем у функції?
30. Як перевірити коректність даних структури у функції перед обробкою?

16. Тип union та його використання

У темі розглянемо тип union у мові програмування С, який дозволяє зберігати різні типи даних в одній і тій самій області пам'яті. Далі вивчимо його основні характеристики, синтаксис, особливості використання та наведемо практичні приклади, що демонструють, як і коли застосовувати union у реальних задачах.

16.1. Поняття типу union

Union (об'єднання) – це спеціальний тип даних у мові С, який дозволяє зберігати різні типи даних у спільній області пам'яті. На відміну від структур (**`struct`**), де кожен член займає власну область пам'яті, у **`union`** усі члени використовують одну й ту саму область. Розмір **`union`** дорівнює розміру його найбільшого члена.

Основні характеристики:

- **Спільна пам'ять:** Усі члени **`union`** зберігаються в одній області пам'яті.
- **Розмір:** Визначається розміром найбільшого члена.
- **Використання:** Економить пам'ять, коли потрібно зберігати лише один із кількох типів даних.

Синтаксис:

```
union Ім'я_об'єднання {
    тип1 член1;
    тип2 член2;
    // ...
};
```

Приклад:

```
union Data {
    int i;
    float f;
    char str[20];
```

```
};
```

У цьому прикладі **union Data** може зберігати або ціле число (**int**), або дійсне число (**float**), або масив символів (**char[20]**), але лише один із них активний у певний момент.

16.2. Особливості використання union

- **Спільна пам'ять:** Зміна одного члена перезаписує пам'ять, що впливає на інші члени.
- **Ініціалізація:** Можна ініціалізувати лише один член (за замовчуванням – перший).
- **Доступ:** Використовується оператор крапки (.), як у структурах.

Приклад:

```
union Data data;
data.i = 10;
printf("data.i: %d\n", data.i);
data.f = 3.14;
printf("data.f: %f\n", data.f);
```

Примітка: Після присвоєння **data.f = 3.14** значення **data.i** стає невизначеним.

16.3. Порівняння union і struct

Характеристика	**union**	**struct**
Пам'ять	Спільна для всіх членів	Окрема для кожного члена
Розмір	Розмір найбільшого члена	Сума розмірів усіх членів (з вирівнюванням)
Використання	Один активний тип	Усі члени доступні одночасно
Зміна значень	Впливає на інші члени	Не впливає на інші члени

union підходить для економії пам'яті, коли потрібен лише один тип даних, а **struct** – для зберігання кількох значень одночасно.

16.4. Практичні приклади використання union

Нижче наведено приклади реального використання union, які демонструють його переваги в задачах прикладної математики, інженерії та програмування.

Приклад 16.1: Зберігання різних типів числових даних

У задачах обчислювальної математики часто потрібно працювати з цілими, дійсними або комплексними числами. **Union** дозволяє зберігати ці типи в одній структурі.

```
#include <stdio.h>

union Number {
    int integer;
    float real;
    struct {
        float real_part;
        float imag_part;
    } complex;
};

int main() {
    union Number num;

    num.integer = 42;
    printf("Ціле число: %d\n", num.integer);

    num.real = 3.14;
    printf("Дійсне число: %f\n", num.real);

    num.complex.real_part = 1.0;
    num.complex.imag_part = 2.0;
    printf("Комплексне число: %f + %fi\n",
num.complex.real_part, num.complex.imag_part);
```

```
    return 0;
}
```

Пояснення: `union Number` зберігає або ціле, або дійсне, або комплексне число, економлячи пам’ять.

Приклад 16.2: Різні одиниці виміру

У фізичних або інженерних задачах корисно зберігати значення в різних одиницях виміру (метри, фути, кілометри).

```
#include <stdio.h>

union Distance {
    float meters;
    float feet;
    float kilometers;
};

int main() {
    union Distance dist;

    dist.meters = 100.0;
    printf("Метри: %f\n", dist.meters);

    dist.feet = 328.084;
    printf("Фути: %f\n", dist.feet);

    dist.kilometers = 0.1;
    printf("Кілометри: %f\n", dist.kilometers);

    return 0;
}
```

Пояснення: Лише одна одиниця виміру активна, що зменшує використання пам’яті.

Приклад 16.3: Оптимізація пам’яті в структурах даних

У вузлах дерева або списку union дозволяє зберігати різні типи даних, наприклад, число або рядок.

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

union NodeData {
    int int_val;
    char str_val[20];
};

struct Node {
    int type;  // 0 - int, 1 - char[]
    union NodeData data;
};

int main() {
    struct Node node1, node2;

    node1.type = 0;
    node1.data.int_val = 100;

    node2.type = 1;
    strcpy(node2.data.str_val, "Hello, Union!");

    if (node1.type == 0) printf("Node1: %d\n",
node1.data.int_val);
    if (node2.type == 1) printf("Node2: %s\n",
node2.data.str_val);

    return 0;
}
```

Пояснення: Поле **`type`** вказує, який тип даних активний у union.

Приклад 16.4: Універсальний контейнер

Union може слугувати для створення контейнера, що підтримує різні типи об'єктів (низькорівневий поліморфізм).

```
#include <stdio.h>
union Object {
    int int_obj;
    float float_obj;
    char char_obj;
};

struct Container {
    int type;  // 0 - int, 1 - float, 2 - char
    union Object obj;
};

void printObject(struct Container cont) {
    switch (cont.type) {
        case 0: printf("Ціле: %d\n",
cont.obj.int_obj); break;
        case 1: printf("Дійсне: %f\n",
cont.obj.float_obj); break;
        case 2: printf("Символ: %c\n",
cont.obj.char_obj); break;
        default: printf("Невідомий тип\n");
    }
}

int main() {
    struct Container cont1 = {0, {.int_obj =
42}};
    struct Container cont2 = {1, {.float_obj =
3.14}};
    struct Container cont3 = {2, {.char_obj =
'A'}};

    printObject(cont1);
    printObject(cont2);
```

```
    printObject(cont3);

    return 0;
}
```

Пояснення: Контейнер адаптується до різних типів завдяки union.

Приклад 16.5: Робота з бінарними даними

Union зручно використовувати для аналізу порядку байтів (little-endian або big-endian).

```
#include <stdio.h>
union Endian {
    int int_val;
    char bytes[4];
};

int main() {
    union Endian data;
    data.int_val = 0x12345678;

    printf("Байти: ");
    for (int i = 0; i < 4; i++) printf("%02x ",
(unsigned char)data.bytes[i]);
    printf("\n");

    if (data.bytes[0] == 0x78) printf("Little-
endian\n");
    else if (data.bytes[0] == 0x12) printf("Big-
endian\n");

    return 0;
}
```

Пояснення: **union** дозволяє інтерпретувати число як масив байтів.

16.5. Рекомендації щодо використання union

- **Обережність:** Зміна одного члена впливає на інші, тому працюйте лише з активним членом.
- **Теги:** Використовуйте додаткове поле для вказівки активного члена.
- **Економія пам'яті:** Застосовуйте union, коли потрібно зберігати один із кількох типів.
- **Низький рівень:** Використовуйте для роботи з бінарними даними або поліморфізму.
- **Уникайте помилок:** Не читайте неактивні члени, щоб уникнути невизначеної поведінки.

Висновки

Тип **`union`** – це ефективний інструмент для економії пам'яті та роботи з різними типами даних у спільній області. Він особливо корисний у задачах оптимізації, низькорівневого програмування та математичного моделювання. У наступному розділі ми розглянемо тип **`enum`** для створення іменованих констант.

Запитання та завдання для самоконтролю

1. Що таке union у мові C?
2. Як оголосити union із кількома членами?
3. У чому різниця між union і struct?
4. Як отримати доступ до члена union?
5. Що стається з іншими членами при зміні одного?
6. Як union економить пам'ять?
7. Наведіть приклад роботи з бінарними даними за допомогою union.
8. Як комбінувати union зі структурою?
9. Які ризики використання union?
10. У яких випадках краще обрати union замість struct?

17. Перелічувальний тип (enum)

У мові програмування С перелічувальний тип (**enum**) є зручним способом визначення набору іменованих констант. Це дозволяє розробникам працювати з фіксованими значеннями, такими як дні тижня, стани програми чи категорії, у більш зрозумілій формі. Використання **enum** підвищує читабельність коду та зменшує ймовірність помилок, замінюючи "магічні числа" осмисленими назвами.

17.1. Поняття перелічувального типу (enum)

Перелічувальний тип (**enum**, від англ. **enumeration** – перелік) – це спеціальний тип даних у С, який дозволяє визначити набір іменованих цілочислових констант. Кожна константа в **enum** має унікальне ім'я та пов'язане з ним ціле число. За замовчуванням значення починаються з **0** і зростають на **1** для кожної наступної константи, але їх можна налаштувати вручну.

Основні особливості:

- **Іменовані константи:** Замість чисел використовуються зрозумілі імена.
- **Цілочислова природа:** Кожна константа є цілим числом, що дозволяє використовувати її в операціях.
- **Обмежений набір значень:** Змінні типу **enum** можуть набувати лише значення з визначеного переліку.

17.2. Синтаксис оголошення enum

Для створення перелічувального типу використовується ключове слово **enum**, після якого вказується ім'я перелічення та список констант у фігурних дужках.

Базовий синтаксис

```
enum Ім'я_перелічення {
    КОНСТАНТА1,
    КОНСТАНТА2,
    // ...
```

```
};
```

Приклад:

```
enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};
```

У цьому прикладі **MONDAY** має значення **0**, **TUESDAY** – **1**, і так до **SUNDAY** – **6**.

Присвоєння конкретних значень

Можна явно вказати значення для констант:

```
enum State {
    IDLE = 0,
    RUNNING = 1,
    PAUSED = 2,
    STOPPED = 3
};
```

17.3. Використання enum

Після оголошення перелічувального типу можна створювати змінні цього типу та присвоювати їм значення з переліку.

Оголошення змінної

```
enum Day today;
today = WEDNESDAY;
```

Використання в умовних операторах

```
if (today == WEDNESDAY) {
    printf("Сьогодні середа!\n");
}
```

Використання в циклах

```
for (enum Day d = MONDAY; d <= SUNDAY; d++) {
    printf("День: %d\n", d);
}
```

Примітка: Оскільки **enum** є цілочисловим типом, його можна використовувати в арифметичних операціях, але це не рекомендується для збереження зрозумілості коду.

17.4. Практичні приклади використання enum

Приклад 17.1: Дні тижня

```
#include <stdio.h>

enum Day {
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY,
    SUNDAY
};

int main() {
    enum Day today = WEDNESDAY;
    printf("Сьогодні: %d\n", today);
    // Виведе: 2
    return 0;
```

```
}
```

Приклад 17.2: Стани програми

```
#include <stdio.h>

enum State {
    IDLE = 0,
    RUNNING = 1,
    PAUSED = 2,
    STOPPED = 3
};

void printState(enum State s) {
    switch (s) {
        case IDLE:
           printf("Очікування\n"); break;
        case RUNNING:
           printf("Виконання\n"); break;
        case PAUSED:
           printf("Пауза\n"); break;
        case STOPPED:
           printf("Зупинено\n"); break;
        default:
           printf("Невідомий стан\n");
    }
}

int main() {
    enum State current = RUNNING;
    printState(current); // Виведе: Виконання
    return 0;
}
```

Приклад 17.3: Кольори

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

```
enum Color {
    RED = 0xFF0000,
    GREEN = 0x00FF00,
    BLUE = 0x0000FF
};

void setColor(enum Color c) {
    printf("Колір: 0x%06X\n", c);
}

int main() {
    setColor(BLUE); // Виведе: Колір: 0x0000FF
    return 0;
}
```

17.5. Порівняння enum з іншими типами даних

- **Звичайні константи** (`const int`): Менш зручні, оскільки не групують значення логічно.
- **Макроси** (`#define`): Не забезпечують типової безпеки та менш читабельні.
- **Масиви**: Зберігають дані, тоді як `enum` лише визначає набір значень.

Переваги enum

- Логічна організація констант.
- Зручність у `switch`-операторах.
- Поліпшена читабельність.

17.6. Рекомендації щодо використання enum

- Використовуйте зрозумілі імена для констант.
- Уникайте "магічних чисел", замінюючи їх `enum`.
- Обмежуйте значення змінних за допомогою `enum` для зменшення помилок.
- Не зловживайте арифметичними операціями з `enum`.

Висновки

Перелічувальний тип (**enum**) у C є ефективним інструментом для роботи з фіксованими наборами значень. Він спрощує код, робить його більш читабельним і допомагає уникнути помилок, пов'язаних із некоректними значеннями. Завдяки **enum** програмісти можуть створювати логічно структуровані програми з мінімальними зусиллями.

Питання для самоконтролю

1. Що таке перелічувальний тип (**enum**) у мові C?
2. Як оголосити **enum** із кількома константами?
3. Як можна присвоїти конкретні значення константам у **enum**?
4. Як створити змінну типу **enum** і присвоїти їй значення?
5. Як використовувати **enum** у **switch**-операторі?
6. Наведіть приклад **enum** для місяців року.
7. Як **enum** поліпшує читабельність коду?
8. У чому переваги **enum** перед макросами **#define**?
9. Чи можна порівнювати значення **enum** у циклах? Як це зробити?
10. Які рекомендації щодо іменування констант у **enum**?

18. Динамічне виділення пам’яті

У С мові є можливість виділяти пам’ять динамічно – під час виконання програми. Це потрібно, коли розмір даних невідомий на етапі компіляції або може змінюватися під час роботи. На відміну від статичної або автоматичної пам’яті (локальних змінних), динамічна пам’ять виділяється в спеціальній області – **купі** (heap). Після виділення область залишається зайнятою до явного звільнення. Динамічне виділення забезпечує гнучкість: ми можемо створювати масиви та структури довільного розміру, керувати їх тривалістю життя і змінювати розмір у процесі роботи. Однак це вимагає суворої обережності: виділену пам’ять потрібно звільняти після використання, щоб уникнути помилок (витоків пам’яті, «висячих» вказівників тощо).

18.1. Стек і купа

Стек (stack) і **купа (heap)** – це дві основні області пам’яті, які використовує програма. Стек використовується для автоматичних (локальних) змінних і параметрів функцій. Стек – це область пам’яті, над якою ви не маєте ніякого контролю. У нього записуються змінні та інформація, що створюється у результаті виклику функцій. Коли функція закінчує роботу, то вся інформація про її виклик та її змінні видаляються з стеку автоматично. Іншими словами, локальні змінні живуть лише в межах функції і зникають при виході з неї. У купі навпаки – пам’ять виділяється самостійно за допомогою функцій стандартної бібліотеки. Купа – це та область пам’яті, над якою програмісти мають безпосередній контроль. Пам’ять тут виділяється у результаті виклику функції malloc [21].

Виділені з купи об’єкти (масиви, структури тощо) зберігаються доти, поки не звільняться викликом **`free`**. Купа зазвичай набагато більша за стек і не має суворих обмежень на розмір об’єктів. Але вона фрагментується і повільніше виділяється, тому слід ретельно керувати ресурсами.

18.2. Функції malloc, calloc, realloc і free

У С стандартні функції динамічного розподілу пам'яті оголошені в заголовку **<stdlib.h>**. Найпоширеніші з них:

- **malloc(size)** – виділяє безперервний блок пам'яті розміром **size** байт. Повертає вказівник на початок блоку або **NULL** при помилці (наприклад, якщо недостатньо пам'яті). Виділені дані **не ініціалізуються** – можуть містити «сміття» з пам'яті.
- **calloc(n, size)** – виділяє пам'ять для **n** елементів по **size** байт, подібно до **malloc(n * size)**, але заповнює її нулями. Корисно, коли потрібна нульова ініціалізація.
- **realloc(ptr, new_size)** – змінює розмір уже виділеного блоку, на який вказує **ptr**. Може збільшувати або зменшувати обсяг; при цьому може виділити новий блок і скопіювати туди дані, якщо розширити старий не вдасться. В результаті повертається новий указівник (старий при цьому стає недійсним). Якщо **realloc** повертає **NULL** (помилка), попередній блок залишається незміненим, і про нього слід потурбуватися (звільнити чи обробити).
- **free(ptr)** – звільняє раніше виділений блок пам'яті (з **malloc/calloc/realloc**), адреса якого збережена у **ptr**. Після виклику **free** сам вказівник стає «висячим» (посилається на вже звільнену область). **Важливо:** звільняти слід саме ті області, що були виділені; подвійний виклик **free(ptr)** для одного блоку чи **free** для нерозподіленого вказівника призводять до помилок і невизначеної поведінки.

Синтаксис використання цих функцій простий. Наприклад:

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

int main() {
```

```
    int *array = malloc(10 * sizeof *array);
    // виділили пам'ять для 10 цілих
    if (array == NULL) {
        fprintf(stderr, "Помилка виділення
пам'яті\n");
        return 1;
    }
    // Використання масиву array[0]..array[9]...
    free(array);
    // звільняємо пам'ять після використання
    return 0;
}
```

Тут **malloc** повертає **int*** після виділення **10*sizeof(int)** байт. Ми перевіряємо на **NULL** – гарна практика, бо без пам'яті програма не мусить продовжувати роботу. Пам'ять звільняємо викликом **free(array)**. Аналогічно працює **calloc** (але пам'ять буде нульовою) і **realloc** (розмір блоку змінюється).

Типові помилки: не звільнити пам'ять (витік), звільнити «чужий» чи вже звільнений вказівник, забути перевірити **NULL**. Усі ці ситуації ведуть до помилок виконання. Наприклад, після **free(p);** будь-яке ***p** (або другий **free(p)**) призведе до невизначеної поведінки.

18.3. Виділення пам'яті під масиви

Динамічно виділені масиви реалізуються за допомогою вказівників та **malloc/calloc**.

Одновимірні масиви:

```
int n = 5;
int *arr = malloc(n * sizeof *arr);
// пам'ять для n цілих
if (arr == NULL) { /* обробка помилки */ }
for(int i = 0; i < n; i++) {
    arr[i] = i * 2; // запис у масив
}
// ...
```

```
free(arr);  // звільнення
```

Тут виділяємо блок пам'яті розміром **n*sizeof(int)** та використовуємо його як звичайний масив.

Двовимірні масиви: оскільки С не має вбудованої підтримки для динамічних двомірних масивів, використовується один з двох підходів:

- ***Масив вказівників на рядки:***

```
int rows = 3, cols = 4;
int **mat = malloc(rows * sizeof *mat);
// масив покажчиків на рядки
for(int i = 0; i < rows; i++) {
    mat[i] = malloc(cols * sizeof *mat[i]);
// кожен рядок
}
// приклад заповнення
for(int i = 0; i < rows; i++)
    for(int j = 0; j < cols; j++)
        mat[i][j] = i + j;
// ...
// звільнення: спочатку кожен рядок,
// потім сам масив покажчиків
for(int i = 0; i < rows; i++) {
    free(mat[i]);
}
free(mat);
```

- ***«Суцільний» масив:*** виділити один великий блок під всі елементи і додатково – масив покажчиків на рядки:

```
int *data = malloc(rows * cols * sizeof *data);
int **mat = malloc(rows * sizeof *mat);
for(int i = 0; i < rows; i++) {
    mat[i] = data + i*cols;
}
// тепер mat[i][j] коректно індексує елемент
...
free(mat);
free(data);
```

Перший варіант простіший у реалізації і частіше вживається; другий іноді використовують для суміщення в пам’яті, але потребує ретельнішої звільнення (два окремих виклики **free**).

18.4. Виділення пам’яті під структури

Так само можна виділяти пам’ять для структури або масиву структур. Наприклад:

```
typedef struct {
    char name[50];
    int age;
} Person;

int main() {
    Person *p = malloc(sizeof *p);
    // пам'ять під одну структуру
    if (!p) return 1;  // перевіримо NULL
    // ініціалізація
    strcpy(p->name, "Alice");
    p->age = 30;
    // використання p->...
    free(p);  // звільнення пам'яті
    return 0;
}
```

Або динамічний масив структур:

```
int n = 10;
Person *group = malloc(n * sizeof *group);
// масив з n структур Person
if (!group) { /* обробка */ }
for(int i = 0; i < n; i++) {
    // group[i].name = ...; group[i].age = ...;
}
free(group);
```

Зверніть увагу: у **malloc** використовується **sizeof *p** або **sizeof(Person)**, що знижує ризик помилки при зміні типу. Пам’ять для структури звільняємо як завжди через **free**.

18.5. Повторне виділення пам’яті з realloc

Функція **realloc(ptr, newSize)** змінює розмір блоку, початково виділеного **ptr**. Наприклад, щоб розширити масив:

```
int *arr = malloc(2 * sizeof *arr);
// спочатку 2 елементи
arr[0] = 10; arr[1] = 20;
// хочемо збільшити розмір
int *temp = realloc(arr, 5 * sizeof *arr);
if (temp == NULL) {
    // обробка помилки: arr все ще дійсний,
    // але ми вирішуємо звільнити
    free(arr);
    // ...
} else {
    arr = temp;  // arr вказує на новий блок
(можливо змінений)
    // тепер можна використовувати
    // arr[2], arr[3], arr[4]
}
```

При виклику **realloc** блок може переміститися в інше місце пам’яті, і повертається нова адреса. Якщо успішне виконання, старий вказівник більше не треба звільняти (його простір уже оброблено внутрішньо). Якщо **realloc** повертає **NULL**, початковий блок **arr** усе ще доступний і потребує уваги (звільнення або повторної спроби). Щоб не втратити адресу пам’яті у випадку помилки, рекомендовано спочатку зберігати результат в тимчасовий вказівник (як показано вище). Не можна писати просто **arr = realloc(arr, ...)** без перевірки, інакше втратиться старий блок при помилці.

18.6. Звільнення пам’яті й уникнення витоків

Після того, як ви більше не потребуєте виділеного блоку, обов’язково викликайте **free(ptr)**. Інакше це спричинить витік пам’яті. Витік пам’яті (memory leak) – процес неконтрольованого зменшення обсягу вільної пам’яті, пов’язаний з поми-

лками програм, які не звільняють пам'ять. Іншими словами, якщо забути викликати **free**, виділена пам'ять залишиться зайнятою до завершення програми. При великій кількості таких витоків програма споживатиме дедалі більше пам'яті; зрештою система може видати помилку виділення (або повністю зависнути).

Тому для кожного виклику **malloc/calloc/realloc** має бути відповідний **free**. Наприклад, для динамічного масиву **int *arr = malloc(n*sizeof *arr);** виклик має вигляд **free(arr);** після того, як масив більше не потрібен. У випадку двовимірного масиву потрібно викликати **free** для кожного підмасиву та для масиву вказівників, як показано вище. Також рекомендується одразу після **free(ptr)** встановлювати **ptr = NULL**, щоб уникнути випадкового повторного використання цього вказівника (перевірка **if(ptr)** згодом захистить від помилки).

18.7. Типові помилки при роботі з динамічною пам'яттю

Висячі вказівники: після звільнення пам'яті **free(ptr)**, вказівник **ptr** стає «висячим» (dangling) – він вказує на вже звільнену область. Спроба розіменувати такий вказівник призведе до непередбачуваних результатів. Наприклад:

```
int *p = malloc(sizeof *p);
*p = 42;
free(p);
// *p = 7;    // помилка: p висячий,
              // цього не можна робити
p = NULL;     // робочий спосіб уникнути
              // висячого вказівника
```

Подвійне звільнення: якщо викликати **free(ptr)** двічі для одного й того самого блоку, це знову призведе до неви-

значеної поведінки. Стежте, щоб у програмі було не більше одного **`free`** для кожного **`malloc`**.

Витоки пам'яті: вже згадувалося, що несанкціоноване невивільнення призводить до витоку. Навіть короткочасний витік під час великої ітерації (наприклад, у циклі) може швидко заповнити оперативну пам'ять.

Ініціалізація: виділена через **`malloc`** пам'ять не ініціалізована, тому дані у ній будуть невизначеними. Після **`malloc`** рекомендується одразу присвоювати значення або використовувати **`calloc`**, якщо потрібно ініціалізувати нулями.

Неправильний розмір: типова помилка – неправильно обчислена кількість байтів. Наприклад, **`malloc(n)`** замість **`malloc(n*sizeof(int))`** призведе до помилкового обсягу. Щоб уникнути такого, звичайно пишуть **`malloc(n * sizeof *ptr)`**, що автоматично правильно розраховує розмір об'єкта.

18.8. Рекомендації з безпечного використання

Перевірка результату: завжди перевіряйте, чи не повернув **`malloc`** або **`realloc`** **`NULL`**.

Використання `sizeof *ptr`: при виділенні пам'яті використовувати **`sizeof *ptr`** замість явного типу (**`sizeof(int)`**), це зменшує ризик помилки.

Встановлення NULL: після **`free(ptr)`** бажано писати **`ptr = NULL;`**, щоб подальша випадкова спроба **`free`** чи розіменування не викликала проблему.

Розподіл і звільнення парою: завжди підтримувати баланс: якщо виділяєте пам'ять, плануйте, де і коли її звільнятимете.

Уникати «витоків» у циклах: наприклад, якщо використовуєте malloc або realloc у циклі, подбайте про звільнення тимчасових буферів перед наступною ітерацією.

Інструменти перевірки: використовувати програми-аналізатори (див. нижче) для знаходження витоків або неправильних доступів.

18.9. Приклади

Приклад 18.1. Створення динамічного масиву чисел із введеним розміром.

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

int main() {
    int n, *arr;
    printf("Введіть розмір масиву: ");
    scanf("%d", &n);

    arr = (int*)malloc(n * sizeof(int));
    if (arr == NULL) {
        printf("Помилка виділення пам'яті!\n");
        return 1;
    }

    for (int i = 0; i < n; i++) {
        arr[i] = i + 1;
    }
    printf("Масив: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Приклад 18.2. Зміна розміру масиву за допомогою realloc

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

int main() {
```

```
    int *arr = (int*)malloc(3 * sizeof(int));
    if (arr == NULL) {
        printf("Помилка!\n");
        return 1;
    }

    arr[0] = 1; arr[1] = 2; arr[2] = 3;
    printf("Початковий масив: %d %d %d\n",
arr[0], arr[1], arr[2]);

    arr = (int*)realloc(arr, 5 * sizeof(int));
    if (arr == NULL) {
        printf("Помилка!\n");
        return 1;
    }

    arr[3] = 4; arr[4] = 5;
    printf("Розширений масив: ");
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Приклад 18.3. Ініціалізація нулями з calloc

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

int main() {
    int n = 5;
    int *arr = (int*)calloc(n, sizeof(int));
    if (arr == NULL) {
        printf("Помилка виділення пам'яті!\n");
        return 1;
    }
```

```
    printf("Масив, ініціалізований нулями: ");
    for (int i = 0; i < n; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

Приклад 18.4. Динамічний масив із можливістю розширення (push_back)

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

int main() {
    int capacity = 4; // початкова ємність
    int size = 0; // поточна кількість елементів
    int *arr = malloc(capacity * sizeof *arr);
    if (!arr) return 1;

    // Додаємо 8 елементів у масив
    for(int i = 1; i <= 8; i++) {
        if (size == capacity) {
            int newCap = capacity * 2;
            int *tmp = realloc(arr, newCap *
sizeof *arr);
            if (tmp == NULL) {
                // помилка розширення
                free(arr);
                return 1;
            }
            arr = tmp;
            capacity = newCap;
        }
        arr[size++] = i * 10;
        // додаємо новий елемент
```

```
    }
    // Вивід масиву
    for(int i = 0; i < size; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");

    free(arr);
    return 0;
}
```

У цьому прикладі починаємо з ємності vfcb 4. Кожного разу, коли масив заповнюється (**size == capacity**), використовуємо **realloc**, щоб подвоїти ємність. При успіху нова пам'ять копіює старі дані, а змінна **arr** оновлюється на новий блок. Такий підхід дозволяє додавати невідомо скільки елементів, поступово збільшуючи розмір.

Приклад 18.5. Створення простого зв'язного списку

Розглянемо створення простого односпрямованого списку з двома елементами:

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

struct Node {
    int data;
    struct Node *next;
};

int main() {
    // Створення голови списку
    struct Node *head = malloc(sizeof *head);
    if (!head) return 1;
    head->data = 10;
    head->next = NULL;

    // Додавання другого вузла
```

```
    struct Node *second = malloc(sizeof
*second);
    if (!second) {
        free(head);
        return 1;
    }
    second->data = 20;
    second->next = NULL;
    head->next = second;

    // Виведення списку
    struct Node *cur = head;
    while (cur != NULL) {
        printf("%d -> ", cur->data);
        cur = cur->next;
    }
    printf("NULL\n");

    // Звільнення списку
    cur = head;
    while (cur != NULL) {
        struct Node *tmp = cur;
        cur = cur->next;
        free(tmp);  // звільняємо поточний вузол
    }
    return 0;
}
```

Тут вручну виділяємо пам’ять для кожного вузла (**head** і **second**), зберігаємо значення і посилання на наступний. Після використання проходимося по списку та викликаємо **free** для кожного елемента (зберігаючи перед переміщенням **cur = cur->next**, щоб не втратити доступ).

18.10. Інструменти для перевірки витоків пам’яті

Для контролю за динамічною пам’яттю існують спеціальні засоби. Одним із найвідоміших є **Valgrind** на Linux. Valgrind – інструмент для налагодження використання пам’яті, виявлення витоків пам’яті. Запустивши програму через Valgrind,

можна отримати звіт про всі виклики **malloc/free**, неініціалізований доступ і витоки. На Windows є аналоги (наприклад, інструменти Visual Studio або сторонні перевіряльники). Рекомендується прогнати програму через такі інструменти після розробки, щоб переконатися у відсутності витоків та невірних доступів.

Висновки

Динамічне виділення пам'яті є потужним механізмом, що надає програмісту гнучкість у керуванні обсягом даних під час виконання. Завдяки функціям **malloc**, **calloc**, **realloc** і **free** можна виділяти пам'ять різного розміру, створювати структури та масиви невизначених розмірів, а також реалізовувати складні структури даних (списки, дерева тощо). Однак це накладає обов'язки: слід ретельно перевіряти успішність виділення, звільняти пам'ять після використання та уникати помилок («висячих» вказівників, подвійного звільнення тощо). Дотримання рекомендацій (перевірка **NULL**, присвоєння **NULL** після **free**, правильний розрахунок розміру) значно знижує ризик помилок. Тому важливо практикуватися на прикладах, а також користуватися інструментами, що знаходять витоки, щоб виробити правильні навички роботи з динамічною пам'яттю.

Запитання та завдання для самоконтролю

1. Яка різниця між стеком і купою у контексті розподілу пам'яті?
2. Як працюють функції **malloc** і **calloc**? У чому їхня відмінність?
3. Наведіть приклад виділення пам'яті для динамічного масиву та його звільнення.
4. Як розподілити пам'ять під двовимірний масив за допомогою **malloc**? Як його звільнити?
5. Що робить **realloc**? Які небезпеки його використання?
6. Що таке «висячий» вказівник і як цього уникнути?
7. Чим небезпечні витоки пам'яті (memory leaks)? Як їх виявити?

8. Наведіть рекомендації з безпечного використання динамічної пам’яті (перевірка **`malloc`**, присвоєння **`NULL`** після **`free`** тощо).
9. У чому полягає різниця між режимами реального та захищеного доступу до пам’яті?
10. Як саме перехід у захищений режим поліпшує стабільність і безпеку операційної системи?

19. Робота з файлами

Робота з файлами у мові С забезпечує можливість зберігати дані між запусками програми та обмінюватися інформацією з іншими програмами. Файли можуть бути текстовими (читабельними людиною) або двійковими (збереженими у вигляді сирих байтів). Для роботи з файлами у стандартній бібліотеці С (**<stdio.h>**) використовується тип **FILE** і відповідні функції. У цій темі розглянемо, як відкривати/закривати файли, читати й записувати дані різними способами (побайтово, рядками, форматовано), а також розберемо двійкові файли та маніпуляції положенням у файлі.

19.1. Оголошення та відкриття файлів

Файл у С представлений покажчиком на структуру типу **FILE**, який зазвичай називають *файловим потоком*. Щоб почати роботу з файлом, потрібно його **відкрити** за допомогою функції **fopen**:

```
FILE *f = fopen("data.txt", "r");
```

Перший аргумент – ім'я файлу (рядок), другий – режим відкриття (рядок). Режим задає, чи відкриваємо файл на читання, запис тощо.

Поширені режими:

- **"r"** – відкрити файл тільки для читання. Якщо файл не існує, **fopen** поверне **NULL**.
- **"w"** – відкрити файл тільки для запису. Якщо файл уже існує, він буде **обрізаний** (вміст видалений) і перезаписаний. Якщо не існує – створюється новий.
- **"a"** – відкрити файл для **допису** (append). Якщо файл не існує – створюється. У цьому режимі нові дані завжди додаються в кінець.
- **"r+"**, **"w+"**, **"a+"** – подібно до **"r"**, **"w"**, **"a"**, але також дозволяють читання і запис у файлі (тобто двосторонній доступ).

- До кожного з цих режимів можна додати **"b"** для роботи з файлами у двійковому режимі (наприклад, **"rb"**, **"wb"**, **"ab"**). У бінарному режимі дані читаються/записуються без текстового перетворення (корисно, наприклад, у Windows, де зазвичай **"\n"** перетворюється у **"\r\n"** у текстовому режимі). У Linux це перетворення зазвичай непомітне, але все ж краще явно вказувати **"b"**, якщо потрібні двійкові дані.

Якщо **fopen** не може відкрити файл (наприклад, через відсутність файлу в режимі **"r"** або через брак прав), вона повертає **NULL**. Тому завжди перевіряйте успішність:

```
FILE *f = fopen("input.txt", "r");
if (f == NULL) {
    printf("Не вдалося відкрити файл\n");
    exit(1);
}
```

Після завершення роботи з файлом його потрібно закрити викликом **fclose**:

```
fclose(f);
```

Це звільняє ресурси і гарантує запис буферизованих даних у файл. Важливо не забувати закривати файли, щоб уникнути витоку ресурсів.

19.2. Читання та запис текстових файлів

Для посимвольного читання з файлу використовується функція **fgetc**, а для запису символів – **fputc**. Приклад читання всіх символів з файлу:

```
FILE *f = fopen("data.txt", "r");
int ch;
while ((ch = fgetc(f)) != EOF) {
    // читаємо символ, поки не кінець файлу
    putchar(ch);
    // виводимо символ на екран
}
```

```
fclose(f);
```

Пояснення: `fgetc(f)` повертає прочитаний символ як int, або макроконстанту `EOF`, якщо досягнуто кінця файлу. Тому змінну для прийому маємо оголосити як `int`. Функція `fputc(ch, f)` аналогічно записує символ `ch` у файл `f`. Щоб працювати не по одному символу, а цілими рядками, є функції `fgets` і `fputs`. Функція `fgets(buffer, size, f)` читає з файлу `f` рядок (або фрагмент) довжиною до `size-1` символів і ставить у масив `buffer`. Приклад читання рядків:

```
char line[100];
FILE *f = fopen("data.txt", "r");
while (fgets(line, sizeof(line), f) != NULL) {
    printf("Зчитано рядок: %s", line);
    // fgets зберігає '\n' наприкінці,
    // якщо вистачає розміру
}
fclose(f);
```

Тут `fgets` повертає `NULL`, коли дійде кінця файлу або виникне помилка. Функція `fputs(str, f)` записує рядок `str` у файл `f`:

```
FILE *f = fopen("out.txt", "w");
fputs("Привіт, світ!\n", f);
// записує текст у файл
fclose(f);
```

Існують також `fputs` та `fgets` варіанти з `printf`-подібним форматуванням.

19.3. Форматований ввід/вивід у файли (fprintf, fscanf)

Якщо потрібно читати або записувати різні типи даних у фіксованому форматі (числа, рядки з форматами), використовують `fprintf` і `fscanf`. Вони аналогічні `printf` та `scanf`, але з першим аргументом – файловим потоком.

Приклад запису числа:

```
int a = 42;
FILE *f = fopen("nums.txt", "w");
fprintf(f, "Число: %d\n", a);
// форматований запис у файл
fclose(f);
```

Читання числа з файлу:

```
int x;
FILE *f = fopen("nums.txt", "r");
if (fscanf(f, "Число: %d", &x) == 1) {
    printf("Прочитали число %d\n", x);
}
fclose(f);
```

Форматний ввід дає можливість комбінувати різні специфікатори (наприклад, `%d`, `%f`, `%s` тощо) і одночасно зчитувати/записувати цілий ряд даних. Наприклад, щоб зчитати три числа з файлу:

```
int a, b, c;
FILE *f = fopen("data.txt", "r");
// Припускаємо, у файлі три числа, наприклад:
"10 20 30"
fscanf(f, "%d %d %d", &a, &b, &c);
fclose(f);
printf("%d %d %d\n", a, b, c);
```

Порада: Використовуйте `fscanf` обережно: при неправильному форматі або кінці файлу вона може повернути число зчитаних полів, що менше очікуваного. Тому перевіряйте її результат (кількість успішно прочитаних елементів).

19.4. Робота з двійковими файлами (fread, fwrite)

Двійкові файли дозволяють зберігати структури даних у сирому вигляді байтів. Це зручно для швидкого збереження чисел, масивів чи структур, проте файл стає нечитаємим для людини. Двійковий режим відкривається з літерою `"b"` у режимі: `"rb"`, `"wb"`, `"ab"`, ...

Основні функції:

- **fwrite(ptr, size, count, f)** – записати **count** елементів по **size** байт з буферу **ptr** у файл **f**;
- **fread(ptr, size, count, f)** – прочитати count елементів по **size** байт із файлу у буфер **ptr**. Повертають кількість прочитаних або записаних елементів.

Приклад запису масиву цілих у двійковий файл:

```
#include <stdio.h>

int main() {
    int arr[5] = {1, 2, 3, 4, 5};
    FILE *f = fopen("data.bin", "wb");
    // відкрити для запису у двійковому режимі
    if (f == NULL) return 1;
    // fwrite записує масив arr
    // як 5 елементів типу int
    size_t written = fwrite(arr, sizeof(int), 5,
f);
    if (written != 5) {
        printf("Помилка запису у файл\n");
    }
    fclose(f);
    return 0;
}
```

Тут **fwrite** бере з **arr** байти і записує їх у файл. Щоб прочитати ще раз:

```
#include <stdio.h>

int main() {
    int arr[5];
    FILE *f = fopen("data.bin", "rb");
    // відкрити файл на читання
    if (f == NULL) return 1;
    size_t read = fread(arr, sizeof(int), 5, f);
    if (read != 5) {
        printf("Помилка читання з файлу\n");
    }
```

```
    // Тепер arr[] містить прочитані числа
    for (int i = 0; i < 5; i++) {
        printf("%d ", arr[i]);
    }
    printf("\n");
    fclose(f);
    return 0;
}
```

Приклад читання/запису структури в бінарному файлі:

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

struct Student {
    int id;
    char name[20];
    float score;
};

int main() {
    struct Student s = { 1, "Ivan Petrenko",
95.5f };
    // Запис структури у файл
    FILE *f = fopen("stud.bin", "wb");
    if (f == NULL) return 1;
    fwrite(&s, sizeof(struct Student), 1, f);
    fclose(f);

    // Читання структури з файлу
    struct Student t;
    f = fopen("stud.bin", "rb");
    if (f == NULL) return 1;
    fread(&t, sizeof(struct Student), 1, f);
    fclose(f);

    printf("ID: %d, Name: %s, Score: %.1f\n",
t.id, t.name, t.score);
    return 0;
```

```
}
```

У цьому прикладі напряму записуємо вміст пам'яті структури. Зауважте: такий двійковий запис залежить від макета пам'яті (вирівнювання) та може не бути сумісним між різними системами. Рядок **name** зберігається без термінального нуля, якщо масив повністю заповнений, тож слід бути обережним. Для текстових даних у структурі варто передбачити достатній розмір буферу.

19.5. Переміщення по файлу: fseek, ftell, rewind

Коли потрібно прочитати або записати дані з довільного місця файлу, використовують функції для зміни поточної позиції у файлі:

- **fseek(f, offset, whence)** – перемістити потік **f** на **offset** байтів відносно точки **whence**. whence може бути:
  - **SEEK_SET** – початок файлу;
  - **SEEK_CUR** – поточна позиція;
  - **SEEK_END** – кінець файлу.

  Повертає 0 при успіху або невід'ємне значення при помилці.
- **ftell(f)** – повертає поточну позицію у файлі (кількість байтів від початку) як **long**. Повертає **-1L** при помилці.
- **rewind(f)** – переміщує позицію на початок файлу (еквівалентно **fseek(f, 0, SEEK_SET)**).

Приклад використання **fseek** і **ftell**:

```
FILE *f = fopen("data.txt", "r");
if (f == NULL) return 1;
// Йдемо в кінець файлу,
// щоб дізнатися його розмір
fseek(f, 0, SEEK_END);
long size = ftell(f);
printf("Розмір файлу: %ld байт\n", size);
// Повертаємося на початок
```

```
// для подальшого читання
rewind(f);
fclose(f);
```

Це може бути корисно, наприклад, за потреби резервувати достатньо пам’яті або рухатися у файлі на кроки. При обчисленні офсету пам’ятайте, що **`offset`** може бути від’ємним (для переміщення назад від кінця або поточної позиції). **Увага:** при роботі з текстовими файлами у деяких системах можуть бути додаткові зміни позиції через перекодування рядків (крім Windows у бінарному режимі). Тому для повної точності у двійкових файлах (**`"rb"`**, **`"wb"`**) **`fseek`** та **`ftell`** працюють без невідчутних змін.

19.6. Перевірка помилок і завершення роботи з файлом

При роботі з файлами потрібно контролювати наявність помилок:

- Перевірка успішного відкриття: переконайтеся, що **`fopen`** не повернула **`NULL`** перед використанням файлу.
- **`feof(f)`** – повертає ненульове значення, якщо у файлі досягнуто кінця (EOF). Зазвичай використовується в циклах читання, хоча краще перевіряти результат читання функцій (як у прикладах вище).
- **`ferror(f)`** – повертає ненульове значення, якщо сталася помилка при останній операції з файлом. Після виявлення помилки можна викликати **`clearerr(f)`**, щоб очистити стан помилки.
- Після запису даних стандартні C-функції буферизують вивід, тому перед закриттям варто переконатися, що всі дані записані. Функція **`fclose(f)`** автоматично зливає (flush) буфер. Якщо потрібно примусово записати дані без закриття, використовуйте **`fflush(f)`**.
- Перевіряйте значення повернення **`fread`**/**`fwrite`**: вони повертають кількість записаних/прочитаних елементів. Наприклад:

```
size_t rc = fread(buffer, 1, n, f);
if (rc < n && ferror(f)) {
    printf("Помилка при читанні файлу\n");
}
```

- Завжди **закривайте** файли викликом **`fclose(f)`**. Це гарантує, що всі ресурси звільнені, а останні дані записані. Незакритий файл може призвести до втрати даних або блокування файлу іншою програмою.

19.7. Рекомендації

- **Перевірка результатів.** Завжди перевіряйте, чи **`fopen`** успішно відкрив файл, а **`fread/fwrite`** прочитали/записали потрібну кількість даних. Це запобігає некоректній роботі при помилках вводу/виводу.
- **Вибір режиму.** Обирайте режим відкриття (**`"r"`**, **`"w"`**, **`"a"`** тощо) відповідно до завдання: для безпечного читання використовуйте **`"r"`**, для запису (перезапису) – **`"w"`**, для додавання даних – **`"a"`**. Для двійкових файлів не забувайте додавати **`"b"`**.
- **Закриття файлів.** Після завершення роботи з файлом обов'язково викликайте **`fclose`**. Ігнорування цього може призвести до витоків ресурсів або неповного запису даних.
- **Читання рядків.** При використанні **`fgets`**, якщо рядок може бути довшим за буфер, треба обробляти частини або використовувати більший буфер. **`fgets`** зберігає символ нового рядка **`\n`**, якщо є місце.
- **Позиціювання.** Якщо використовуєте **`fseek/ftell`**, точно вказуйте базову точку (**`SEEK_SET/CUR/END`**). При зміні напряму читання/запису у файлі може знадобитися перевірка ferror.
- **Двійкові файли.** Пам'ятайте, що двійкові файли не можна читати як текст. Запис структур або масивів робіть з **`<stddef.h>`** – **`sizeof`**. Переконайтеся, що

структура не містить покажчиків (інакше збережеться лише адреса, а не самі дані).

- **Ефективність.** Робота з файлами зазвичай повільніша ніж робота з пам'яттю, тому буферизуйте дані за можливості. Використовуйте **`fread/fwrite`** для великих блоків даних замість побайтового **`fgetc/fputc`** – це швидше.

Висновки

У цій темі розглянуто, як у мові C здійснювати ввід/вивід інформації у файли, відкривати і закривати файли за допомогою **`fopen/fclose`**, читати та записувати дані текстовими та двійковими методами. Базові функції (**`fgetc`**, **`fputc`**, **`fgets`**, **`fputs`**) дозволяють обробляти потік по символах або рядках, а **`fprintf/fscanf`** – виконувати форматований ввід/вивід (з числовими форматами). Двійкові функції (**`fread`**, **`fwrite`**) дають змогу зберігати структури та масиви у файли у сирому вигляді для швидкого доступу. Ми також розібралися з різними режимами відкриття файлів: **`"r"`**, **`"w"`**, **`"a"`** тощо, та їх двійковими аналогами. Маніпуляції позицією у файлі (**`fseek`**, **`ftell`**) корисні для випадків випадкового доступу до даних. Нарешті, підкреслили важливість перевірки помилок (**`feof`**, **`ferror`**) і навели рекомендації щодо надійної роботи з файлами (перевірка повернених значень, закриття файлів).

Запитання та завдання для самоконтролю

1. Що таке тип **`FILE`** і для чого він використовується?
2. Як за допомогою fopen відкрити файл тільки для читання? Як для запису? Поясніть, що станеться, якщо файл не знайдено в режимі **`"r"`**.
3. Яка різниця між режимами **`"r"`**, **`"w"`** та **`"a"`** при відкритті файлу?
4. Як прочитати з файлу всі його символи по одному? Яку роль відіграє **`EOF`**?
5. Як записати у файл рядок тексту і окремо – одне число з форматуванням?

6. Чим відрізняються функції **fprintf/fscanf від fwrite/fread**?
7. Як за допомогою **fgets** прочитати цілий рядок з файлу? Що потрібно пам'ятати про буфер?
8. Що означають константи **SEEK_SET**, **SEEK_CUR** та **SEEK_END** у функції **fseek**?
9. Які функції перевіряють кінець файлу або наявність помилок (**EOF**)?
10. Наведіть приклад запису/читання однієї структури у двійковому файлі за допомогою **fwrite/fread**.

20. Макроси

Макроси – це іменовані фрагменти коду, які обробляються препроцесором перед основною компіляцією програми. Директиви препроцесора починаються із символу **#** і задають правила текстової заміни або умовної обробки коду. Вони дозволяють визначати константи, включати файли, проводити умовну компіляцію та інше. Наприклад, директива **#define** замінює ідентифікатор на задане значення або вираз у всьому коді, а **#include** додає до програми вміст іншого файлу (заголовка). Макроси особливо корисні для визначення констант і спрощення повторюваного коду на етапі препроцесору.

20.1. Основні директиви препроцесора

- **#define** – дозволяє визначити макрос: він замінює кожне входження імені макроса на заданий текст (тіло макроса) при препроцесорній обробці. Зазвичай використовується для створення констант і простих функціональних макросів.
- **#include** – директива включення: вставляє в програму вміст іншого файлу. Існують два синтаксиси: **#include <файл>** шукає заголовки в системних директоріях, а **#include "файл"** – спочатку в поточній папці, потім у системних. Зазвичай підключаються заголовкові файли з деклараціями.
- **#undef** – видаляє раніше визначений макрос. Використовується, якщо потрібно звільнити ім'я макроса або перепризначити його іншим значенням. Наприклад, щоб перевизначити макрос, спочатку його "анульовують" через **#undef**, а потім задають нове визначення.

Кожна директива **#define** або **#undef** працює на рівні препроцесора і не породжує код під час виконання. Визначення макросів глобальні для файлу (та інших файлів, куди цей файл включено). Наприклад, у програмі можна так задати константи:

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

```
#define TRUE 1              // Визначення цілих
констант
#define FALSE 0
#define COUNTRY "INDIA"    // Рядкова константа

int main() {
    printf("Country: %s\n", COUNTRY);
    printf("TRUE as int: %d\n", TRUE);
    return 0;
}
```

Після препроцесора всі входження **TRUE**, **FALSE**, **COUNTRY** будуть замінені на відповідні значення.

20.2. Параметризовані макроси (макрофункції)

Макроси можуть мати параметри, подібно до функцій. Такі макроси часто називають макрофункціями. Приклад:

```
#define SUM(a, b) ((a) + (b))
#define SQUARE(x) ((x) * (x))
```

Коли в коді зустрічається виклик макроса з аргументами, препроцесор вставляє його тіло з підставленими значеннями аргументів. Наприклад, **SUM(10, 20)** розгорнеться в **(10) + (20)**. Це дозволяє записати "функції" без виклику (без накладних витрат на перехід у функцію). Проте такі макроси **зберігають текстову природу**: обчислення виразу відбувається без типізації, і кожне входження аргументів підставляється буквально.

Наприклад:

```
#include <stdio.h>

#define SQUARE(x) ((x) * (x))
// Макрос для обчислення квадрата

int main() {
    int a = 3;
    int b = SQUARE(a + 1);
    // Розгортається як ((3 + 1) * (3 + 1))
```

```
    printf("b = %d\n", b);     // b = 16
    return 0;
}
```

У цьому прикладі макрос **SQUARE(x)** обчислює квадрат аргументу. Зверніть увагу на оточення аргументів і всього виразу дужками – це захищає від помилок із пріоритетом операцій. Приклад: якби ми визначили **#define SQUARE(x) (x * x)**, виклик **SQUARE(a+1)** дав би несподіваний результат через порядок дій (експансія без дужок дала б **a + 1 * a + 1**, що дорівнює **a + a + 1**, а не **(a+1)*(a+1)**). Параметризовані макроси використовуються, наприклад, для обгортки невеликих функцій задля пришвидшення виконання. Функціональні макроси «переписують малі функції за допомогою макросів, щоб заощадити час виконання».

Приклади:

```
#define MAX(a, b) ((a) > (b) ? (a) : (b))
// Макрос для взяття максимуму
#define ABS(x) ((x) < 0 ? -(x) : (x))
// Макрос абсолютного значення
```

Усе це відбувається ще до компіляції – препроцесор просто підставляє текст замість макроса.

20.3. Умовна компіляція

Директиви умовної компіляції дозволяють включати або виключати фрагменти коду залежно від визначених макросів або значень констант. Основні директиви такого типу:

#ifdef MACRO – включає блок коду, якщо макрос **MACRO** визначений.

#ifndef MACRO – включає блок, якщо макрос **MACRO** не визначений.

#if expr / #elif expr / #else / #endif – перевіряють константний вираз. Якщо **expr** (константний вираз) невід'ємний (не дорівнює нулю), то код між **#if** і наступним **#elif** чи **#else** обирається для компіляції. Можна вказати

кілька умов через **#elif**, а **#else** – гілка на випадок, якщо жодна з умов не спрацювала.

Приклад використання **#ifdef** та **#ifndef**:

```
#include <stdio.h>

// Визначаємо макрос для демонстрації
#define COUNTRY "INDIA"

int main() {
    // Блок виконається,
    // оскільки COUNTRY визначено
    #ifdef COUNTRY
        printf("Macro COUNTRY is defined.\n");
    #endif

    // Блок виконається,
    // бо STATE ще не визначено
    #ifndef STATE
        printf("Macro STATE is not defined.
Defining it now.\n");
        #define STATE "PATNA"
    #endif

    printf("STATE: %s\n", STATE);
    return 0;
}
```

У цьому прикладі використано **#ifdef/#ifndef** для перевірки наявності макросів. Зауважте, що кожна така директива обов'язково завершується **#endif**. Це обрамлення дозволяє ізолювати умовний код. Зокрема, такі директиви часто застосовують для запобігання повторному включенню заголовків (інклудів). Наприклад, у файлі **myfile.h** можна написати:

```
#ifndef _MYFILE_H_
#define _MYFILE_H_
// тут оголошення функцій, структур, тощо
#endif
```

При першому включенні (**#include**) макрос **_MYFILE_H_** не визначений, тому він визначається всередині файлу, і вміст підключається. При наступних включеннях той самий макрос уже визначений, тому блок пропускається. Отже, вміст заголовка завжди вставляється лише один раз, уникнувши помилок подвійного оголошення. Альтернативно для цього використовують директиву **#pragma once**, яка гарантує одноразове включення файлу незалежно від умінь препроцесора.

Для перевірки значень макросів застосовують також **#if**, **#elif** та **#else**. Наприклад:

```
#define DEBUG 1
#define VERSION 2

#if DEBUG
    // цей код буде включений у збірку,
    // якщо DEBUG != 0
#endif

#if VERSION == 1
    // перша версія
#elif VERSION == 2
    // друга версія
#else
    // інша версія
#endif
```

Таким чином, директиви умови керують тим, яка частина коду потрапляє в остаточну компіляцію. Як зазначено у документації Microsoft, директива **#if** разом з **#elif**, **#else** та **#endif** «контролює компіляцію окремих частин файлу». Кожен **#if** обов'язково має відповідний **#endif**, і можна вкладати одні умовні блоки в інші.

20.4. Директива #pragma

Директива **#pragma** служить для передачі компілятору спеціальних команд, які виходять за рамки стандартного синтаксису C. Це компілятор-залежна директива: її поведінка

може відрізнятися в різних компіляторах. Синтаксис простий – **#pragma** після чого йде ім'я або список параметрів. Наприклад, часто використовують:

- **#pragma once** – вказує, що цей заголовочний файл потрібно включати лише один раз. Ефективно замінює традиційні захисні блоки **#ifndef/#define;**
- **#pragma pack** – керує вирівнюванням структур у пам'яті. Наприклад, **#pragma pack(1)** може вказати упакувати структуру без вирівнювання (компілятор сприймає це по-різному);
- **#pragma warning** або компілятор-залежні попередження – дозволяє включати/відключати певні повідомлення компілятора.

Приклад використання:

```
#pragma once
// Далі вміст заголовного файлу,
// він інклюдиться тільки один раз у програмі
void foo();
```

У цьому випадку **#pragma once** забезпечує, що інклуд-контент буде оброблено лише один раз у рамках всієї програми. Інші **#pragma** застосовують у проєктних налаштуваннях (вимикання певних ворнінгів, налаштування сегментів пам'яті тощо), але їх підтримка залежить від компілятора. В офіційній документації GNU сказано: «**#pragma** – спеціальна директива, яка передає додаткову інформацію компілятору».

20.5. Розширені можливості макросів: ## та

Макроси в C підтримують **склеювання токенів** і **перетворення в рядок**.

- Оператор **#** (**stringizing**) перед аргументом макроса перетворює переданий аргумент на рядкову константу. Наприклад, макрос:

```
#define STR(x) #x
```

```
STR(Hello World)  // Розгортається в "Hello
World"
```

Оператор # перед аргументом виступає як оператор перетворення в рядок, що поміщає відповідний аргумент у подвійні лапки. Тобто текст аргумента буде «взятий у лапки» в результаті препроцесу.

- Оператор **##** (**token-pasting**) дозволяє склеювати два токени макроса в один. Якщо у визначенні макроса наводити **##** між двома фрагментами, то під час розгортання ці фрагменти зіллються в єдиний токен. Наприклад:

```
#define CONCAT(a, b) a##b
int xy = 30;
printf("%d", CONCAT(x, y));
// CONCAT(x, y) -> xy
```

У цьому прикладі **CONCAT(x,y)** після препроцесу стає **xy**, оскільки оператор **##**. Таке склеювання використовується, наприклад, для формування імен змінних за частинами або для створення унікальних імен макросів під час їх розширення.

Ці можливості дозволяють створювати динамічні іменування або повідомлення:

```
#define MAKE_NAME(prefix, name) prefix##_##name
#define TO_STRING(x) #x

int main() {
    int var = 100;
    printf("Name is %s\n", TO_STRING(var));
    // Виведе: Name is var
    printf("Value is %d\n", var);

    // Приклад CONCAT:
    int value = 50;
    printf("Skleinyi name: %d\n", MAKE_NAME(var,
value));  // Збере var_value
    return 0;
```

```
}
```

Тут **TO_STRING(var)** перетворює символи **var** в рядок **"var"**, а **MAKE_NAME(var, value)** збирає токени у **var_value**. Оператор **##** дозволяє конкатенувати токени… коли макрос розгортається, два токени по обидва боки від **'##'** зливаються в один. Користь у тому, що ім'я змінної можна будувати з частин, а також створювати умовно унікальні ідентифікатори.

20.6. Типові помилки при використанні макросів

Макроси та їх розгортання мають низку складнощів:

- **Відсутність дужок і порядок операцій.** Якщо аргументи чи все тіло макросу не взяти в круглі дужки, можливі непередбачувані результати. Наприклад, макрос #define ceil_div(x,y) (x + y - 1)/y при виклику **ceil_div(b & c, sizeof(int))** розгорнеться так, що операції виконаються не за планом через пріоритети. Як показано в документації GCC, щоб уникнути проблем із пріоритетами, потрібно взяти й аргументи, і весь вираз у дужки, наприклад:

```
#define CEIL_DIV(x, y) (((x) + (y) - 1) / (y))
```

Це гарантує, що обчислення відбувається коректно.

- **Подвійне обчислення аргументів (побічні ефекти).** Якщо параметризований макрос містить вираз, аргументи можуть підставлятися кілька разів, що приведе до повторного виконання. Наприклад, макрос

```
#define MIN(a, b) ((a) < (b) ? (a) : (b))
```

та виклик **MIN(x++, y)** спричинять те, що **x++** розгорнеться двічі в різних частинах умовного виразу. Як попереджає документація GCC, у цьому випадку **x++** виконається двічі, і результат може бути неочікуваним – такий макрос називають небезпечним. Щоб уникнути цього, в макросі доводиться самостійно зберігати аргументи в тимчасових змінних (рішення з

GNU-розширеннями) або просто бути обережним з аргументами, що мають побічні ефекти.

- **Засипання крапкою з комою.** Якщо макрос розгортається в складений оператор, необхідно продумати, як він впишеться в місце виклику. Якщо такий макрос викликати з крапкою з комою після нього перед **`else`**, це може порушити синтаксис. Рекомендована практика – обгортати багатооператорні макроси конструкцією **`do { ... } while(0)`**, щоб кожен виклик закінчувався одним оператором.

Загалом, при визначенні макросів слід пам'ятати про пріоритети операцій і кількість оцінок аргументів. Наприклад, модифікуючи макрос зі сторінки документації GCC, їх рекомендують брати і аргументи і усе тіло в дужки, щоб уникнути несподіваних результатів. А при використанні макросів із побічними ефектами слід усвідомлювати, що один аргумент може бути замінений кілька разів.

20.7. Практичні приклади

Приклад 20.1. Математичні макроси (обгортки)

Макроси зручно використовувати як обгортки математичних виразів для скорочення коду.

```
#include <stdio.h>

#define SQUARE(x) ((x) * (x))
// Квадрат числа
#define MAX(a, b) (((a) > (b)) ? (a) : (b))
// Максимум з двох

int main() {
    int v = 5, w = 3;
    printf("SQUARE(v+1) = %d\n",
SQUARE(v+1)); // 36, обчислюється (5+1)*(5+1)
    printf("MAX(v, w) = %d\n", MAX(v,
w));     // 5
    return 0;
```

```
}
```

У цьому прикладі **SQUARE(v+1)** розгортається як **((5+1)*(5+1))**, а **MAX(v, w)** – як **((5 > 3) ? 5 : 3)**. Макроси дозволяють уникнути написання тих самих формул багаторазово. При цьому слід бути уважним: якщо замість **((x)*(x))** написати **x*x**, виклик **SQUARE(1+2)** дасть неправильний результат (помилка пріоритетів). Як зазначено раніше, додавання дужок розв'язує цю проблему.

Приклад 20.2. Захист заголовкових файлів від повторного включення

Один із найпоширеніших прийомів – використовувати макроси для захисту файлів заголовків від повторного підключення. Це робиться так:

```
#ifndef MY_HEADER_H
#define MY_HEADER_H

void foo();
// Оголошення функції
// (або визначення структур і т. ін.)

#endif  // MY_HEADER_H
```

При першому включенні **_MY_HEADER_H** не визначений, тому вміст підключається і макрос встановлюється. При наступних **#include** цього файлу **_MY_HEADER_H** вже буде визначеним, і весь блок опиниться під охороною **#ifndef**, тобто не включиться вдруге. Такий прийом дозволяє обмежити багаторазове завантаження файлу й уникнути помилок із подвійною декларацією. Альтернативно цього можна досягти директивою **#pragma once** на початку файлу:

```
#pragma once
// ті самі декларації або визначення
```

#pragma once забезпечує включення вмісту лише один раз, незалежно від системи включення. Це еквівалент традиційному блокові охорони, але простіше записати.

Приклад 20.3. Платформозалежні налаштування

Макроси широко використовують для умовної компіляції **платформозалежного коду**. Наприклад:

```
#include <stdio.h>

#ifdef _WIN32
  // Код для Windows (WinAPI,
  // бібліотечні виклики Windows)
  #define OS_NAME "Windows"
#elif defined(__linux__)
  // Код для Linux
  #define OS_NAME "Linux"
#elif defined(__APPLE__)
  // Код для macOS
  #define OS_NAME "macOS"
#else
  #define OS_NAME "Unknown OS"
#endif

int main() {
    printf("Operating System: %s\n", OS_NAME);
    return 0;
}
```

У цьому коді вибір залежить від того, який макрос (**_WIN32**, **__linux__**, **__APPLE__** тощо) визначено компілятором. Такможна мати єдиний вихідний код, а під час компіляції активувати лише ту частину, що стосується поточної ОС. Так роблять і для архітектури (32/64 біт тощо). Ці директиви («частина препроцесора») дозволяють контролювати, які частини коду будуть скомпільовані на основі певних умов.

Приклад 20.4. Макроси для відлагодження (DEBUG)

Часто використовують макрос **DEBUG** для ввімкнення/вимкнення відлагоджувальних повідомлень. Наприклад:

```
#include <stdio.h>

#ifdef DEBUG
  #define LOG(msg) printf("DEBUG: %s\n", (msg))
#else
  #define LOG(msg) ((void)0)
  // порожній макрос, нічого не робить
#endif

int main() {
    LOG("Starting program");
    // буде виведено тільки якщо DEBUG визначено
    printf("Working...\n");
    LOG("Ending program");
    return 0;
}
```

Якщо перед компіляцією визначити **-DDEBUG** (або написати **#define DEBUG 1**), то **LOG("...")** розгортається у виклик **printf**, і ви бачите вивід. Якщо **DEBUG** не визначено, **LOG** стане порожнім макросом (нічого не виводить). Це простий спосіб додати діагностичний код, який включається лише під час налагодження програми.

Висновки

Макроси в мові C – це потужний механізм препроцесора для текстової заміни й умовної обробки коду. Вони дозволяють визначати іменовані константи, писати макрофункції, включати файли та виконувати умовну компіляцію. Макроси також можуть бути використані для реалізації різних шаблонних конструкцій (склеювання імен, формування рядків тощо). Основні директиви – **#define**, **#include**, **#undef** та умовні (**#ifdef**, **#ifndef**, **#if**, **#else**, **#elif**, **#endif**) – описують базовий

функціонал препроцесора. Для складних завдань існують розширення **#pragma**, оператори **#/##** та інші.

Важливо пам'ятати про потенційні помилки макросів: відсутність дужок при їх визначенні і повторне обчислення аргументів можуть призвести до несподіваної поведінки. Наприклад, GCC у документації наголошує, що макроси потрібно обгортати дужками, щоб уникнути проблем із пріоритетами. При правильному застосуванні макроси роблять код гнучкішим і зручнішим, проте вони залишаються суто текстовою заміною, а не функціями чи змінними під час виконання програми.

Запитання та завдання для самоконтролю

1. Що таке макрос у мові С і на якому етапі обробки коду він діє?
2. Як працює директива **#define** і чим вона відрізняється від звичайної константи?
3. Для чого служать **#ifdef** і **#ifndef**? Наведіть приклад їх використання.
4. Як забезпечити, щоб заголовковий файл підключався лише один раз?
5. Що робить оператор **#** у визначенні макроса? А оператор **##**?
6. Наведіть приклад помилки макроса, пов'язаної з відсутністю дужок або з повторним обчисленням аргументів.
7. Як за допомогою макросів можна включати код тільки в режимі налагодження (**debug**)?
8. Поясніть, що таке директива **#pragma** і наведіть приклад її використання.
9. Які переваги і недоліки макрофункцій у порівнянні зі звичайними функціями в мові С?
10. Як можна за допомогою макросів реалізувати платформонезалежний код для різних операційних систем? Наведіть приклад.

Висновки

Мова програмування С, незважаючи на свій поважний вік, залишається однією з ключових технологій у світі інформаційних технологій. Цей посібник охопив широкий спектр тем – від базових понять, таких як змінні, оператори та введення/виведення, до складніших конструкцій, таких як вказівники, структури, динамічні масиви та рекурсивні алгоритми. Кожен розділ був спрямований на формування у читача цілісного розуміння мови, її можливостей і особливостей, а також на розвиток практичних навичок програмування.

С вирізняється своєю універсальністю та ефективністю, що робить її незамінною в системному програмуванні, розробці вбудованих систем і високопродуктивних додатків. Посібник продемонстрував, як низькорівневий доступ до пам'яті через вказівники та ручне керування ресурсами дозволяють досягати максимальної продуктивності, хоча й вимагають від розробника ретельності та дисципліни. Водночас простота синтаксису та портативність С забезпечують її доступність для початківців і адаптивність до сучасних викликів.

Особливу увагу приділено практичним аспектам: численні приклади програм – від обчислення площі кола до сортування масивів і роботи з файлами – ілюструють реальне застосування теоретичних знань. Питання для самоконтролю в кінці кожного розділу допомагають закріпити матеріал і розвинути аналітичне мислення, що критично важливо для студентів прикладної математики та інших технічних спеціальностей.

Хоча сучасні мови, такі як Python чи Java, пропонують вищий рівень абстракції та автоматизації, С зберігає свою актуальність завдяки неперевершеній швидкості та контролю. Цей посібник показав, що освоєння С не лише відкриває двері до розуміння роботи комп'ютерів на низькому рівні, а й закладає міцну основу для вивчення складніших мов і технологій. У майбутньому читачі зможуть розширити отримані знання, досліджуючи об'єктно-орієнтоване програмування в С++ чи розробку вбудованих систем, де С залишається стандартом де-факто.

Отже, цей посібник є не лише навчальним ресурсом, а й дороговказом до глибшого розуміння програмування як науки й мистецтва. Він створений, щоб надихати на подальше навчання та практичне застосування C у реальних проєктах, від простих алгоритмів до складних системних рішень.

Список літератури

1. Brian W. Kernighan, Dennis M. Ritchie. C Programming Language. 2nd edition. Prentice Hall, 1988. 278 p.
2. Bryant R. E., O'Hallaron D. R. Computer Systems: A Programmer's Perspective. 3rd edition. Boston : Pearson, 2015. 1120 p.
3. Dey P., Ghosh M. Programming in C. 2nd Edition. New Delhi : Oxford University Press, 2011. 512 p.
4. Gookin D. C Programming For Dummies. 2nd edition. Hoboken : John Wiley & Sons, 2021. 464 p.
5. Gustedt J. Modern C. Shelter Island : Manning Publications, 2019. 408 p.
6. Klemens B. 21st Century C. 2nd edition. Sebastopol : O'Reilly Media, 2015. 408 p.
7. Kochan S. G. Programming in C. 4th edition. Upper Saddle River : Addison-Wesley Professional, 2014. 552 p.
8. Prinz P., Crawford T. C in a Nutshell. 2nd edition. Sebastopol : O'Reilly Media, 2016. 824 p.
9. Reese R. Understanding and Using C Pointers. Sebastopol : O'Reilly Media, 2013. 226 p.
10. Schildt H. C: The Complete Reference. 4th edition. New York : McGraw-Hill, 2000. 805 p.
11. Seacord R. C. Effective C: An Introduction to Professional C Programming. San Franciscom : No Starch Press, 2020. 272 p.
12. Shaw Z. A. Learn C The Hard Way. Upper Saddle River : Addison-Wesley Professional, 2015. 380 p.
13. Spraul V. Think Like a Programmer: An Introduction to Creative Problem Solving. San Francisco : No Starch Press, 2012. 256 p.
14. Безкоштовний онлайн-курс із C. URL : https:// www.tutorialspoint.com/cprogramming/index.htm.
15. Вільний підручник із C. URL : https://en.wikibooks.org/ wiki/C_Programming.
16. Документація C на сайті ISO. URL : https://www.iso.org/ standard/82075.html.

17. Документація стандартів C11/C17/C23. URL : https://en.cppreference.com/w/c.
18. Офіційна документація бібліотеки GNU C. URL : https://www.gnu.org/software/libc/manual/html_node/index.html.
19. C Programming Tutorial. URL : https://github.com/b09/c_resources/blob/master/C%20Programming%20Tutorial.pdf.
20. Essential C – Stanford CS Library. URL : http://cslibrary.stanford.edu/101/.
21. Introduction to C – Harvard CS50. URL : https://cs50.harvard.edu/x/2024/weeks/1/ .
22. Learn C Programming – GeeksforGeeks. URL : https://www.geeksforgeeks.org/c-programming-language/.
23. The GNU C Compiler (GCC). URL : https://gcc.gnu.org/.

Текстова хвиля з символів ASCII

```
#include <stdio.h>
#include <math.h>
int main(void) {
  const int width = 80;
  const int height = 20;
  const double freq = 0.3;
  for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
    double value = sin(x * freq) +
                   cos(y * freq);
    if (value > 1.0) putchar('*');
    else if (value > 0.0) putchar('+');
    else if (value > -1.0) putchar('.');
    else putchar(' ');
   }
   putchar('\n');
  }
  return 0;
}
```

Зоряне небо з випадковими елементами

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
int main(void) {
  const int width = 80;
  const int height = 25;
  srand((unsigned)time(NULL));
  for (int y = 0; y < height; y++) {
    for (int x = 0; x < width; x++) {
      int r = rand() % 100;
      if (r < 2)
        putchar('*');
      else if (r < 5)
        putchar('+');
```

```
      else if (r < 8)
        putchar('.');
      else
        putchar(' ');
    }
    putchar('\n');
  }
  return 0;
}
```

Фрактал — трикутник Серпінського

```
#include <stdio.h>
#define SIZE 64
int canvas[SIZE][SIZE];
void sierpinski(int x, int y, int size) {
  if (size == 1) {
    canvas[y][x] = 1;
    return;
  }
  int half = size / 2;
  sierpinski(x, y, half);
  sierpinski(x - half, y + half, half);
  sierpinski(x + half, y + half, half);
}
int main(void) {
  for (int y = 0; y < SIZE; y++)
    for (int x = 0; x < SIZE; x++)
      canvas[y][x] = 0;
  sierpinski(SIZE / 2, 0, SIZE / 2);
  for (int y = 0; y < SIZE; y++) {
    for (int x = 0; x < SIZE; x++)
      putchar(canvas[y][x] ? '*' : ' ');
    putchar('\n');
  }
  return 0;
}
```

Навчальне видання

Іван Михайлович **Данилюк**
Галина Михайлівна **Унгурян**

Програмування. Мова С

Навчальний посібник

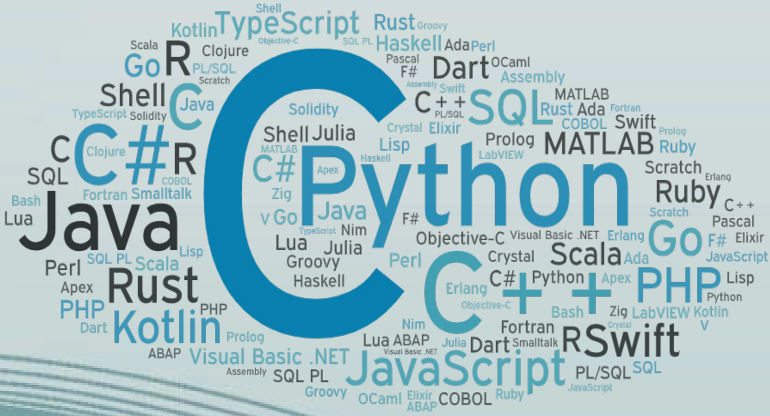
Відповідальний за випуск ***Бігун Я.Й.***

Літературний редактор ***Колодій О.В.***

Технічне редагування
та дизайн обкладинки ***Чораєва Г.К.***

Електронне видання
Підписано до друку 31.10.2025
Умов.-друк. арк. 14,7. Обл.-вид. арк. 15,8.
Зам. Н-079.
Видавництво Чернівецького національного університету.
58002, Чернівці, вул. Коцюбинського, 2.
e-mail: ruta@chnu.edu.ua

Свідоцтво суб'єкта видавничої справи ДК № 891 від 08.04.2002.



ЧНУ РУТА
ISBN 978-617-8703-21-9