

МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ

ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ  
УНІВЕРСИТЕТ РАДІОЕЛЕКТРОНІКИ

Кафедра «Програмна інженерія»

**ЗВІТ**

до практичного заняття №1 з дисципліни

«Аналіз та рефакторинг коду»

На тему: «Правила оформлення програмного коду»

Виконав:

ПЗП-22-10

Петриков Олександр Дмитрович

Перевірив: ст. гр.

ст. викладач кафедри ПІ

Сокорчук Ігор Петрович

Харків 2024

**Мета:**

Навчитися рефакторингу програмного коду, закріпiti основнi правила оформлення коду.

**Завдання:**

Обрати мову програмування для прикладiв коду. Створити презентацiю на тему «Правила оформлення програмного коду».

**Хiд роботи:**

Було обрано мову програмування C.

**Вступ**

Мова програмування C є однiєю з найпопулярнiших мов для системного програмування, розробки вбудованих систем та створення високопродуктивних додаткiв. Вона вiдома своєю простотою та потужнiстю, але через низькорiвневий характер мови написання чистого, ефективного та пiдтримуваного коду вимагає дотримання певних правил. У цiй роботi розглянемо основнi рекомендацiї для написання якiсного коду на C.

Вiдео-презентацiя: <https://youtu.be/yrTrxtf0ZUc>

### 1. Стильове оформлення коду

Використовуйте послiдовний стиль оформлення коду (вidstупи, розташування дужок, пробiли).

Послiдовний стиль покращує читабельнiсть коду та полегшує його пiдтримку.

```
// Поганий приклад
int sum(int a,int b){
    return a+b;}
```

```
// Гарний приклад
int sum(int a, int b) {
    return a + b;
}
```

Пояснення:

Використання пробілів між аргументами та операторами робить код більш зрозумілим.

Відступи та розташування дужок допомагають візуально відокремити блоки коду.

## 2. Правила найменування змінних та функцій

Використовуйте зрозумілі та описові імена для змінних, функцій та констант.

Імена повинні відображати призначення змінної або функції.

// Поганий приклад

```
int x, y;  
int f(int a, int b) {  
    return a + b;  
}
```

// Гарний приклад

```
int width, height;  
int calculate_area(int width, int height) {  
    return width * height;  
}
```

Пояснення:

- Використання зрозумілих імен (width, height, calculate\_area) полегшує розуміння коду.
- Уникайте однолітерних імен, окрім тимчасових змінних у коротких циклах.

## 3. Дотримуйтесь єдиного стилю форматування коду

Структурування коду уніфікованим способом підвищує його читабельність і спрощує командну роботу.

Поганий приклад:

```
int main() {  
int x=0;  
if(x==0){printf("Zero");}  
}
```

Гарний приклад:

```
int main() {
    int x = 0;
    if (x == 0) {
        printf("Zero");
    }
    return 0;
}
```

Пояснення:

Використання відступів і правильного форматування робить код структурованим і зрозумілим.

#### 4. Структура коду

Розділяйте код на модулі та функції, дотримуючись принципу єдиної відповідальності.

Кожна функція повинна виконувати лише одну задачу.

Приклад:

```
// Поганий приклад

void process_data(int* data, int size) {

    // Обробка даних

    for (int i = 0; i < size; i++) {

        data[i] *= 2;

    }

    // Вивід даних

    for (int i = 0; i < size; i++) {

        printf("%d ", data[i]);

    }

}
```

```
}
```

```
// Гарний приклад
```

```
void multiply_by_two(int* data, int size) {  
    for (int i = 0; i < size; i++) {  
        data[i] *= 2;  
    }  
}
```

```
void print_data(int* data, int size) {  
    for (int i = 0; i < size; i++) {  
        printf("%d ", data[i]);  
    }  
}
```

Пояснення:

- Розділення коду на функції покращує модульність та полегшує тестування.
- Кожна функція має чітке призначення, що робить код більш зрозумілим.

## 5. Уникайте магічних чисел

Опис:

Магічні числа (константи без пояснення) ускладнюють розуміння коду.

Поганий приклад:

```
if (speed > 60) {  
    printf("Over speed limit!");
```

```
}
```

Гарний приклад:

```
#define SPEED_LIMIT 60
```

```
if (speed > SPEED_LIMIT) {  
    printf("Over speed limit!");  
}
```

Пояснення:

Константи з осмисленими іменами полегшують зміни в майбутньому.

## 6. Оптимізація продуктивності

Уникайте передчасних оптимізацій, але враховуйте ефективність при роботі з ресурсами.

На першому етапі пишіть зрозумілий код, а потім оптимізуйте його за необхідності.

```
// Поганий приклад (передчасна оптимізація)  
for (int i = 0; i < strlen(str); i++) {  
    // ...  
}
```

```
// Гарний приклад  
int length = strlen(str);  
for (int i = 0; i < length; i++) {  
    // ...  
}
```

Пояснення:

- Виклик `strlen` у циклі призводить до непотрібних обчислень.
- Оптимізація полягає у винесенні виклику функції за межі циклу.

## Висновки

Дотримання рекомендацій щодо написання коду на С дозволяє створювати чистий, ефективний та підтримуваний код. Це включає:

Використання зрозумілих імен.

Розділення коду на модулі та функції.

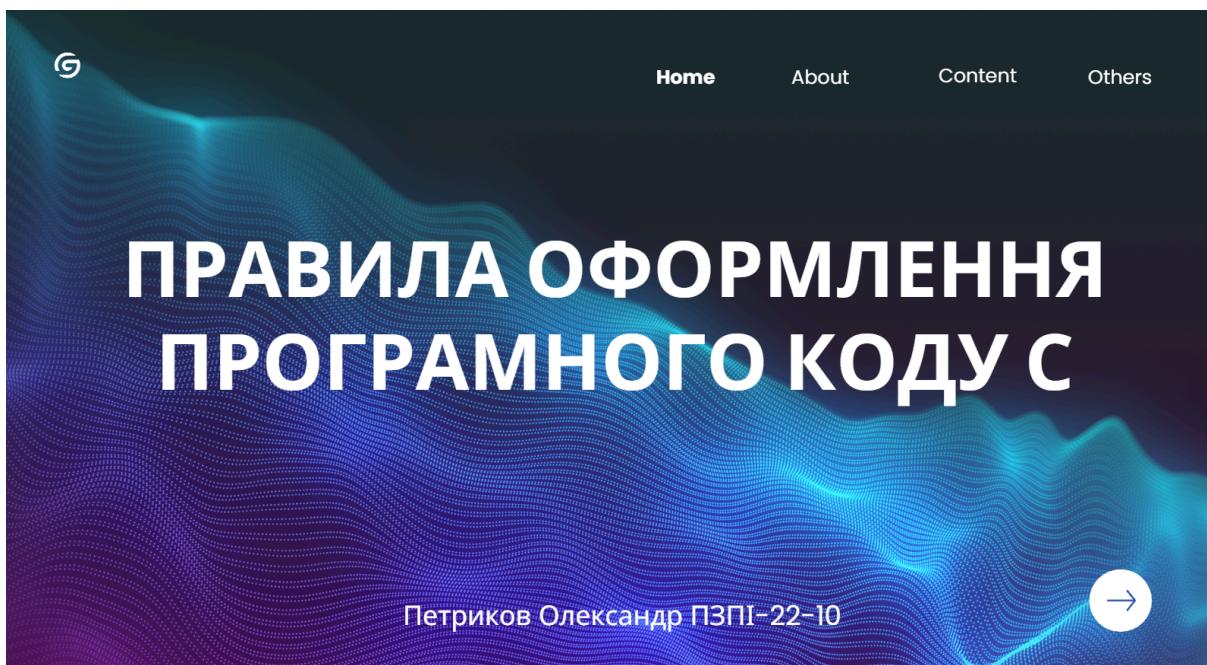
Обробку помилок та документацію.

Оптимізацію продуктивності без шкоди читабельності.

Ці принципи допомагають уникнути багатьох поширеніх помилок і полегшують роботу з кодом у майбутньому.

Відео-презентація: <https://youtu.be/yrTrxtf0ZUc>

Слайди:



Слайд 1

[Home](#)[About](#)[Content](#)[Others](#)

Мова програмування С є однією з найпопулярніших мов для системного програмування, розробки вбудованих систем та створення високопродуктивних додатків. Вона відома своєю простотою та потужністю, але через низькорівневий характер мови написання чистого, ефективного та підтримуваного коду вимагає дотримання певних правил. У цій роботі розглянемо основні рекомендації для написання якісного коду на С.

## Слайд 2

[Home](#)[About](#)[Content](#)[Others](#)

### 1. Стильове оформлення коду

Використовуйте послідовний стиль оформлення коду (відступи, розташування дужок, пробіли).  
Послідовний стиль покращує читабельність коду та полегшує його підтримку.

```
// Поганий приклад  
int sum(int a,int b){  
    return a+b;}
```

```
// Гарний приклад  
int sum(int a, int b) {  
    return a + b;  
}
```

#### Пояснення:

Використання пробілів між аргументами та операторами робить код більш зрозумілим.  
Відступи та розташування дужок допомагають візуально відокремити блоки коду.

## Слайд 3

[Home](#)[About](#)**Content**[Others](#)

## 2. Правила найменування змінних та функцій

Використовуйте зрозумілі та описові імена для змінних, функцій та констант.

Імена повинні відображати призначення змінної або функції.

```
// Поганий приклад  
int x, y;  
int f(int a, int b) {  
    return a + b;  
}
```

```
// Гарний приклад  
int width, height;  
int calculate_area(int width, int height) {  
    return width * height;  
}
```

### Пояснення:

- Використання зрозумілих імен (width, height, calculate\_area) полегшує розуміння коду.
- Уникайте однолітерних імен, окрім тимчасових змінних у коротких циклах.

## Слайд 4

[Home](#)[About](#)**Content**[Others](#)

## 3. Дотримуйтесь єдиного стилю форматування коду

Структурування коду уніфікованим способом підвищує його читабельність і спрощує командну роботу.

### Поганий приклад:

```
int main() {  
    int x=0;  
    if(x==0){printf("Zero");}  
}
```

### Гарний приклад:

```
int main() {  
    int x = 0;  
    if (x == 0) {  
        printf("Zero");  
    }  
    return 0;  
}
```

### Пояснення:

Використання відступів і правильного форматування робить код структурованим і зрозумілим.

## Слайд 5



#### 4. Структура коду

Розділяйте код на модулі та функції, дотримуючись принципу єдиної відповідальності.  
Кожна функція повинна виконувати лише одну задачу.

Приклад:

```
// Поганий приклад
void process_data(int* data, int size) {
    // Обробка даних
    for (int i = 0; i < size; i++) {
        data[i] *= 2;
    }
    // Вивід даних
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
}
```

```
// Гарний приклад
void multiply_by_two(int* data, int size) {
    for (int i = 0; i < size; i++) {
        data[i] *= 2;
    }
}
void print_data(int* data, int size) {
    for (int i = 0; i < size; i++) {
        printf("%d ", data[i]);
    }
}
```

Пояснення:

- Розділення коду на функції покращує модульність та полегшує тестування.
- Кожна функція має чітке призначення, що робить код більш зрозумілим.

#### Слайд 6



#### 5. Уникайте магічних чисел

Магічні числа (константи без пояснення) ускладнюють розуміння коду

Поганий приклад:

```
if (speed > 60) {
    printf("Over speed limit!");
}
```

Гарний приклад:

```
#define SPEED_LIMIT 60
if (speed > SPEED_LIMIT) {
    printf("Over speed limit!");
}
```

Пояснення:

Константи із зрозумілими іменами полегшують зміни в майбутньому.

#### Слайд 7



## 6. Оптимізація продуктивності

Уникайте передчасних оптимізацій, але враховуйте ефективність при роботі з ресурсами.  
На першому етапі пишіть зрозумілий код, а потім оптимізуйте його за необхідності.

```
// Поганий приклад (передчасна оптимізація)
for (int i = 0; i < strlen(str); i++) {
    // ...
}
```

```
// Гарний приклад
int length = strlen(str);
for (int i = 0; i < length; i++) {
    // ...
}
```

### Пояснення:

Виклик `strlen` у циклі призводить до непотрібних обчислень.  
Оптимізація полягає у винесенні виклику функції за межі циклу.

## Слайд 8



## Висновки

Дотримання рекомендацій щодо написання коду на С дозволяє створювати чистий, ефективний та підтримуваний код. Це включає:

Використання зрозумілих імен.

Розділення коду на модулі та функції.

Обробку помилок та документацію.

Оптимізацію продуктивності без шкоди читабельності.

Ці принципи допомагають уникнути багатьох поширеніх помилок і полегшують роботу з кодом у майбутньому.

## Слайд 9