



ПІДХОДИ ДО НАЛАГОДЖЕННЯ ПРОГРАМНОГО КОДУ

Питання 2.4.

Налагодження коду

- **Налагодження** – це процес виправлення (визначення, аналізу та видалення) помилок (bug) у програмному забезпеченні.
 - Процес починається після виявлення неможливості подальшого коректного виконання програми та призводить до вирішення проблеми й успішного тестування ПЗ.
- **Кроки процесу налагодження:**
 - Визначення проблеми та підготовка звіту.
 - Призначення звіту для інженера ПЗ відносно дефекту для відтворення цього дефекту.
 - Аналіз дефекту за допомогою моделювання, документації, виявлення та тестування недоліків тощо.
 - Виправлення дефекту шляхом внесення змін у систему.
 - Перевірка (Validation) внесених змін.

Налагодження коду

■ Стратегії налагодження:

- Ширше дослідити систему для кращого її розуміння. Дозволяє налагоджувати систему в різних станах залежно від потреби, знаходити внесені зміни в код.
- Зворотний (Backward) аналіз проблеми від місця появи повідомлення про помилку, щоб визначити збійну частину коду. Детальний розгляд цієї області дозволяє знайти причини дефектів.
- Форвардний (Forward) аналіз програми включає трасування (tracing) програми за допомогою точок розриву або операторів мови програмування для друку. Область коду з неправильним виводом інформації вказує на потребу звернути на неї увагу.

■ Існує кілька видів помилок, які можуть виникати на різних етапах процесу програмування

- Помилки компілятора (compiler errors)
- Помилки компоувальника (linker errors)
- Помилки часу виконання (run-time errors)

Помилки компіляції (Compiler Errors)

- Помилки, про які повідомляє компілятор.
 - Зазвичай це прості синтаксичні помилки: опечатки, пропущена пунктуація тощо.

```
int main()
{
    cout << "Hello World!"
    return 0;
}
```



Line	File	Message
3	c:\hello.cpp	In function 'int main()': 'cout' undeclared (first use this function)
4	c:\hello.cpp	(Each undeclared identifier is reported only once for each function it appears in) expected ';' before "return"

- Спочатку маємо попередження (warning) в рядку 3.
 - Компілятор не може знайти визначення cout.
 - cout визначений у стандартній бібліотеці iostream, яку слід підключити в програму. Також потрібно використовувати стандартний простір імен.
- Також маємо помилку (error) у рядку 4.
 - Забули крапку з комою в попередньому рядку.
- Виправлена програма:

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

int main()
{
    cout << "Hello World!";
    return 0;
}
```

Помилки компоувальника (Linker Errors)

- Часто спричинені відсутністю оголошення або реалізації (implement) функції.

```
void myFunction(); // prototype  
  
int main()  
{  
    myFunction ();  
    return 0;  
}
```



Line	File	Message
	c:\temp\ccURgaaa.o(text+)x2b)	In function 'int main()': [Linker error] undefined reference to 'myFunction()'
	c:\temp\ccURgaaa.o(text+)x2b)	ld returned 1 exit status

- Так компоувальник каже: «Я не можу знайти визначення функції myFunction».
 - Хоч записано прототип функції myFunction, вона ніколи не була визначена.

Помилки часу виконання (Run-time Errors)

- Логічні помилки, тобто помилки в проектуванні коду.
 - Найскладніші для знаходження та виявляються під час роботи програми.
 - Єдиний прояв – неправильний вивід або збій програми.
 - Програма може працювати роками, а потім раптово зазбоїти при певному вводі даних через логічну помилку.
- Налаштовувальник допомагає знайти логічні помилки.
 - Він дозволяє виконати програму покроково або поблочно, зупинити виконання в заданий момент та перевірити значення змінних у процесі роботи програми.
 - Так можна краще **зрозуміти**, як працює програма, та знайти точне місце проблеми.
 - Налаштовувальник не може сказати, чому виникла проблема – це робота програміста.

Метод налагоджування 1

- Простий, проте часто корисний: вставка оператора виводу (тут – `cout`) у різні місця програми. Це дозволяє
 - Визначити, чи конкретна частина програми виконується взагалі.
 - Вивести значення різних змінних у конкретні моменти виконання програми.

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

int main()
{
    int num, product = 1;

    do {
        cout << "Enter a number (0 to exit): ";
        cin >> num;
        product *= num;
    } while (num != 0);

    cout << "Product = " << product << endl << endl;

    system ("PAUSE");
    return 0;
}
```

Виві при запуску:

Product = 0

Press any key to continue ...

Можемо вставити інструкцію `cout` відразу після `cin >> num`, щоб вивести кожне число, яке йде в добуток (перевірити, чи кількість зчитувань правильна).

Метод налагоджування 1

- Також можна вставити оператор `cout` після кожного обчислення добутку.

```
do {  
    cout << "Enter a number (0 to exit): ";  
    cin >> num;  
    cout << "    Multiply by " << num << endl;  
    product *= num;  
    cout << "    Running product " << product << endl;  
} while (num != 0);
```

- Після запуску бачимо, що останнє помножене число є 0, яке мало позначати точку зупинки:

```
Enter a number (0 to exit): 2  
    Multiply by 2  
    Running product 2  
Enter a number (0 to exit): 3  
    Multiply by 3  
    Running product 6  
Enter a number (0 to exit): 4  
    Multiply by 4  
    Running product 24  
Enter a number (0 to exit): 0  
    Multiply by 0  
    Running product 0  
Product = 0
```

Вирішення проблеми:

```
do {  
    cout << "Enter a number (0 to exit): ";  
    cin >> num;  
    if (num != 0)  
        product *= num;  
} while (num != 0);
```

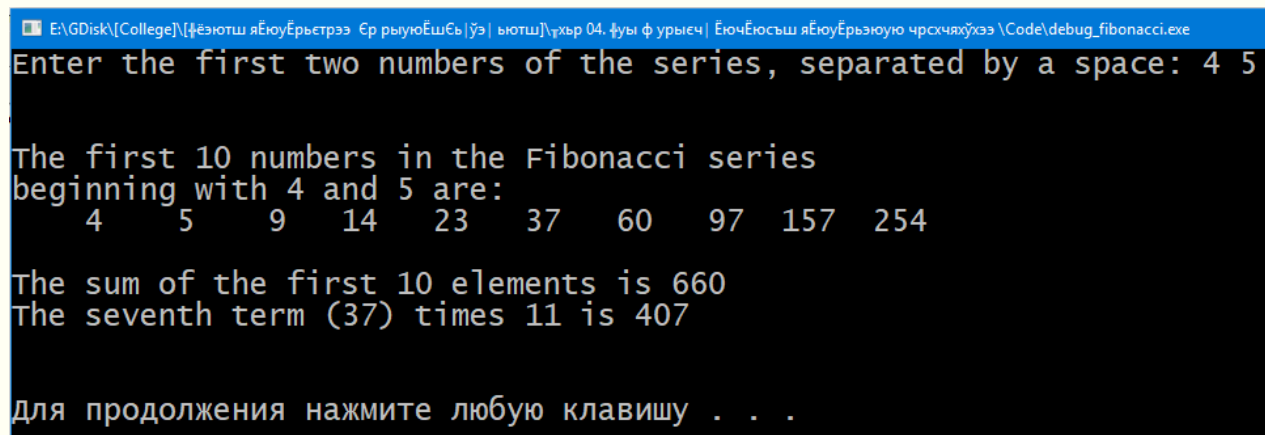

Метод налагоджування 2

```
1  #include <iostream>
2  #include <iomanip>
3
4  using namespace std;
5
6  int main()
7  {
8      int num1, num2,      // First & second numbers in series
9          seventh,        // Seventh term
10         newTerm,        // Next term in the series
11         sum;            // Running sum of elements
12     cout << "Enter the first two numbers of the series, separated by a space: ";
13     cin >> num1 >> num2;
14     cout << endl << endl << "The first 10 numbers in the Fibonacci series" << endl;
15     cout << "beginning with " << num1 << " and " << num2 << " are:" << endl;
16     cout << setw(5) << num1 << setw(5) << num2;
17     sum = num1 + num2;      // initial sum (first 2 terms)
18     for (int count = 3; count <= 10; ++count) // generate/display 3rd -10th terms
19     {
20         if (count == 7)
21             seventh = newTerm;      // save seventh term
22         newTerm = num1 + num2;
23         sum += newTerm;
24         cout << setw(5) << newTerm;
25         num1 = num2; num2 = newTerm;
26     }
27     cout << endl << endl;
28     cout << "The sum of the first 10 elements is " << sum << endl;
29     cout << "The seventh term (" << seventh << ") times 11 is " << seventh * 11 << endl;
30     cout << endl << endl;
31     system ("PAUSE");
32     return 0;
33 }
```

- Використати налагоджувальник, вбудований у Dev C++.

Метод налагоджування 2

- Результат має бути іншим, оскільки сума перших 10 елементів повинна дорівнювати 7му елементу послідовності, помноженому на 11.



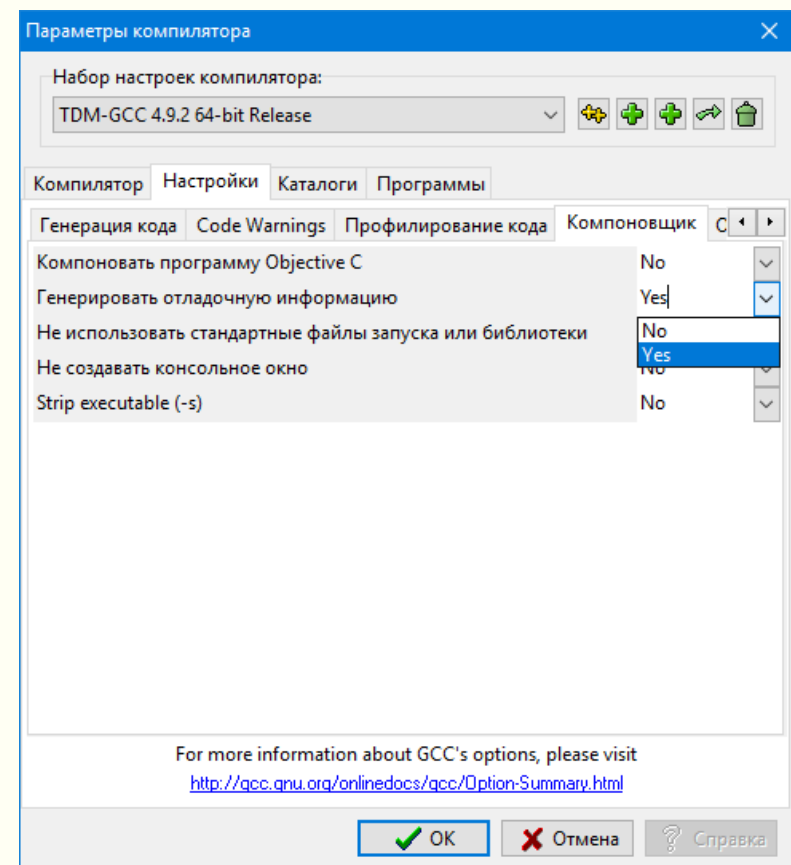
```
E:\GDisk\College\{фэзютш яёюёрьстрээ ёр рыуюёшсь|ўз| ьютш|тхър 04. фуы ф урыеч| ёючёюсьш яёюёрьзуюю чрсхчяхўзэ \Code\debug_fibonacci.exe
Enter the first two numbers of the series, separated by a space: 4 5

The first 10 numbers in the Fibonacci series
beginning with 4 and 5 are:
 4   5   9  14  23  37  60  97 157 254

The sum of the first 10 elements is 660
The seventh term (37) times 11 is 407

для продолжения нажмите любую клавишу . . .
```

- Налаштуємо налагоджувальник (debugger):
 - Меню Сервис → Параметры компилятора



Метод налагоджування 2. Встановлення точок переривання (breakpoints)

- Точка переривання – місце в програмі, де розробник бажає тимчасово зупинити її виконання, щоб переглянути значення змінних.
- Поставимо точку переривання всередині оператора if.

- Натисніть на номер рядка для встановлення.

```
20 | if (count == 7)
21 | seventh = newTerm; // save seventh term
22 | newTerm = num1 + num2;
```

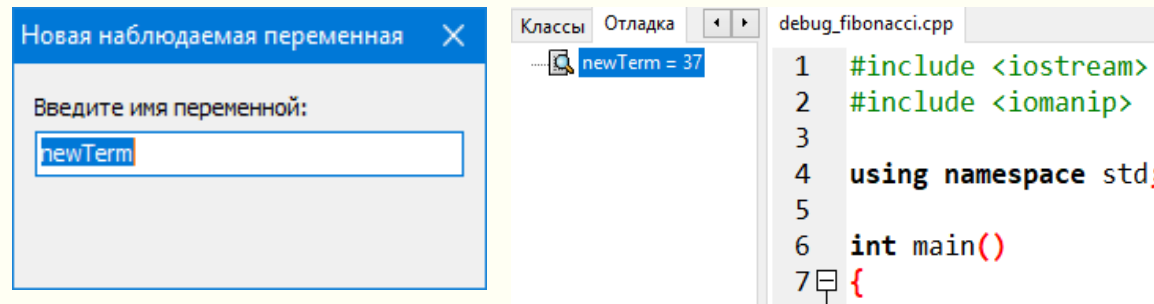
- Для виконання програми до точки переривання натисніть кнопку Debug button (✓) АБО клавішу F5.
- У програмі вводяться 2 числа, щоб дійти до точки переривання.
- Наступний рядок після точки переривання буде підсвічений синім кольором.

```
20 | if (count == 7)
21 | seventh = newTerm; // save seventh term
22 | newTerm = num1 + num2;
```

Метод налагоджування 2. Додавання змінних для стеження за їх значеннями

- Після досягнення точки переривання бажаємо перевірити вміст змінної newTerm.

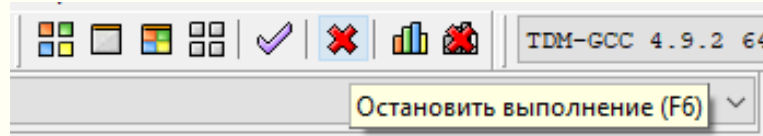
```
13 cin >> num1 >> num2;
14 cout << endl << endl;
15 cout << "beginning with " << num1 << " and " << num2 << " are\n";
16 cout << setw(5) << num1 << " + " << num2 << " = ";
17 sum = num1 + num2;
18 for (int count = 3; count <= 7; count++)
19 {
20     if (count == 7)
21         seventh = newTerm;
22     newTerm = num1 + num2;
23     sum += newTerm;
24     cout << setw(5) << newTerm << " ";
25     num1 = num2; num2 = newTerm;
26 }
```



Значення newTerm = 37.
Це значення 6-го елемента, а не 7-го!

Метод налагоджування 2. Додавання змінних для стеження за їх значеннями

- Спочатку зупинемо процес налагодження та перезапустимо його.



- Далі видалимо точку переривання, оскільки вона вже відіграла свою роль.
 - Зніміть червону точку переривання, натиснувши на неї.

A screenshot of the Visual Studio IDE showing the 'bad_fib.cpp' file. The 'Debug' window on the left shows the current state of variables: num1 = 37, num2 = 60, newTerm = 97, sum = 249, and count = 8. The main code window shows the following code:

```
11  
12     cout << "Enter the first two numbers of the series, separated by a space: ";  
13     cin >> num1 >> num2;  
14     cout << endl << endl << "The first 10 numbers in the Fibonacci series" << endl;  
15     cout << "beginning with " << num1 << " and " << num2 << " are:" << endl;  
16     cout << setw(5) << num1 << setw(5) << num2;  
17     sum = num1 + num2; // initial sum (first 2 terms)  
18     for (int count = 3; count <= 10; ++count) // generate/display 3rd -10th terms  
19     {  
20         if (count == 7)  
21             seventh = newTerm; // save seventh term  
22         newTerm = num1 + num2;  
23         sum += newTerm;  
24         cout << setw(5) << newTerm;  
25         num1 = num2;  
26         num2 = newTerm;
```

The line 17 is highlighted in red, and line 24 is highlighted in blue. A red breakpoint icon is visible on line 17, and a blue breakpoint icon is visible on line 24.

Трасування програми та налагоджувальна інформація

- Запустимо налагоджувальник (кнопка F8).
 - За успішного виконання програма зупиниться на першій точці переривання.
 - Далі можна покроково пройти код (комбінація Shift-F7) без заходу в функції або із заходом у них (клавіша F7 або кнопка "next step").
 - Натиснувши Ctrl-F7 або кнопку "continue", можна продовжити виконання до наступної точки.
 - Точки переривання можна додавати та видаляти в будь-який момент.

The screenshot displays a debugger interface with a toolbar at the top containing buttons for Compiler, Resources, Compile Log, Debug (checked), Find Results, and Close. Below the toolbar are buttons for Debug, Add watch, Next line, Continue, Next instruction, Stop Execution, View CPU window, Into function, Skip function, and Into instruction. An Evaluate: input field is located below these buttons.

The CPU Window is open, showing the disassembly of the `main()` function. The instruction list includes:

- 57: `0x00000000040166d <+248>: mov rcx, rax`
- 58: `0x000000000401670 <+251>: call 0x44d6b0 <_ZNSolsEi>`
- 59: `=> 0x000000000401675 <+256>: mov edi, DWORD PTR [rsp+0x2c]`
- 60: `0x000000000401679 <+260>: add edi, DWORD PTR [rsp+0x28]` (highlighted)
- 61: `0x00000000040167d <+264>: mov esi, 0x3`
- 62: `0x000000000401682 <+269>: jmp 0x4016c4 <main()+335>`
- 63: `0x000000000401684 <+271>: cmp esi, 0x7`
- 64: `0x000000000401687 <+274>: jne 0x40168b <main()+278>`
- 65: `0x000000000401689 <+276>: mov ebp, ebx`
- 66: `0x00000000040168b <+278>: mov ebx, DWORD PTR [rsp+0x2c]`
- 67: `0x00000000040168f <+282>: add ebx, DWORD PTR [rsp+0x28]`
- 68: `0x000000000401693 <+286>: add edi, ebx`
- 69: `0x000000000401695 <+288>: mov rcx, DWORD PTR [rin+0x8b0f41]`

Below the instruction list is a Backtrace table:

Function	File	Line
main()	bad_fib.cpp	17

On the right side, the Register window shows the current state of registers:

Register	Hex
RAX	0x486a00
RBX	0x1
RCX	0xffffffff
RDX	0x490d98
RSI	0x5
RDI	0x486a08
RBP	0x8b14b0
RSP	0x70fdd0
R8	0x5
R9	0x70e060
R10	0x0
R11	0x246
R12	0x1
R13	0x8
R14	0x0
R15	0x0
RIP	0x401675
EFLAGS	0x202
CS	0x33
SS	0x2b
DS	0x0
ES	0x0
FS	0x0
GS	0x0

Відмінності між налагодженням та тестуванням

- Налаштування відрізняється від тестування.
 - Тестування концентрується на виявленні багів, помилок тощо, а налаштування починається вже після виявлення багу в ПЗ.
 - Тестування використовується для забезпечення коректної роботи програми та передбачається його виконання з певним мінімальним рівнем успішності.
 - Тестування може бути ручним (manual) або автоматизованим (automated).
 - Існує кілька видів тестування: модульне (unit testing), інтеграційне (integration testing), альфа-, бета-тестування та ін.
- Налаштування вимагає значних знань, навичок та досвіду.
 - Воно не підтримується автоматизованими інструментами, а є більш ручним процесом, оскільки кожний баг відрізняється та вимагає різних підходів до свого подолання.



ДЯКУЮ ЗА УВАГУ!

Наступна тема: Програми та їх представлення в пам'яті комп'ютера