**Лабораторна робота 06. Написання програм з вкладеними циклами та розгалуженнями, використанням математичних функцій та виконанням операцій введення- виведення**

**На період** карантину в дистанційній формі навчання потрібно використовувати замість **С++ Builder онлайн компілятор C++ Shell, який доступний за адресом** [**http://cpp.sh**](http://cpp.sh) (інструкція надана після тексту ЛР№3). При наявності на домашньому комп’ютері іншого завдання можна виконати на ньому. Результати надсилати на електронну адресу викладача [**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді URL-посилання для робіт, виконаних у **C++ Shell,** або для робіт, на компіляторі С++ домашнього комп’ютера cpp-файл з іменем у форматі

**АР<Номер групи><Номер практичної / лабораторної [літера позначення типу роботи P – практична, R – лабораторна]<Прізвище англійською>**. Наприклад, **АРPTBD-2104P**buts.срр.

При відсутності можливості використовувати компілятор або відсутності можливості доступу до Інтернету текст програми набрати в Блокноті або WordPad Windows, в крайньому випадку текст програми можна записати у зошити, сфотографувати, оформити відповідним чином (включити титульний лист, записати завдання, а потім включити файли з фотографіями) та надіслати на електронну адресу викладача.

Оформлені роботи повинні надсилатися у вигляді текстового документу з іменем файлу:

**АР<Номер групи><Номер лекції / практичної / лабораторної [літера позначення типу роботи L – лекція, P – практична, R – лабораторна]<Прізвище англійською>**. Наприклад, **АРPTBD-2104P**buts.doc.

**Тему в заголовку листа записати**

**АР <Номер групи>-><Номер лекції / практичної / лабораторної [літера позначення типу роботи L – лекція, P – практична, R – лабораторна]<Прізвище англійською>**

**Строк відсилки ЛР 24.03.2021.**

Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, тему в заголовку листа записати

**АР<Номер групи>-Запитання-<Прізвище англійською>**.

**Мета**: навчитися писати програми з вкладеними циклами та розгалуженнями, використанням математичних функцій та виконанням операцій введення- виведення **на мові С++** в **консольному режимі.**

**Методичні вказівки щодо організації самостійної роботи студентів**

1. Запустити середовище програмування С++ .
2. Записати програму, що виконує 3 завдання з пп.3.1-3.3. В першому рядку кожної програми записати

*// ПТБД-21 Група № Прізвище*

вказавши номер своєї групи та своє прізвище.

Як і в ЛР №5, програма повинна запитати номер завдання (число 1, 2, 3 або 0 для закінчення) і в залежності від введеного значення виконувати відповідне завдання. Якщо введений 0 – програма припиняє роботу.

Вхідні дані ввести, а результати вивести, використовуючи потокове введення-виведення даних.

Значення ***N*** вводиться з консолі і не повинне мати обмеження.

Виводяться змінні Хта ***N*** перед початком розрахунку*,* а змінну циклу підсумкове значення ***y*** на кожній ітерації/кроку циклу***.*** Розрахунок виконується з точністюε=10-3 (до третьогодесяткового знаку).

1. Завдання:
   1. Написати код, що виконує розрахунок ***y*** підсумку ***N*** перших членів з точністю ε=10-3:



Підсумок позначити як Y, значення ***N*** ввести перед початком роботи.

Проаналізуйте вираз підсумку та складіть компактну формулу розрахунку в циклі.

* 1. Написати код, що виконує розрахунок ***y*** підсумку ***N*** перших членів з точністю ε=10-3



* 1. Написати код, що виконує розрахунок:

, якщо    

**Теоретичні відомості.**

Розглянемо декілька прикладів на розрахунок підсумку членів деякого ряду значень.

***Приклад 1.***



Тут чисельник зменшується на 2, а знаменник збільшується на 2, тобто формула буде мати вигляд.

**((R – (2+J))/(R + J)**

Цей запис означає, що J змінюється від 0 до N з кроком 2, а сам вираз для розрахунку надається в дужках.

***Приклад 2.***



Тут до чисельника додається число **j,** яке змінюється від 1 до ***N*** з кроком 1***,*** а до знаменника – число 3**j**. Тобто формула бути мати вигляд:

**((x+j)/ (x+3j))**

**ДЛЯ ДОВІДКИ**

**Математичні функції (**заголовний файл **math.h)**

| **Прототип функції** | **Ім’я** | **Призначення** |
| --- | --- | --- |
| double sin (double \_х); | **sin (x)** | синус x (в радіанах) — **sin x** |
| double cos (double \_x); | **cos (x)** | косинус x (в радіанах) — **cos х** |
| double tan (double \_x); | **tan (x)** | тангенс х (в радіанах) — **tg х** |
| double asin (double \_x); | **asin (x)** | арксинус х — **arcsin х** |
| double acos (double \_x); | **acos (x)** | арккосинус х — **arcos х** |
| double atan (double \_x); | **atan (x)** | арктангенс х — **arctg х** |
| double atan2 (double \_y, double\_x); | **atan2 (y,x)** | арктангенс у/х — **arctg (у/х)** |
| double sinh (double \_x); | **sinh (x)** | синус гіперболічний х — **sh х** |
| double cosh (double \_x); | **cosh (x)** | косинус гіперболічний х — **ch х** |
| double tanh (double \_x); | **tanh (x)** | тангенс гіперболічний х — **th х** |
| double log (double \_x); | **log (x)** | натуральний логарифм х — **ln х** |
| double log10 (double \_x); | **log10 (x)** | десятковий логарифм х — **log х** |
| double exp (double \_x); | **exp (x)** | піднесення е до степеня х — **ех** |
| double pow (double \_x, double\_y); | **pow (x,y)** | піднесення х до степеня у — **ху** |
| double pow 10 (int \_p) | **pow10 (p)** | повертає **10р** |
| double sqrt (double \_х); | **sqrt (x)** | корінь iз x, x > 0 |
| double hypot (double\_x, double\_y); | **hypot (x,y)** | корінь із (х2+у2) |
| double fabs (double \_\_x); | **fabs (x)** | абсолютне значення х — |х| типу **double** |
| int abs (int \_x); | **abs (x)** | абсолютне значення х — |х| типу **int** |
| long labs (long \_x); | **labs (x)** | абсолютне значення х — |х| типу **long** |
| double fmod (double \_\_x, double\_y); | **fmod (x,y)** | залишок від ділення х на у |
| double ceil (double \_\_x); | **ceil (x)** | округлення до більшого |
| double floor (double \_x); | **floor (x)** | повертає найближче ціле, не більше за х |
| double modf (double \_x, double); | **modf(x,&p)** | виділяє цілу й дробову частинні числа |
| double atof(const char\* \_s); | **atof (s)** | перетворює рядок символів у число з плаваючою крапкою |

**ЗАСТОРОЖЕННЯ ЩОДО ВИКОРИСТАННЯ**

**ЦИКЛІВ ТА ОПЕРАТОРУ GOTO**

**Нескінченні цикли**

Якщо умова циклу завжди приймає значення true, то і сам цикл буде виконуватися нескінченно. Це називається **нескінченним циклом**. Наприклад:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**int count = 0;**

**while (count < 10) // ця умова ніколи не буде false**

**// тому цей рядок буде виконуватися постійно**

**cout << count << " ";**

**return 0; // а цей рядок ніколи не виконається**

**}**

Оскільки змінна count не збільшується на одиницю в програмі, то умова count < 10 завжди буде true. Відповідно, цикл ніколи не завершиться і програма буде постійно виводити 0 0 0 0 0 ... .

Можна навмисно оголосити нескінченний цикл наступним чином:

**while (1) // чи while (true)**

**{**

**// Цей цикл буде виконуватися постійно**

**}**

Єдиний спосіб вийти з нескінченного циклу — використати один з наступних операторів: [**return**](https://acode.com.ua/urok-15-funktsiyi-i-operator-return/), break, exit, [**goto**](https://acode.com.ua/urok-69-operator-goto/) або згенерувати виняток. Програми, які працюють до тих пір, поки користувач не вирішить зупинити їх, іноді навмисно використовують нескінченні цикли разом з операторами return, break або exit для завершення циклу. Поширена така практика в серверних веб-додатках, які працюють безперервно і постійно обслуговують веб-запити.

## Оператор goto

**Оператор goto** — це **оператор управління потоком виконання програм**, який змушує центральний процесор виконати перехід з однієї ділянки коду в іншу (тобто здійснити стрибок). Інша ділянка коду ідентифікується за допомогою **мітки**. Наприклад:

**#include <iostream>**

**#include <cmath> // для функції sqrt()**

**using namespace std; *// простір імен***

**int main()**

**{**

**double z;**

**tryAgain: // це мітка**

**cout << "Enter a non-negative number: ";**

**cin >> z;**

**if (z < 0.0)**

**goto tryAgain; // а це оператор goto**

**cout << "The sqrt of " << z << " is " << sqrt(z) << endl;**

**return 0;**

**}**

У цій програмі користувачеві пропонується ввести додатне число. Однак, якщо користувач введе від’ємне число, то програма, використовуючи оператор goto, виконає перехід назад до мітки tryAgain. Потім користувачеві знову потрібно буде ввести число. Таким чином, ми можемо постійно просити користувача ввести число, поки він не зробить це коректно.

Раніше ми розглядали два типи області видимості: **локальна** (або ще “блокова”) і **глобальна** (або ще “файлова”). Лейбли використовують третій тип області видимості: **область видимості функції**. Оператор goto і відповідна мітка повинні знаходитися в одній і тій же функції.

Існують деякі обмеження на використання операторів goto. Наприклад, ви не зможете перестрибнути вперед через змінну, яка ініціалізована в тому ж блоці, що і goto:

**int main()**

**{**

**goto skip; // стрибок вперед заборонений**

**int z = 7;**

**skip: // мітка**

**z += 4; // яке значення буде в цій змінній?**

**return 0;**

**}**

Загалом програмісти уникають використання оператора goto в C++ (і в більшості інших високорівневих мов програмування). Основна проблема з ним полягає в тому, що він дозволяє програмісту керувати виконанням коду так, що точка виконання може довільно переміщуватися в коді. А це, в свою чергу, створює те, що досвідчені програмісти називають «спагетті-кодом». **Спагетті-код** — це код, порядок виконання якого нагадує тарілку зі спагеті (все заплутано і закручено), що вкрай ускладнює слідування та розуміння логіки виконання такого коду.

Як казав один відомий фахівець в інформатиці та в програмуванні, Едсгер Дейкстра: «Якість програмістів — це функція щільності використання операторів goto в програмах, які вони пишуть».

Оператор goto часто використовується в деяких старих мовах програмування, таких як Basic чи Fortran, або навіть в C. Однак в C++ goto майже ніколи не використовується, оскільки будь-який код, написаний з ним, можна більш ефективно переписати з використанням інших об’єктів в C++, таких як цикли, обробники винятків або деструктори (це все ми розглянемо у відповідних уроках).

**Правило: Уникайте використання операторів goto, якщо на це немає вагомої причини.**

## Лічильник циклу while

Часто потрібно щоб цикл виконувався певну кількість разів. Для цього зазвичай використовується змінна у вигляді лічильника циклу. **Лічильник циклу** — це цілочисельна змінна, яка оголошується з єдиною метою: рахувати, скільки разів виконався цикл.

Лічильникам циклу часто дають прості імена, такі як i, j чи k. Проте в цих іменах є одна серйозна проблема. Якщо ви захочете дізнатися, де у вашій програмі використовується лічильник циклу і скористаєтеся функцією пошуку i, j чи k, то в результаті отримаєте половину своєї програми, так як i, j та k використовуються в багатьох іменах. Отже, краще використовувати iii, jjj чи kkk в якості імен для лічильників. Вони більш унікальні, їх значно простіше знайти, і вони виділяються в коді. А ще краще використовувати «реальні» імена для змінних, наприклад, count або будь-яке інше ім’я, яке надає контекст використання цієї змінної.

Також для лічильників циклу краще використовувати тип **signed int**. Використання unsigned int може призвести до несподіваних результатів, наприклад:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**unsigned int count = 10;**

**// Рахуємо від 10 до 0**

**while (count >= 0)**

**{**

**if (count == 0) cout << "blastoff!";**

**else cout << count << " ";**

**--count;**

**}**

**return 0;**

**}**

В програмі вище є помилка, вона являє собою нескінченний цикл. Вона починається з виведення 10 9 8 7 6 5 4 3 2 1 blastoff! як і передбачалося, але потім все йде шкереберть і починається відлік з 4294967295. Тому що умова циклу count >= 0 ніколи не буде хибною! Коли count = 0, то і умова 0 >= 0 має значення true, виводиться blastoff, а потім виконується декремент змінної count, відбувається **переповнення** і значенням змінної стає 4294967295. **Переповнення** (англ. “**Overflow**“) трапляється при втраті біт через те, що змінній не було виділено достатньо пам’яті для їх зберігання. І так як умова 4294967295 >= 0 є істинною, то програма продовжує своє виконання. А оскільки лічильник циклу є типу unsigned, то він ніколи не зможе бути від’ємним і цикл ніколи не завершиться.

**Правило: Завжди використовуйте тип signed int для лічильників циклу.**

## Ітерації

Кожне виконання циклу називається **ітерацією** (або ще “**повтором**“).

Оскільки тіло циклу зазвичай є блоком, і оскільки цей блок виконується з кожним повтором по новому, то будь-які змінні, оголошені всередині тіла циклу, створюються, а в кінці тіла знищуються. У наступному прикладі змінна z створюється і знищується 6 разів:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**int count = 1;**

**int result = 0; // змінна result визначена тут, //оскільки вона нам знадобиться пізніше (поза тілом циклу)**

**while (count <= 6) // буде 6 ітерацій**

**{// змінна z створюється тут кожен раз з кожною ітерацією**

**int z;**

**cout << "Enter integer #" << count << ':';**

**cin >> z;**

**result += z;**

**// Збільшуємо значення лічильника циклу на одиницю**

**++count;**

**} // змінна z знищується тут з кожною ітерацією!**

**cout << "The sum of all numbers entered is: " << result;**

**return 0;**

**}**

Для фундаментальних типів змінних це нормально. Для НЕ фундаментальних типів змінних (таких як **структури** або класи) це може позначитися на продуктивності. Отже, не фундаментальні типи змінних краще визначати перед циклом.

Важливо, що змінна count оголошена поза тілом циклу, оскільки потрібно, щоб значення змінної зберігалося протягом усіх ітерацій (а не знищувалося з кожною ітерацією).

Коли може знадобитися виконати щось при досягненні певної кількості ітерацій, наприклад, вставити символ нового рядка, то це легко здійснити, використовуючи оператор залишку від ділення з лічильником циклу:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**int count = 1;**

**while (count <= 50)**

**{**

**// Виводимо числа до 10 (додаємо до кожного числа 0)**

**if (count < 10) cout << "0" << count << " ";**

**else**

**cout << count << " "; // виводимо інші числа**

**// Якщо лічильник циклу ділиться на 10 без**

**// залишку, то тоді вставляємо символ нового рядка**

**if (count % 10 == 0) cout << "\n";**

**// Збільшуємо значення лічильника циклу на одиницю**

**++count;**

**}**

**return 0;**

**}**

Результат виконання програми:

01 02 03 04 05 06 07 08 09 10  
11 12 13 14 15 16 17 18 19 20  
21 22 23 24 25 26 27 28 29 30  
31 32 33 34 35 36 37 38 39 40  
41 42 43 44 45 46 47 48 49 50

## Цикл for

**for (оголошення змінних; умова; інкремент/декремент лічильника)**

**тіло циклу;**

Або, конвертуючи for в еквівалентний **цикл while**:

**{ // зверніть увагу, що цикл знаходиться в блоці**

**// оголошення змінних;**

**while (умова)**

**{**

**тіло циклу;**

**інкремент/декремент лічильника;**

**}**

**} // змінні, оголошені всередині циклу, виходять з**

**// області видимості тут**

Змінні, визначені всередині циклу for, мають спеціальний **тип області видимості:** **область видимості циклу**. Такі змінні існують тільки всередині циклу і недоступні за його межами.

## Помилка неврахованої одиниці

Однією з найчастіших проблем з якою стикаються початківці в циклах for (а також і в інших типах циклів) є **помилка неврахованої одиниці**. Вона виникає, коли цикл повторюється на 1 раз більше або на 1 раз менше від потрібної кількості ітерацій. Це зазвичай відбувається через те, що в умові використовується некоректний оператор порівняння (наприклад, > замість >= або навпаки). Як правило, ці помилки важко відстежити, так як компілятор не буде скаржитися на них і програма працюватиме нормально, але її результати будуть неправильні.

При написанні циклів for пам’ятайте, що цикл буде виконуватися до тих пір, поки умова є істинною. Рекомендується тестувати цикли, використовуючи різні значення для перевірки працездатності циклу. Хорошою практикою є перевірка циклів за допомогою вхідних даних (чисел, символів та іншого), які змушують цикл виконатися 0, 1 і 2 рази. Якщо цикл працює справно, значить все ОК.

**Правило: Тестуйте свої цикли, використовуючи вхідні дані, які змушують цикл виконатися 0, 1 і 2 рази.**

## Пропущені вирази в циклі

Також в циклах можна пропускати один або відразу всі вирази. Наприклад:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**int count = 0;**

**for (; count < 10; )**

**{**

**cout << count << " ";**

**++count;**

**}**

**return 0;**

**}**

Результат:

0 1 2 3 4 5 6 7 8 9

Ініціалізацію лічильника ми прописали поза тілом циклу, а інкремент лічильника — всередині тіла циклу. У самому операторі for ми вказали лише умову. Іноді бувають випадки, коли не потрібно оголошувати лічильник циклу (тому що у нас вже є один) або збільшувати його (так як ми збільшуємо його якимось іншим способом).

Хоча це і не часто можна спостерігати, але в операторі for можна взагалі нічого не вказувати. Варто зазначити, що подібне призведе до нескінченного циклу:

**for (;;)  
    тіло циклу;**

Вищенаведений приклад еквівалентний наступному:

**while (true)  
    тіло циклу;**

## Оголошення змінних в циклі for

Хоча в циклах for зазвичай використовується тільки один лічильник, іноді можуть виникати ситуації, коли потрібно працювати відразу з декількома змінними. Для цього використовується **оператор Кома**. Наприклад:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**int aaa, bbb;**

**for (aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)**

**cout << aaa << " " << bbb << std::endl;**

**return 0;**

**}**

Цей цикл присвоює значення двом раніше оголошеним змінним: aaa = 0 і bbb = 9. Тільки з кожною ітерацією змінна aaa збільшується на одиницю, а bbb — зменшується на одиницю.

Результат виконання програми:

0 9  
1 8  
2 7  
3 6  
4 5  
5 4  
6 3  
7 2  
8 1  
9 0

Примітка: Вищенаведений цикл можна переписати наступним чином:

**#include <iostream>**

**using namespace std; *// простір імен***

**int main()**

**{**

**for (int aaa = 0, bbb = 9; aaa < 10; ++aaa, --bbb)**

**cout << aaa << " " << bbb << std::endl;**

**return 0;**

**}**

В такому випадку кома в оголошенні змінних є частиною синтаксису, а не використанням оператора Кома. Але ефект ідентичний.

**Оператор Кома** (або ще ***“оператор Comma”***) дозволяє обчислювати декілька виразів (в той час як допускається тільки один).

|  |  |  |  |
| --- | --- | --- | --- |
| **Оператор** | **Символ** | **Приклад** | **Операція** |
| Кома | , | x, y | Обчислюється x, потім обчислюється y, а потім повертається значення y |

Вираз, в якому знаходиться цей оператор, буде мати значення правого операнду, наприклад:

**int x = 0;**

**int y = 2;**

**int z = (++x, ++y); // інкремент x і y**

Змінній z присвоюється результат обчислення ++у (правого операнду), що дорівнює 3.

Майже в кожному випадку, оператор, в якому є оператор Comma, краще записувати у вигляді окремих інструкцій. Вищенаведений код коректніше буде записати наступним чином:

**int x = 0;**

**int y = 2;**

**++x;**

**++y;**

**int z = y;**

Оператор Comma має найнижчий **пріоритет** з усіх операторів (навіть нижче, ніж в оператора присвоювання). Через це, наступні два рядки коду виконують не одне і те ж:

|  |
| --- |
| z = (a, b); // спочатку обчислюється вираз (a, b), який дорівнює значенню b, а потім результат присвоюється змінній z  z = a, b; // обчислюється як "(z = a), b", тому змінній z присвоюється значення a, а змінна b ігнорується |

Більшість програмістів не використовують оператор Comma взагалі (хіба що тільки в циклах for).

Кома, яка використовується у викликах функцій, не є оператором Comma:

**int sum = add(x, y); // ця кома не є оператором Comma**

Аналогічно, при оголошенні декількох змінних в одному рядку, кома використовується як роздільник, а не як оператор:

**int x(3), y(5); // ця кома не є оператором Comma**

**Правило: Уникайте використання оператора Comma (винятком є використання в циклах for).**