**Лабораторна робота 06. Розробка програми з використанням блоків лінійної, розгалуженої та циклічної структури**

**Мета**: навчитися писати програми лінійної, розгалуженої та циклічної структури з вкладеними циклами та 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

*// Група № Прізвище Номер ЛР*

вказавши номер своєї групи, своє прізвище та номер ЛР.

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. Виводяться (змінні ***a*** або ***N*** перед початком розрахунку*, а* ***х, y*** та відповідні повідомлення на кожній ітерації/кроку циклу***).***

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



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

Проаналізуйте вираз підсумку та складіть компактну формулу розрахунку в циклі.

* 1. Написати код, що виконує розрахунок:

 якщо  

Запис  означає, х може приймати значення з діапазону [-1;2]. тобто х ≥ -1 та х ≤2. Запис  означає, що х змінюється з кроком 0.2.

* 1. . Написати код, що виконує розрахунок:

 якщо    

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

1. Запустити програму на виконання. Перевірити всі варіанти виконання завдання. При відсутності помилок текст програми надіслати викладачу.
2. При необхідності проведення консультації з питань виконання лабораторних робіт відправте відповідне повідомлення із запитаннями на електронну пошту викладача.

Результати надсилати на електронну адресу викладача [**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді cpp-файлу з іменем у форматі

**<Номер групи><Номер лабораторної><Прізвище англійською>**

Наприклад, 21-01Ivanov.cpp.

Іншим рішенням є надсилання поштою посилання на текст програми за URL адреси, яку надає C++Shell, вказавши в темі листа, номер групи прізвище студента та номер ПР.

В темі листа вказати, номер групи, прізвище студента та номер ПР як "ЛР№6".

**Строк відсилки ЛР для ІПЗ-21 25.10.2021**

**ІПЗ-22 26.10.2021**

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

**ОП+АМ-Запитання-<Номер групи>-<Прізвище >**.

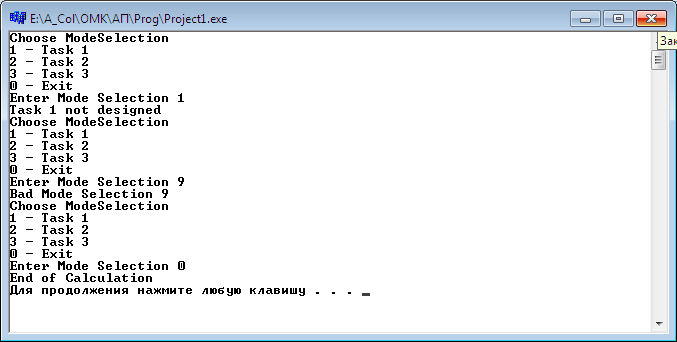
**Запитання**

1. Які оператори розгалуження ви знаєте?
2. Які оператори циклу ви знаєте?
3. Яка бібліотека С++ використовується для програмування математичних формул?
4. Як підключити математичні константи до програми С++?

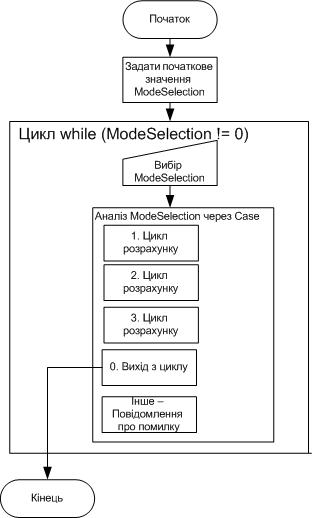
**Теоретичні відомості.**

Такого типу завдання доцільно виконувати використовуючи запит на обрання варіанту розрахунку (ModeSelection) через меню, а сам варіант розрахунку обирати через оператор вибору **switch** (вибір **case)**, і все це виконувати в циклі.

Приклад запиту нижче (оскільки розрахунок тут не зроблено, то видається повідомлення "Task not designed".



Всі три завдання – це розрахунки в циклі, які в разі обрання **case** для вибору оператору розрахунку потрібно вкладати у відповідну гілку. Схематично це представлено нижче (це не блок-схема, а ілюстрація).



Найбільшу трудність в ЛР№5 може викликати завдання 3.1. Розглянемо його.

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

 (1)

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

Проаналізуйте вираз підсумку та складіть компактну формулу розрахунку в циклі.

Такі задачі вимагають попереднього аналізу і складання компактної формули. Відповімо на запитання: що потрібно знайти? – Підсумок перших ***N*** членів. Тобто у нас повинен бути цикл, наприклад, по змінній **j,** яка змінюється від 1 до ***N.***

Для того, щоб виконувати розрахунок в циклі, нам потрібно "згорнути" вираз в (1) в компактну формулу, де фігурує **j.**

Подивимся уважно на вираз. Що ми бачимо в чисельнику. Ступень змінної **x** в кожному доданку збільшується на одиницю, тобто в узагальненому вигляді чисельник може бути записаний як **xj ,** де **j** змінюється від 1 до ***N.***

Тепер розглянемо знаменник: як він може залежати від **j**? Числа 3, 9, 27 – це 31, 32, 33. Тобто ми можемо знаменник в узагальненому вигляді записати як **x+3j,** де **j** змінюється від 1 до ***N.***

Тепер ми можемо записати формулу для розрахунку в циклі:

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

Як записується на мові С++ такий розрахунок?

…

**int n, j** **;**

**double x, S=0;**

**<Ввести x,n >**

**for (j = 1; j <= n; n++)**

**{ S=S+pow(x,j)/(x + pow(3.0, j);**

**<Вивести j та S з точністю до 3 десяткових знаків (тобто після "коми", в С++ - "крапки"), саме це означає запис ε=10-3>**

**}**

**…**

Текст в кутових дужках потрібно реалізувати у вигляді відповідного оператору введення - виведення.

**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 |

У бібліотеці **cmath** означено константи з іменами **M\_PI** (число π), **M\_PI\_2** (π /2), **M\_PI\_4** (π /4), **M\_1\_PI** (1/ π), **M\_E** (число *e*), **M\_LN2** (ln 2), **M\_LN10** (ln 10) і деякі інші. Щоб користуватися ними, необхідно перед підключенням бібліотеки **cmath** записати директиву

**#define \_USE\_MATH\_DEFINES** (define – означити).

**Приклад**. Надана програма виводить значення математичних констант π та *e*.

**#include <iostream>**

**#define \_USE\_MATH\_DEFINES**

**#include <cmath>**

**using namespace std;**

**int main() {**

**cout<<"pi="<<M\_PI<<endl;**

**cout<<"e="<<M\_E<<endl;**

**cout<<endl;**

**}**

**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”). Мітки 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).**

**Представлення даних в пам’яті комп'ютера**

https://studopedia.net/1\_905\_predstavlennya-danih-v-pamyati-kompyutera.html

Внутрішнє представлення змінної цілого типу — ціле число у двійковому коді. Згідно формату IEEE всі додатні цілі числа зберігаються в пам'яті комп'ютера в прямому коді, а всі від'ємні – в доповняльному коді. Цілі числа зберігаються в пам'яті комп'ютера у зворотньому порядку розміщення байт числа.

Приклад 3.

Додатнє число 25 типу **int** в пам'яті комп’ютера зберігається в прямому двійковому коді і займає 4 байти: 2510, 1916, 110012

0000 0000 0000 0000 0000 0000 0001 1001

В пам’яті комп’ютера *цілі числа зберігаються у зворотному порядку* розміщення байт числа:

0001 10001 0000 0000 0000 0000 0000 0000

Результат в 16-ковій системі числення: 19 00 00 00.

Приклад 4.

Від'ємне число -3500 типу **long int** в пам'яті комп’ютера зберігається в доповняльному двійковому коді і займає 4 байти: - 350010 = - DAC16 = - 1101 1010 11002

0000 0000 0000 0000 0000 1101 1010 1100 - прямий код;

1111 1111 1111 1111 1111 0010 0101 0011 - обернений код;

         +                                                             1

             1111 1111 1111 1111 1111 0010 0101 0100 - доповняльний код;

                F  F  F F  F  2   5  4 - в 16- ковій системі числення

 В пам’яті комп’ютера зберігається у зворотному порядку розміщення байт числа:

0101 0100 1111 0010 1111 1111 1111 1111

 Результат в 16- ковій системі числення: 54 F2 FF FF.

**Дійсні типи даних**

Дійсні константи записуються у двох формах – з фіксованою десятковою крапкою або в експонентному виді. В першому випадку крапка використовується для поділу цілої і дробової частин константи. Як ціла, так і дробова частини можуть бути відсутніми (наприклад 1.2, 0.725, 1., .35, 0.). В трьох останніх випадках відсутня або дробова, або ціла частина. Десяткова крапка повинна обов'язково бути присутньою, інакше константа буде вважатись цілою.

Експонентна форма запису дійсної константи містить знак, мантису і десятковий порядок (експоненту). Мантиса – це будь-яка додатня дійсна константа у формі з фіксованою крапкою або цілою константою. Порядок вказує степінь числа 10, на яку домножується мантиса. Порядок відокремлюється від мантиси буквою 'E' або 'e' (від слова exponent). Порядок може мати знак плюс або мінус, у випадку додатнього порядку знак плюс можна опускати. Наприклад:

1.5e+6 - константа еквівалентна 1500000.0

1e-4  - константа еквівалентна 0.0001

-.75E3 - константа еквівалентна -750.0

У мові С++ дійсні типи або типи з рухомою комою представляються трьома розмірами, що характеризують точність представлення дійсних чисел:

**float** – одиничної точності;

**double** - подвійної точності;

**long double** – розширеної точності (у деяких реалізаціях тип long double може бути відсутній)

Константи з рухомою комою мають за замовчуванням тип double. Саме він є найбільш природнім для комп'ютера. У програмуванні треба по можливості уникати типу float, тому що його точність недостатня, а процесор однаково при виконанні операцій перетворить його в тип double. Один з випадків, де застосування типу float виправдане – тривимірна комп'ютерна графіка.

Можна явно вказати тип константи за допомогою суфіксів F, f (float) і L, l (long). Наприклад, константа 2E+6L буде мати тип long double.

В  пам'яті  комп'ютера  змінна  типу float займає 4 байти, в яких один біт виділяється під знак, 8 – під порядок, 23 – під мантису.

Розряди мантиси включають один розряд цілої частини, що завжди дорівнює одиниці, і фіксовану кількість розрядів дробової частини. Оскільки старший двійковий розряд мантиси завжди дорівнює одиниці, зберігати його необов'язково, і у двійковому коді він відсутній. Фактично двійковий код зберігає тільки розряди дробової частини мантиси. Отже, насправді, у типу float мантиса містить 24 розряди, але старший розряд завжди дорівнює одиниці, тому зберігати його не потрібно.

Тип double займає 8 байт, у яких один розряд виділяється під знак, 11 – під порядок, 52 – під мантису. Насправді в мантисі 53 розряди, але старший завжди дорівнює одиниці і тому не зберігається.

Тип long double займає 10 байт (або 8 байт), в яких один розряд виділяється під знак, 15 – під порядок, інші 64 – під мантису. Записуються всі 64 розряди мантиси разом зі старшою одиницею.

Оскільки порядок може бути додатній і від'ємний, у двійковому коді він зберігається в **зміщеному виді**: до нього додається константа, яка рівна абсолютній величині максимального по модулю від'ємного порядку. У випадку типу float вона дорівнює 127, у випадку double – 1023, long double – 16383. Таким чином, максимальний по модулю від'ємний порядок представляється нульовим кодом.

Дійсні числа зберігаються в пам'яті комп'ютера у зворотньому порядку розміщення байт числа.

Таблиця 4.7

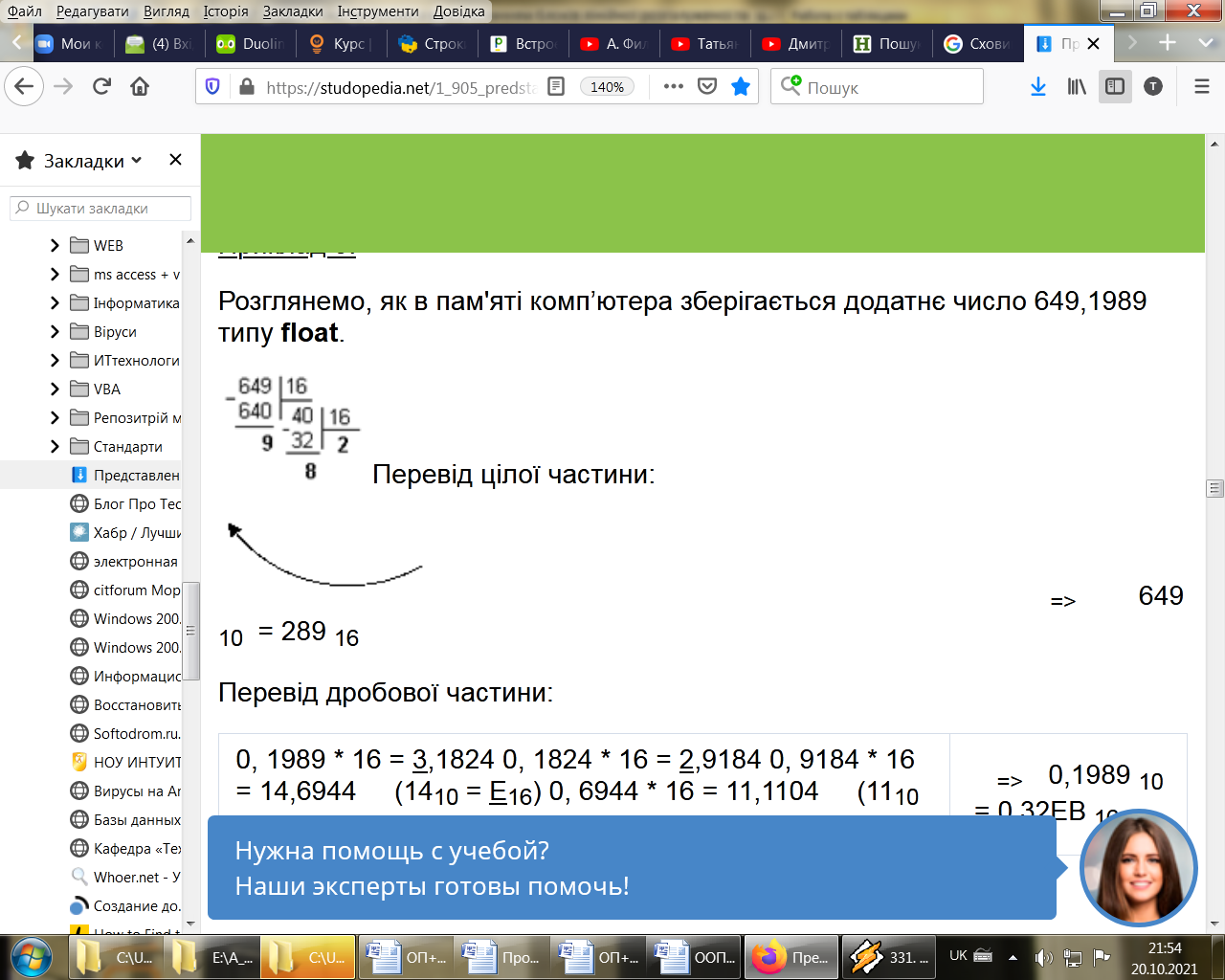
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Назва типу | Іденти-фікатор | Діапазон значень | Внутрішній формат: s–знак,e–експонента,m–мантиса | Значення числа | Розмір пам’яті в байтах |
| Дійсне одинарної точності | float | від 3.4 10-38 до 3.4 \*1038 | 1 біт 8 біт        23 біта   |  |  |  |  |  | | --- | --- | --- | --- | --- | | s | e |  | m |  | | (-1)S \*1,m \*2e -127 | 4 |
| Дійсне подвійної точності | double | від 1.7 10-308 до 1.7 \*10308 | 1 біт 11 біт       52 біта   |  |  |  |  |  | | --- | --- | --- | --- | --- | | s | e |  | m |  | | (-1)S \*1,m \*2e –1023 | 8 |
| Дійсне підвищеної точності | long double | від 3.4 10-4932 до 3.4 \*104932 | 1 біт 15 біт 1біт 63 біта   |  |  |  |  |  |  | | --- | --- | --- | --- | --- | --- | | s | e | 1 |  | m |  | | (-1)S \*1,m \*2e -16383 | 10 (8)\* |

\* Розмір типу long double для різних компіляторів може відрізнятися, наприклад для BCB 6.0 буде займати 10 байт, для VC++ 6.0 буде займати 8 байт (в цьому випадку формат збереження типу long double співпадає з форматом типу double).

Приклад 5.

Розглянемо, як в пам'яті комп’ютера зберігається додатнє число 649,1989 типу **float**.

Перевід цілої частини:



Перевід дробової частини:

|  |  |
| --- | --- |
| 0, 1989 \* 16 = 3,1824 0, 1824 \* 16 = 2,9184 0, 9184 \* 16 = 14,6944     (1410 = E16) 0, 6944 \* 16 = 11,1104     (1110 = B16) | => 0,1989 10  = 0,32EB 16 |

 Отже: 649,1989 10 = 289,32EB 16  = 0010 1000 1001 , 0011 0010 1110 1011 2

Нормалізація: 001 , 0 1000 1001 0011 0010 1110 1011 2 \* 101001

Заокруглення:

1 , 0 1000 1001 0011 0010 1110 10│11

+                                                           1

1 , 0 1000 1001 0011 0010 1110 11

Визначення мантиси: m=0 1000 1001 0011 0010 1110 11

Визначення зміщеного порядку: е = 12710 + 910  = 136 10  =  88 16  = 1000 1000 2

Інший спосіб: 12710   =    111  1111 2

                                   +           1001

                                      1000  1000 2

Визначення знакового розряду: s=0 (бо число додатнє).

Схема внутрішнього представлення:

|  |  |  |
| --- | --- | --- |
| s | e | m |
| 1 біт | 8 біт | 23 біт |

Зборка за схемою:

|  |  |  |
| --- | --- | --- |
| s | e | m |
| 0 | 1000 1000 | 0 1000 1001 0011 0010  1110 11 |

В 16- ковій системі числення: 0100 0100 0010 0010 0100 1100 1011 1011 2= 44 22 4С BB 16

В пам’яті комп’ютера буде зберігатися у зворотному порядку розміщення байт числа:

                                      1011  1011  0100  1100 0010 0010 0100 0100

Результат в 16- ковій системі числення: BB 4С 22 44

Приклад 6.

Розглянемо, як в пам'яті комп’ютера зберігається від'ємне число – 649,1989 типу **float**.

Різниця з попереднім прикладом буде полягати тільки у представленні знакового розряду.

Визначення знакового розряду: s=1 (бо число від'ємне).

Зборка за схемою (відміну від попереднього прикладу будемо виділяти жирним шріфтом):

|  |  |  |
| --- | --- | --- |
| s | e | m |
| 1 | 1000 1000 | 0 1000 1001 0011 0010 1110 11 |

В 16- ковій системі числення:  **1**100 0100 0010 0010 0100 1100 1011 10112 = **С**4 22 4С BB 16

В пам’яті комп’ютера буде зберігатися у зворотному порядку розміщення байт числа:

1011 1011 0100 1100 0010 0010 **1**100 0100

Результат в 16- ковій системі числення: BB 4С 22 **С**4