**Лабораторна робота №5.** Розробка простої програми з блоками, операціями порівняння при використанні математичних формул та операціями введення та виведення виразів.

**Мета**: навчитися писати прості програми лінійної та розгалуженої з використанням математичних функцій та виконанням операцій введення- виведення **на мові С++** в **консольному режимі.**

**Завдання.**

Написати на мові С++ програму введення даних для розрахунку формул, наданих в пп.1-3, та виведення результатів. Використати потокове введення та виведення даних, а також оператори вибору.

**Методичні вказівки щодо організації самостійної роботи студентів**

1. Скласти програму, яка складається з 4-х частин, кожна з яких визначається відповідним коментарем.
2. ***Перша частина***:

Виведітьє охайне повідомлення такого змісту: **Цю програму створив студент … групи … у … році**. Замість "**…**" вставте особисті дані (прізвище, ім'я та по-батькові, номер групи й рік написання програми).

1. ***Друга частина***: вирішити задачу

Лікарі рекомендують контролювати власну вагу Вашого тіла, керуючись значенням так званого індексу маси: ***k = m/h*2 *,*** де ***m* —** вага тіла в *кг***, *h*** — зріст в метрах**.** Якщо *k* <= 27, то все в нормі; якщо 27 < *k* <= 30, то має місце ожиріння; якщо же *k* > 30, то потрібне лікування від ожиріння. Скласти фрагмент програми, яка вводить по запиту Вашу вагу та зріст, а потім виводить на екран відповідний результат тестування.

1. ***Третя частина***:

Програма запитує у користувача його оцінку по 12-ти бальній шкалі та повідомляє оцінку у вигляді: відмінно, добре, задовільно або погано, a якщо введена оцінка не попадає до діапазону 2-12, видає повідомлення про помилку. Використати при написанні програми оператор **SWITCH**.

1. ***Четверта частина***:

Дійсні змінні **a**, **b**, **c** задають коефіцієнти рівняння *ax*2+*bx*+*c* = 0. Написати умову того, що воно:

а) має рівно два дійсні корені;

б) має єдиний дійсний корінь;

в) не має жодного розв'язку;

г) має нескінченно багато розв'язків.

Запитати коефіцієнти, перевірити умову, вивести рівняння з введеними коефіцієнтами та повідомлення щодо кількості коренів рівняння.

1. Перший рядок програми повинен містити коментар з номером ЛР, номером групи та прізвищем студента.
2. **В кінці теоретичної частини надані завдання для отримання додаткових балів.**

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, 31-01Ivanov.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 "ПР№4".

**Строк відсилки ЛР для ІПЗ-21 17.10.2021**

**ІПЗ-21 16.10.2021**

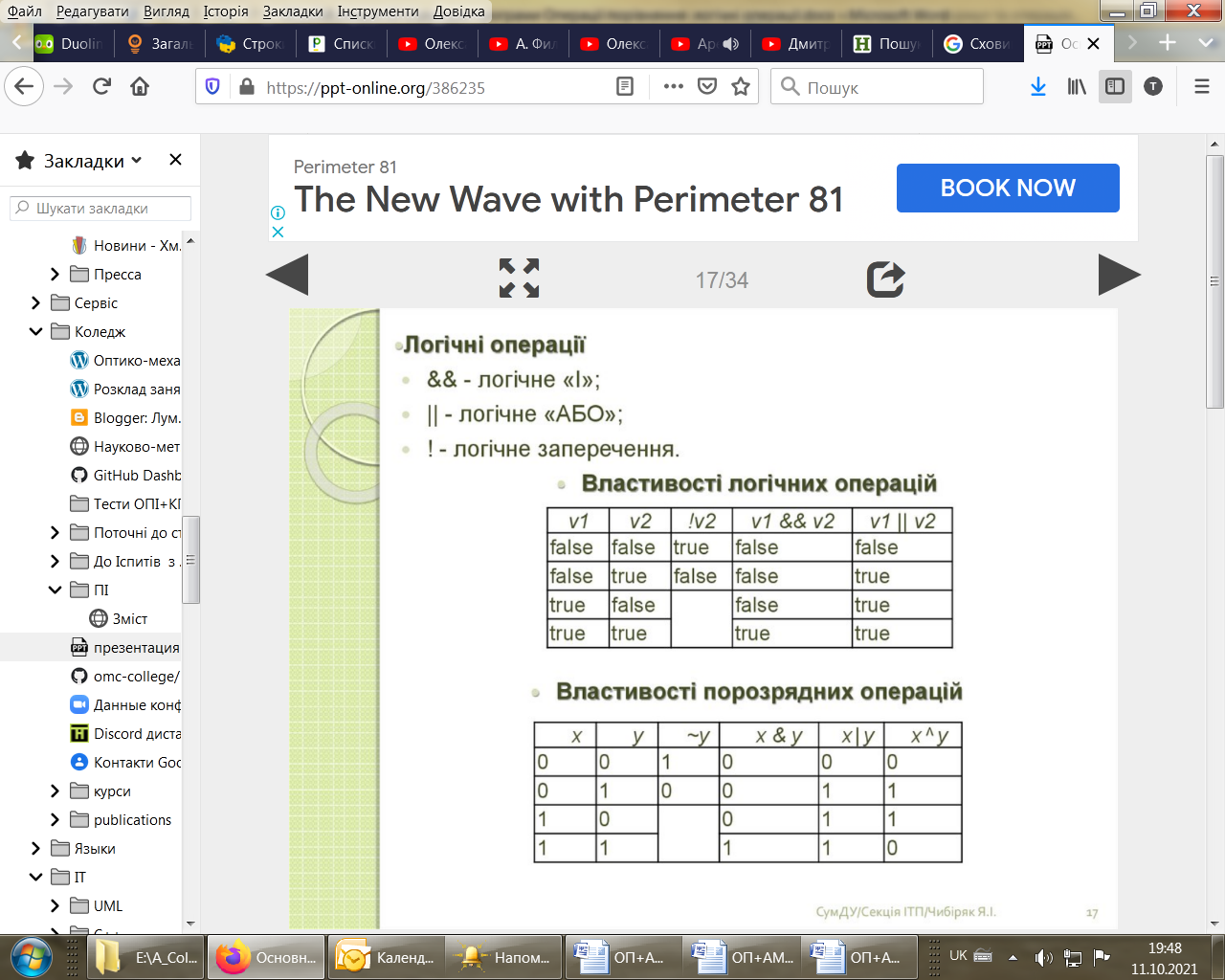
Всі запитання, що виникнуть, надсилайте на електронну адресу викладача, тему в заголовку листа записати

**ОРІ-Запитання-<Номер групи>-<Прізвище >**.

**ТЕОРЕТИЧНА ЧАСТИНА**

**Пригадаємо**

**Логічні операції**

****

**Тернарна операція (?:)**

Тернарна операція —це операція, що має 3 операнда. Використовується як альтернатива запису умовного оператора. Синтаксис тернарної операції:

**B ? V1 : V2.**

Виконання:

1. Спочатку обчислюється значення виразу B.

2. Якщо B==true(1), то обчислюється значення виразу V1, інакше - обчислюється значення виразу V2.

Для читабельності  рекомендується брати у круглі дужки умову та вирази. Приклади:

**(i < 1) ? 1 : i;**

**max = (a<=b) ? b : a;**

**Інструкції розгалуження**.

Інструкції розгалуження реалізують оператори вибору, до яких відносять оператор умовного переходу **if** та оператор-перемикач **switch**.

Оператор умовного переходу **if** використовується для розгалуження процесу обчислень на два напрямки і має такий два форми запису: повна та неповна.

Повна формамає вигляд

**if (*умова*) *інструкція1* else *інструкція2;***

**Оператор-перемикач switch** розглядається нижче за текстом.

Розглянемо питання, які дозволять розробити програму другої частини.

**Перерахування (перелічуваний тип даних)**

C++ дозволяє програмістам створювати свої власні (користувацькі) типи даних, зокрема Перелічуваний тип даних.

**Перерахування** (або ще “**перелічуваний тип даних**“) — це тип даних, де будь-яке значення (або ще “**енумератор/елемент**“) визначається як **символьна константа**. Оголосити перерахування можна за допомогою **ключового слова enum**. Наприклад:

// Оголошуємо нове перерахування Colors

**enum Colors**

**{**

// Нижче знаходяться енумератори

// Це всі можливі значення цього типу даних

// Кожний енумератор розділяється комою (НЕ крапкою з комою)

**COLOR\_RED,**

**COLOR\_BROWN,**

**COLOR\_GRAY,**

**COLOR\_WHITE,**

**COLOR\_PINK,**

**COLOR\_ORANGE,**

**COLOR\_BLUE,**

**COLOR\_PURPLE**

**};** // але сам enum повинен закінчуватися крапкою з комою

// Визначаємо декілька змінних перелічуваного типу Colors

**Colors paint = COLOR\_RED;**

**Colors house(COLOR\_GRAY);**

Оголошення перерахувань не вимагає виділення пам’яті. Тільки коли змінна перелічуваного типу визначена (наприклад, як змінна paint в прикладі вище), тільки тоді виділяється пам’ять для цієї змінної.

**Кожен енумератор розділяється комою, а саме перерахування закінчується крапкою з комою.**

Імена перерахувань часто починаються з великої літери, а імена елементів взагалі складаються тільки з великих букв. Оскільки елементи знаходяться в одному і тому ж **просторі імен**, що і саме перерахування, то одне і те ж ім’я одного енумератора не може бути використано в кількох перерахувань одного і того ж простору імен:

**enum Colors**

**{**

**YELLOW,**

**BLACK, // BLACK знаходиться в глобальному просторі імен**

**PINK**

**};**

**enum Feelings**

**{**

**SAD,**

**ANGRY,**

**BLACK**

**// отримаємо помилку, так як BLACK вже використовується в enum Colors**

**};**

Поширено додавання назви перерахування в якості префіксу до його елементів, наприклад: ANIMAL\_ чи COLOR\_ — як для запобігання **конфліктів імен**, так і з метою коментування коду.

**Значення енумераторів (елементу списку enum)**

Кожному енумератору автоматично присвоюється цілочисельне значення в залежності від його позиції в списку перерахування. За замовчуванням, першому елементу присвоюється ціле число 0, а кожному наступному — на одиницю більше, ніж попередньому:

**#include <iostream>**

**using namespace std;**

**enum Colors**

**{**

**COLOR\_YELLOW, // присвоюється 0**

**COLOR\_WHITE, // присвоюється 1**

**COLOR\_ORANGE, // присвоюється 2**

**COLOR\_GREEN, // присвоюється 3**

**COLOR\_RED, // присвоюється 4**

**COLOR\_GRAY, // присвоюється 5**

**COLOR\_PURPLE, // присвоюється 6**

**COLOR\_BROWN // присвоюється 7**

**};**

**int main()**

**{**

**Colors paint(COLOR\_RED);**

**cout << paint;**

**return 0;**

**}**

Результат виконання програми:

4

Можна й самому визначати значення елементів. Вони можуть бути як додатними, так і від’ємними, або взагалі мати аналогічні іншим елементам значення. Будь-які енумератори, що не визначені вами, — матимуть значення на одиницю більше, ніж значення попередніх елементів. Наприклад:

**// Визначаємо новий перелічуваний тип Animals**

**enum Animals**

**{**

**ANIMAL\_PIG = -4,**

**ANIMAL\_LION, // присвоюється -3**

**ANIMAL\_CAT, // присвоюється -2**

**ANIMAL\_HORSE = 6,**

**ANIMAL\_ZEBRA = 6, // має те ж значення, що і ANIMAL\_HORSE**

**ANIMAL\_COW // присвоюється 7**

**};**

ANIMAL\_HORSE і ANIMAL\_ZEBRA мають однакові значення. Хоча C++ це не забороняє, присвоювати одне значення декільком енумераторам в одному перерахуванні не рекомендується.

Порада**:** Не присвоюйте свої значення енумераторам.

**Правило: Не присвоюйте однакові значення двом енумераторам в одному перерахуванні, якщо на це немає вагомої причини.**

**Обробка перерахувань**

Оскільки значеннями енумераторів є цілі числа, то їх можна присвоювати цілочисельним змінним, а також виводити в консоль (як змінні **типу int**):

**#include <iostream>**

**using namespace std;**

**// Визначаємо новий перелічуваний тип Animals**

**enum Animals**

**{**

**ANIMAL\_PIG = -4,**

**ANIMAL\_LION, // присвоюється -3**

**ANIMAL\_CAT, // присвоюється -2**

**ANIMAL\_HORSE = 6,**

**ANIMAL\_ZEBRA = 6, // має те ж значення, що і ANIMAL\_HORSE**

**ANIMAL\_COW // присвоюється 7**

**};**

**int main()**

**{**

**int mypet = ANIMAL\_PIG;**

**cout << ANIMAL\_HORSE;**

**// конвертується в int, а потім виводиться на екран**

**return 0;**

**}**

Результат виконання програми:

6

Компілятор не буде неявно конвертувати цілочисельне значення в значення енумератора. Наступне викличе помилку компіляції:

**Animals animal = 7; // викличе помилку компіляції**

Компілятор також не дозволить вам вводити енумератори через **cin**:

**#include <iostream>**

**using namespace std;**

**enum Colors**

**{**

**COLOR\_PURPLE, // присвоюється 0**

**COLOR\_GRAY, // присвоюється 1**

**COLOR\_BLUE, // присвоюється 2**

**COLOR\_GREEN, // присвоюється 3**

**COLOR\_BROWN, // присвоюється 4**

**COLOR\_PINK, // присвоюється 5**

**COLOR\_YELLOW, // присвоюється 6**

**COLOR\_MAGENTA // присвоюється 7**

**};**

**int main()**

**{**

**Colors color;**

**cin >> color; // викличе помилку компіляції**

**return 0;**

**}**

Кожен перелічуваний тип вважається окремим типом. Отже, спроба присвоїти енумератор з одного перерахування енумератору з іншого — викличе помилку компіляції:

**Animals animal = COLOR\_BLUE; // викличе помилку компіляцїї**

# Виведення енумераторів

Спроба вивести енумератор за допомогою **cout** призведе до виведення цілочисельного значення самого енумератора (тобто його порядкового номера). Щоб вивести значення енумератора у вигляді тексту можна написати функцію з використанням операторів **if**:

**#include <iostream>**

**using namespace std;**

**enum Colors**

**{**

**COLOR\_PURPLE, // присвоюється 0**

**COLOR\_GRAY, // присвоюється 1**

**COLOR\_BLUE, // присвоюється 2**

**COLOR\_GREEN, // присвоюється 3**

**COLOR\_BROWN, // присвоюється 4**

**COLOR\_PINK, // присвоюється 5**

**COLOR\_YELLOW, // присвоюється 6**

**COLOR\_MAGENTA // присвоюється 7**

**};**

**void printColor(Colors color)**

**{**

**if (color == COLOR\_PURPLE)**

**cout << "Purple";**

**else if (color == COLOR\_GRAY)**

**cout << "Gray";**

**else if (color == COLOR\_BLUE)**

**cout << "Blue";**

**else if (color == COLOR\_GREEN)**

**cout << "Green";**

**else if (color == COLOR\_BROWN)**

**cout << "Brown";**

**else if (color == COLOR\_PINK)**

**cout << "Pink";**

**else if (color == COLOR\_YELLOW)**

**cout << "Yellow";**

**else if (color == COLOR\_MAGENTA)**

**cout << "Magenta";**

**else**

**cout << "Who knows!";**

**}**

**Виділення пам’яті для перерахувань**

Перелічувані типи вважаються частиною сімейства цілочисельних типів, і компілятор сам визначає, скільки пам’яті виділяти для змінних типу enum. C++ повідомляє, що розмір перерахування повинен бути досить великим, щоб мати можливість вмістити всі енумератори. Але найчастіше розміри змінних enum будуть такими ж, як і розміри звичайних змінних типу int.

Оскільки визначення перерахування саме по собі не вимагає виділення пам’яті і якщо перерахування необхідно використовувати в декількох файлах, то його можна визначити в **заголовковому файлі** і підключати цей заголовок всюди, де необхідно використовувати перерахування.

**Користь від перерахувань**

Перелічувані типи корисні для документації коду і поліпшення читабельності. Наприклад, функції часто повертають цілі числа назад в викликаючу функцію в якості кодів помилок, якщо щось пішло не так. Як правило, невеликі від’ємні числа використовуються для представлення можливих кодів помилок. Наприклад:

**int readFileContents()**

**{**

**if (!openFile())**

**return -1;**

**if (!parseFile())**

**return -2;**

**if (!readFile())**

**return -3;**

**return 0; // якщо все пройшло успішно**

**}**

Однак **магічні числа**, як в прикладі вище, — не дуже ефективне рішення. Альтернативним рішенням є використання перерахувань:

**enum ParseResult**

**{**

**SUCCESS = 0,**

**ERROR\_OPENING\_FILE = -1,**

**ERROR\_PARSING\_FILE = -2,**

**ERROR\_READING\_FILE = -3**

**};**

**ParseResult readFileContents()**

**{**

**if (!openFile())**

**return ERROR\_OPENING\_FILE;**

**if (!parseFile())**

**return ERROR\_PARSING\_FILE;**

**if (!readfile())**

**return ERROR\_READING\_FILE;**

**return SUCCESS; // якщо все пройшло успішно**

**}**

Це і читати легше, і зрозуміти простіше. Крім того, функція, яка викликає іншу функцію, може перевірити значення, що повертається на відповідний енумератор. Це краще, ніж самому порівнювати результат, який повертається, з конкретними цілочисельними значеннями, щоб зрозуміти яка саме помилка сталася. Наприклад:

**if (readFileContents() == SUCCESS)**

**{**

**// Робимо що-небудь**

**}**

**else**

**{**

**// Виводимо повідомлення про помилку**

**}**

Перелічувані типи найкраще використовувати при визначенні набору пов’язаних ідентифікаторів. Наприклад, припустимо, що ви пишете гру, в якій гравець може мати один предмет, але цей предмет може бути декількох різних типів. Або, якщо ви пишете функцію для сортування групи значень:

**enum SortType**

**{**

**SORTTYPE\_FORWARD,**

**SORTTYPE\_BACKWARDS**

**};**

**void sortData(SortType type)**

**{**

**if (type == SORTTYPE\_FORWARD)**

**// сортування даних в одному порядку**

**else if (type == SORTTYPE\_BACKWARDS)**

**// сортування даних в зворотному порядку**

**}**

# Оператор switch

Оператор **switch** — **оператор управління потоком виконання програми.** При запуску програми, центральний процесор (ЦП) починає виконання коду з першого рядка функції main(), виконуючи певну кількість операторів, а потім завершує виконання при завершенні блоку main(). Послідовність операторів, які виконує ЦП, називається **порядком виконання програми** (або ще “**потоком виконання програми**“).

**Навіщо використовувати оператор switch?**

Хоча ми і можемо використовувати відразу декілька **операторів if/else** разом — виглядає це не дуже читабельно. Наприклад:

**#include <iostream>**

**using namespace std;**

**enum Colors**

**{**

**COLOR\_GRAY,**

**COLOR\_PINK,**

**COLOR\_BLUE,**

**COLOR\_PURPLE,**

**COLOR\_RED**

**};**

**void printColor(Colors color)**

**{**

**if (color == COLOR\_GRAY)**

**cout << "Gray";**

**else if (color == COLOR\_PINK)**

**cout << "Pink";**

**else if (color == COLOR\_BLUE)**

**cout << "Blue";**

**else if (color == COLOR\_PURPLE)**

**cout << "Purple";**

**else if (color == COLOR\_RED)**

**cout << "Red";**

**else**

**cout << "Unknown";**

**}**

**int main()**

**{**

**printColor(COLOR\_BLUE);**

**return 0;**

**}**

Використання розгалуження **if/else** для перевірки значення однієї змінної — практика поширена, але C++ надає альтернативний і більш ефективний умовний оператор розгалуження — **switch**. Ось та ж програма, що вище, але вже з використанням оператора switch:

**#include <iostream>**

**using namespace std;**

**enum Colors {**

**COLOR\_GRAY,**

**COLOR\_PINK,**

**COLOR\_BLUE,**

**COLOR\_PURPLE,**

**COLOR\_RED**

**};**

**void printColor(Colors color)**

**{**

**switch (color)**

**{**

**case COLOR\_GRAY:**

**cout << "Gray";**

**break;**

**case COLOR\_PINK:**

**cout << "Pink";**

**break;**

**case COLOR\_BLUE:**

**cout << "Blue";**

**break;**

**case COLOR\_PURPLE:**

**cout << "Purple";**

**break;**

**case COLOR\_RED:**

**cout << "Red";**

**break;**

**default:**

**cout << "Unknown";**

**break;**

**}**

**}**

**int main()**

**{**

**printColor(COLOR\_BLUE);**

**return 0;**

**}**

Загальна ідея операторів switch проста: вираз оператора switch (наприклад, switch(color)) має генерувати значення, а кожен випадок (або ще “**case**“) перевіряє це значення на відповідність. Якщо case збігається з виразом switch-а, то виконуються інструкції під відповідним кейсом. Якщо case не відповідає виразу switch-а, то виконуються інструкції після кейса default (якщо його взагалі вказали). Через свою реалізацію, оператори switch зазвичай більш ефективні, ніж ланцюжки if/else.

**Оператор switch**

Спочатку пишеться **ключове слово switch**, за яким слідує вираз, з яким ми хочемо працювати. Зазвичай цим виразом є окрема змінна, але це може бути і щось більш складне, наприклад, nX + 2 чи nX − nY. Єдине обмеження до цього виразу — воно повинно бути інтегрального типу даних (тобто char, short, int, long, long long чи enum). Змінні **типу з плаваючою крапкою** або НЕ інтегральні типи використовуватися не можуть.

Після виразу switch ми оголошуємо блок. Усередині блоку ми використовуємо мітки (англ. “**labels**“) для визначення всіх значень, які ми хочемо перевіряти на відповідність з виразом. Існує два типи міток: case і default.

**Мітки case**

Перший вид міток — це **case** (або просто “**кейс**“), який оголошується з використанням **ключового слова case** і має константний вираз. Константний вираз — це той вираз, який генерує константне значення, іншими словами це **літерал** (наприклад, 5), **перерахування** (наприклад, COLOR\_RED) чи **константа** (наприклад, змінна x, яка була оголошена з ключовим словом const).

Константний вираз, що знаходиться після ключового слова case, перевіряється на рівність з виразом, що знаходяться біля ключового слова switch. Якщо вони збігаються, то тоді виконується код під відповідним кейсом.

Всі вирази case повинні генерувати унікальні значення. Тобто ви не зможете зробити наступне:

**switch (z)**

**{**

**case 3:**

**case 3:  // заборонено, значення 3 вже використовується!**

**case COLOR\_PURPLE: // заборонено, COLOR\_PURPLE обчислюється як 3!**

**};**

**Мітка default**

Другий тип мітки — це **мітка за замовчуванням** (так званий “**default case**“), який оголошується з використанням **ключового слова default**. Код під цією міткою виконується, якщо жоден з кейсів не відповідає виразу switch. Мітка за замовчуванням є необов’язковою. В одному switch може бути тільки один default. Зазвичай його оголошують останнім в блоці switch.

**switch і fall-through**

Одна з найбільш каверзних речей в switch — це послідовність виконання коду. Коли відбувся збіг з кейсом (або виконується default), то виконання починається з першого оператору, який знаходиться після відповідного кейса і триває до тих пір, поки не буде виконано одну з наступних умов завершення:

   Досягнуто кінець блоку switch.

   Виконується оператор return.

   Виконується оператор break.

Зверніть увагу, якщо жодної з цих умов завершення не буде, то виконуватися будуть всі кейси після того кейса, який співпав з виразом switch. Наприклад:

**switch (2)**

**{**

**case 1: // Не співпало!**

**cout << 1 << '\n'; // пропускається**

**case 2: // Співпало!**

**cout << 2 << '\n'; // виконання коду починається тут**

**case 3:**

**cout << 3 << '\n'; // це також виконується**

**case 4:**

**cout << 4 << '\n'; // і це**

**default:**

**cout << 5 << '\n'; // і це**

**}**

Результат:

2  
3  
4  
5

А ось це точно не те, що нам потрібно! Коли виконання переходить з одного кейсу в наступний, то це називається **fall-through**. Програмісти майже ніколи не використовують fall-through, тому в окремих випадках, коли це все-таки використовується — програміст залишає **коментар,** в якому повідомляє, що fall-through є навмисним.

**switch і оператор break**

**Оператор break** (оголошений з використанням **ключового слова break**) повідомляє компілятору, що ми вже зробили все, що хотіли з певним switch-ом (або циклом while, do while чи for) і більше не маємо наміру з ним працювати. Коли компілятор зустрічає оператор break, то виконання коду переходить зі switch-а на наступний рядок після блоку switch. Розглянемо приклад вище, але вже з коректно вставленими операторами break:

**switch (2)**

**{**

**case 1: // не співпало - пропускається**

**cout << 1 << '\n';**

**break;**

**case 2:**

**// співпало!  Виконання починається з наступного оператору**

**cout << 2 << '\n'; // виконання починається тут**

**break;**

**// оператор break завершує виконання switch-а**

**case 3:**

**cout << 3 << '\n';**

**break;**

**case 4:**

**cout << 4 << '\n';**

**break;**

**default:**

**cout << 5 << '\n';**

**break;**

**}**

**// Виконання продовжується тут**

Оскільки другий кейс відповідає виразу switch, то виводиться 2, і оператор break завершує виконання блоку switch. Решта кейсів пропускаються.

***Попередження:*** Не забувайте використовувати оператор break в кінці кожного кейсу. Його відсутність — одна з найпоширеніших помилок новачків!

**Декілька операторів всередині блоку switch**

Ще одна особливість switch полягає в тому, що ви можете використовувати декілька операторів під кожним кейсом, не визначаючи новий блок:

**switch (3)**

**{**

**case 3:**

**cout << 3;**

**boo();**

**cout << 4;**

**break;**

**default:**

**cout << "default case\n";**

**break;**

**}**

**Оголошення змінної і її ініціалізація всередині кейсу**

Ви можете оголосити, але не ініціалізувати змінні всередині блоку case:

**switch (x)**

**{**

**case 1:**

**int z; // ок, оголошення дозволено**

**z = 5; // ок, операція присвоювання дозволена**

**break;**

**case 2:**

**z = 6;**

**// ок, змінна z була оголошена вище, тому ми можемо //використовувати її тут**

**break;**

**case 3:**

**int c = 4;**

**// заборонено, ви не можете ініціалізувати змінні**

**// всередині case**

**break;**

**default:**

**cout << "default case" << endl;**

**break;**

**}**

Зверніть увагу, що, хоча змінна z була визначена в першому кейсі, вона також використовується і в другому кейсі. Всі кейси вважаються частиною однієї і тієї ж **області видимості**, тому, оголосивши змінну в одному кейсі, ми можемо спокійно використовувати її без оголошення і в інших кейсах.

Це може здатися трохи нелогічним, тому давайте розглянемо це детальніше. Коли ми визначаємо **локальну змінну**, наприклад, int y;, то змінна не створюється в цій точці — вона фактично створюється на початку блоку, в якому її оголосили. Однак, в програмі її не видно до точки оголошення. Саме оголошення не виконується, воно просто повідомляє компілятору, що змінна вже може використовуватися в коді. Тому змінна, оголошена в одному кейсі, може використовуватися в іншому кейсі, навіть якщо кейс, що оголошує змінну, ніколи не виконається.

Проте ініціалізація змінних безпосередньо в кейсах заборонена і викличе помилку компіляції. Це пов’язано з тим, що ініціалізація змінної вимагає виконання, а case, що містить ініціалізацію, — може ніколи не виконатися!

Якщо в кейсі потрібно оголосити і/або ініціалізувати нову змінну, то це найкраще зробити, використовуючи[**блок операторів**](https://acode.com.ua/urok-50-bloky-stejtmentiv/) всередині кейсу:

**switch (1)**

**{**

**case 1:**

**{ // зверніть увагу, що тут вказано блок операторів**

**int z = 5;**

**//добре, змінні можна ініціалізувати**

**//всередині блоку, який знаходиться всередині кейсу**

**cout << z;**

**break;**

**}**

**default:**

**cout << "default case" << std::endl;**

**break;**

**}**

**Правило: Якщо потрібно ініціалізувати і/або оголосити змінні всередині кейсу, то використовуйте блоки операторів.**

**1. Завдання (для отримання додаткових 3 балів)**

Визначте перерахування (або **клас enum**) Animal, яке містить наступних тварин:

   pig

   chicken

   goat

   cat

   dog

   ostrich

Напишіть функцію getAnimalName(), яка приймає Animal в якості параметра і використовує switch для повернення типу тварини в вигляді **рядка**. Напишіть ще одну функцію printNumberOfLegs(), яка використовує switch для виведення кількості лапок відповідного типу тварини. Переконайтеся, що обидві функції мають кейс default, який виводить повідомлення про помилку. Викличте printNumberOfLegs() в main(), використовуючи в якості параметрів cat і chicken.

Приклад результату виконання програми:

A cat has 4 legs.  
A chicken has 2 legs.

**2. Завдання (для отримання додаткового балу)**

Заливкою блакитного кольору визначте правильні відповіді для а) та б) :

А) Енумераторам можна:

* присвоювати цілочисельні значення;
* не присвоювати значення;
* явно присвоювати значення **типу з плаваючою крапкою**;
* присвоювати значення попередніх енумераторів (наприклад, COLOR\_BLUE = COLOR\_GRAY).

Б) Енумератори можуть бути:

* від’ємними;
* не унікальними.