

Специфікатори C++ та спеціальні оператори

Лекція **4**

ЗМІСТ

1. Специфікатор типу `const`
2. Специфікатор типу `volatile`
3. Специфікатори класів пам'яті
4. Статичні змінні
5. Регістрові змінні

Література:

1. Бублик В.В. Об'єктно-орієнтоване програмування: [Підручник] / В.В. Бублик. – К.: ІТ-книга, 2015. – 624 с.
2. Вступ до програмування мовою С++. Організація обчислень : навч. посіб. / Ю. А. Белов, Т. О. Карнаух, Ю. В. Коваль, А. Б. Ставровський. – К. : Видавничо-поліграфічний центр "Київський університет", 2012. – 175 с.
3. Зубенко В.В., Омельчук Л.Л. Програмування. Поглиблений курс. – К.: Видавничо-поліграфічний центр "Київський університет", 2011. - 623 с.
4. Страуструп Бьярне. Программирование: принципы и практика с использованием С++, 2-е изд. : Пер. с англ. - М. : ООО "И.Д. Вильямс", 2016. - 1328 с.
5. Шилдт Г. С++: базовый курс, 3-е издание. : Пер. с англ. – М. : Издательский дом «Вильямс», 2010. – 624 с.
6. Stroustrup, Bjarne. The C++ programming language. — Fourth edition. – Addison-Wesley, 2013. – 1361 pp.



Специфікатор типу `const`

Змінні, оголошені з використанням специфікатора `const`, не можуть змінити свої значення під час виконання програми. Однак будь-якій `const`-змінній можна присвоїти деяке початкове значення.

```
const double version = 3.2;
```

Будь-яка `const`-змінна одержує значення або явною ініціалізацією або при використанні апаратно-залежних засобів.

Застосування специфікатора `const` до оголошення змінної гарантує, що вона не буде модифікована іншими частинами вашої програми.

1. Специфікатор `const` найчастіше використовують для створення `const`-параметрів типу покажчик.

Такий параметр-покажчик захищає об'єкт, на який він посилається, від модифікації з боку функції.

Якщо параметр-покажчик випереджається ключовим словом `const`, ніяка інструкція цієї функції не може модифікувати змінну, що адресується цим параметром.

```
#include <iostream>
using namespace std;
void code(const char *str);
int main()
{
    setlocale(LC_ALL, ""); // Локалізація виводу
    code("Це тест."); // Буде надруковано: Чж!ужту/
    return 0;
}
/* Використання специфікатора const гарантує, що str не може змінити
аргумент, на який він вказує. */
void code(const char *str)
{
    while(*str) {
        cout << (char) (*str+1);
        str++;
    }
}
```

// Цей код невірний.

```
void code(const char *str)
{
    while(*str) {
        *str = *str + 1; // Помилка, аргумент модифікувати не можна.
        cout << (char) *str;
        str++;
    }
}
```

Специфікатор `const` найчастіше використовують

2. Для посилальних параметрів, щоб не допустити у функції модифікацію змінних, на які посилаються ці параметри.

// Не можна модифікувати `const`-посилання.

```
#include <iostream>
using namespace std;
void f(const int &i);
int main()
{
    int k = 10;
    f(k);
    return 0;
}
```

// Використання посилального `const`-параметра.

```
void f (const int &i)
{
    i = 100; // Помилка, не можна модифікувати const-посилання.
    cout << i;
}
```

Специфікатор `const` найчастіше використовують

3. Для підтвердження того, що ваша програма не змінює значення деякої змінної.

Змінна типу `const` може бути модифікована зовнішніми пристроями, тобто її значення може бути встановлено яким-небудь апаратним пристроєм (наприклад, датчиком). Оголосивши змінну за допомогою специфікатора `const`, можна гарантувати, що будь-які зміни, яким піддається ця змінна, викликані винятково зовнішніми подіями.

Специфікатор `const` найчастіше ВИКОРИСТОВУЮТЬ

4. Для створення іменованих констант. Часто в програмах багаторазово застосовується те саме значення для різних цілей.

```
#include <iostream>
using namespace std;
const int size = 10;
int main()
{
    int A1[size], A2[size], A3[size];
    // . . .
}
```

Специфікатор типу `volatile`

- Специфікатор `volatile` інформує компілятор про те, що дана змінна може бути змінена зовнішніми (відносно програми) факторами.
- Наприклад, припустимо, що у наступному фрагменті програми змінна `clock` обновляється кожну мілісекунду годинниковим механізмом комп'ютера.

```
int clock, timer; // змінна clock обновляється апаратно системним годинником
// ...
timer = clock; // рядок А: clock отримує значення, яке присвоюється змінній timer
/* ... тут інструкції, що не містять жодного явного присвоювання нового значення
змінній clock, тому оптимізуючий компілятор зафіксує значення clock, отримане у рядку А
*/
cout << "Час, що минув " << clock-timer;
// рядок Б: використовується зафіксоване значення clock
```

Специфікатор типу `volatile`

- Для рішення цієї проблеми необхідно оголосити змінну `clock` із ключовим словом `volatile`.

```
volatile int clock;
```

- Тепер значення змінної `clock` буде опитуватись при кожному її використанні.
- Специфікатори `const` і `volatile` можна використовувати разом. Наприклад, наступне оголошення абсолютно припустиме. Воно створює `const`-показчик на `volatile`-об'єкт.

```
const volatile unsigned char *port = (const volatile char *) 0x2112;
```

- У цьому прикладі для перетворення цілочисельного літерала `0x2112` в `const`-показчик на `volatile`-символ необхідно застосувати операцію приведення типів.

Специфікатори класів пам'яті

C++ підтримує п'ять специфікаторів класів пам'яті:

- **auto**
 - **extern**
 - **register**
 - **static**
 - **mutable**
- Специфікатори класів пам'яті визначають, як повинна зберігатися змінна.
 - Специфікатор класів пам'яті необхідно вказувати на початку оголошення змінної.
 - Специфікатор **mutable** застосовується тільки до об'єктів класів, про які мова буде пізніше.

Специфікатори класів пам'яті

Специфікатор класу пам'яті **auto**

- Специфікатор **auto** оголошує локальну змінну. Використовується досить рідко, оскільки локальні змінні є "автоматичними" за замовчуванням.

Специфікатор класу пам'яті **extern**

- Програма C++ може включати тільки одну копію кожної глобальної змінної. Тому у багатофайлових проектах всі глобальні змінні оголошуються в одному файлі, а в інших використовуються **extern**-оголошення. Специфікатор **extern** робить змінну відомою для модуля повторно не виділяючи для неї пам'яті.

Специфікатор класу пам'яті **extern**

Специфікатор **extern** оголошує змінну, але не виділяє для неї пам'яті.

```
#ifndef F1_H_INCLUDED
#define F1_H_INCLUDED
int x = 123, y;
char ch;
#endif // F1_H_INCLUDED
```

```
#include <iostream>
#include "f1.h"
using namespace std;
extern int x, y;
extern char ch;
int func2(int y)
{
    x=x/y;
    return x;
}
int func3()
{
    y=10;
    return y;
}

int main()
{
    cout << func2(func3()) << endl;
    return 0;
}
```

Вихід



Специфікатор класу пам'яті **extern**

- Специфікатор `extern` робить змінну відомою для модуля повторно не виділяючи для неї пам'яті.
- Зі змінною асоціюються дві величини: **власне значення** (r-значення) – те, що зберігається, і **значення адреси пам'яті** (l-значення) – місце, де зберігається.
- При оголошенні змінної їй присвоюється ім'я й тип, а за допомогою визначення для змінної виділяється пам'ять.
- У більшості випадків оголошення змінних одночасно є визначеннями.
- Специфікатор `extern` дозволяє оголосити змінну, не визначаючи неї.

Специфікатор класу пам'яті **extern**

- Якщо функція використовує глобальну змінну, котра визначається нижче (у тому ж файлі), у тілі функції її можна специфікувати як зовнішню (за допомогою ключового слова **extern**).

```
#include <iostream>
```

```
using namespace std;
```

```
int main()
```

```
{
```

```
    extern int first, last; //Використання глобальних змінних.
```

```
    cout << first << " " << last << "\n";
```

```
    return 0;
```

```
}
```

```
int first = 10, last = 20; //Глобальне визначення змінних first і last.
```

Специфікатор класу пам'яті `extern`

- Якщо компілятор виявляє змінну, котра не була оголошена в поточному блоці, він перевіряє, чи не збігається вона з якою-небудь зі змінних, оголошених раніше в одному з охоплюючих блоків.
- Якщо ні, компілятор переглядає раніше оголошені глобальні змінні. Якщо виявляється збіг їхніх імен, компілятор припускає, що посилання було саме на цю змінну.
- **Специфікатор `extern` необхідний тільки у випадку, коли треба використовувати змінну, котра оголошується або нижче в тому ж файлі, або в іншому.**
- Незважаючи на те що специфікатор `extern` оголошує, але не визначає змінну, існує одне виключення із цього правила.
- **Якщо в `extern`-оголошенні змінна ініціалізується, то таке `extern`-оголошення стає визначенням.**
- **Це дуже важливий момент: будь-яка змінна може мати кілька оголошень, але тільки одне визначення.**

Статичні змінні

Змінні типу `static` - це змінні "довгострокового" зберігання, тобто вони зберігають свої значення в межах своєї функції або файлу. Від глобальних вони відрізняються тим, що за рамками своєї функції або файлу вони невідомі.

Локальні `static`-змінні

Локальна `static`-змінна зберігає своє значення між викликами функції. Для неї виділяється постійна область пам'яті практично так само, як і для глобальної змінної. Ключове розходження між **статичною локальною і **глобальною** змінними полягає в тому, що статична локальна змінна відома тільки блоку, у якому вона оголошена.**

```
static int count;
```

Статичній змінній можна присвоїти деяке початкове значення.

```
static int count = 200;
```

Локальні `static`-змінні ініціалізуються тільки один раз, на початку виконання програми, а не при кожному вході у функцію, у якій вони оголошені.

Можливість використання статичних локальних змінних важлива для створення незалежних функцій, оскільки існують такі типи функцій, які повинні зберігати значення деяких своїх змінних між викликами.

Статичні змінні

- Змінні типу `static` - це змінні "довгострокового" зберігання, тобто вони зберігають свої значення в межах своєї функції або файлу.
- Від глобальних вони відрізняються тим, що за рамками своєї функції або файлу вони невідомі.

Локальні static-змінні

- Локальна static-змінна зберігає своє значення між викликами функції.

Для неї виділяється постійна область пам'яті практично так само, як і для глобальної змінної.

Ключове розходження між **статичною локальною** і **глобальною** змінними полягає в тому, що статична локальна змінна відома тільки блоку, у якому вона оголошена.

```
static int count;
```

- Статичній змінній можна присвоїти деяке початкове значення.

```
static int count = 200;
```

- Локальні static-змінні ініціалізуються тільки один раз, на початку виконання програми, а не при кожному вході у функцію, у якій вони оголошені.
- Можливість використання статичних локальних змінних важлива для створення незалежних функцій, оскільки існують такі типи функцій, які повинні зберігати значення деяких своїх змінних між викликами.

//зберігання поточного середнього значення від чисел, що вводяться користувачем

```
#include <iostream>
using namespace std;
int r_avg(int i);
int main()
{
    int num;
    setlocale(LC_ALL, ""); // Локалізація виводу
    do {
        cout << "Уведіть числа (-1 означає вихід): ";
        cin >> num;
        if(num != -1)
            cout << "Поточне середнє дорівнює: " << r_avg(num);
        cout << '\n';
    } while(num > -1);
    return 0;
}
```

```
// Обчислюємо поточне середнє.
```

```
int r_avg(int i)
```

```
{
```

```
    static int sum = 0, count = 0;
```

```
    sum = sum + i;
```

```
    count++;
```

```
    return sum / count;
```

```
}
```

Глобальні static-змінні

- Глобальна static-змінна відома тільки у файлі, в якому вона оголошена.
- Іншим функціям в інших файлах вона не відома і вони не можуть змінити її вміст.


```
// Перший файл
```

```
#include <iostream>
using namespace std;
int r_avg(int i); void reset();
int main(){
    int num;
    do {
        cout<<"Уведіть числа (-1 вихід, -2 скидання):";
        cin >> num;
        if(num==-2) {
            reset();
            continue;    }
        if(num != -1)
            cout << "Середнє значення дорівнює: "
                << r_avg(num);
        cout << '\n';
    }while(num != -1);
    return 0;}
```

Вихід



```
// Другий файл
#include <iostream>
static int sum=0, count=0;
int r_avg(int i) {
    sum = sum + i;
    count++;
    return sum / count;
}
void reset() {
    sum = 0;
    count = 0;
}
```

Важливо!

- Незважаючи на те що глобальні static-змінні як і раніше широко використовуються в C++-кодi, стандарт C++ заперечує проти їхнього застосування.
- Для управління доступом до глобальних змінних рекомендується інший метод, що полягає у використанні просторів імен.

Регістрові змінні

- Специфікатор `register` в оголошенні змінної означає вимогу оптимізувати код для одержання максимально можливої швидкості доступу до неї.

```
int signed_pwr(register int m, register int e)
{
    register int temp;
    int sign;
    if(m < 0) sign = -1;
    else sign = 1;
    temp = 1;
    for( ; e; e--) temp = temp * m;
    return temp * sign;
}
```

```
/* Вплив використання register-змінної на швидкість виконання програми. */
```

```
#include <iostream>
```

```
#include <ctime>
```

```
#include <windows.h>
```

```
using namespace std;
```

```
unsigned int i;
```

```
unsigned int delay;
```

```
int main()
```

```
{
```

```
    register unsigned int j;
```

```
    long start, end;
```

```
    SetConsoleOutputCP(1251);
```

```
    start = clock();
```

```
for(delay=0; delay<50; delay++)
    for(i=0; i<64000000; i++);
end = clock();
cout << "Кількість тиків для не register-циклу: ";
cout << end-start << "\n";
start = clock();
for(delay=0; delay<50; delay++)
    for(j=0; j<64000000; j++);
end = clock();
cout << "Кількість тиків для register-циклу: ";
cout << end-start << "\n";
return 0;
}
```

Походження модифікатора `register`

- Модифікатор `register` був уперше визначений у мові C. Спочатку він застосовувався тільки до змінних типу `int` і `char` або до **показчиків** і змушував зберігати змінні цього типу в регістрі ЦП, а не в ОЗП, де зберігаються звичайні змінні.
- Відповідно до ANSI-стандарту C модифікатор `register` можна застосовувати до **будь-якого типу даних**.
- *Середовище Visual C++, ігнорує ключове слово `register`. Visual C++ застосовує оптимізацію "як вважає за потрібне". Починаючи із стандарту C++11 вважається застарілим підходом.*

Запитання?

Вихід

