

## Інкапсуляція

Навіщо робити змінні-члени класу закритими?

Як відповідь, скористаємося аналогією. У сучасному житті ми маємо доступ до багатьох електронних пристроїв. До телевізору є пульт дистанційного керування, за допомогою якого можна вмикати / вимикати телевізор. Керування автомобілем дозволяє в разі швидше пересуватися між двома точками. За допомогою фотоапарата можна робити знімки.

Все з цих трьох речей використовують загальний шаблон: вони надають простий інтерфейс для вас (кнопка, кермо і т.д.) для виконання певної дії. Однак, те, як ці пристрої фактично працюють, приховано від вас (як від користувачів). Для натискання кнопки на пульті дистанційного керування вам не потрібно знати, що виконується в середині пульта для взаємодії з телевізором. Коли ви натискаєте на педаль газу в своєму автомобілі, вам не потрібно знати про те, як двигун внутрішнього згоряння надає руху колеса. Коли ви робите знімок, вам не потрібно знати, як датчики збирають світло в піксельний зображення.

Такий поділ інтерфейсу і реалізації надзвичайно корисно, оскільки воно дозволяє використовувати об'єкти, без необхідності розуміння їх реалізації. Це значно знижує складність використання цих пристроїв і значно збільшує їх кількість (пристрої з якими можна взаємодіяти).

З аналогічних причин поділ реалізації та інтерфейсу корисно і в програмуванні.

В об'єктно-орієнтованому програмуванні **інкапсуляція** (або ще «**приховування інформації**») - це процес прихованого зберігання деталей реалізації об'єкта. Користувачі звертаються до об'єкта через відкритий інтерфейс.

У C++ інкапсуляція реалізована через **специфікатор доступу**. Як правило, всі змінні-члени класу є закритими (приховуючи деталі реалізації), а більшість методів є відкритими (з відкритим інтерфейсом для користувача). Хоча вимога до користувачів використовувати публічний інтерфейс може здатися більш обтяжливим, ніж просто відкрити доступ до змінних-членам, але, насправді, це надає велику кількість корисних переваг, які покращують можливість повторного використання коду і його підтримку.

Глобальні змінні небезпечні, так як немає строгого контролю над тим, хто має до них доступ і як їх використовують. Класи з відкритими членами мають ту ж проблему, тільки в менших масштабах. Наприклад, припустимо, що нам потрібно написати строковий клас. Ми могли б почати таким способом:

```
1 class MyString
2 {
3     char *m_string; // динамически выделяем строку
4     int m_length; // используем переменную для отслеживания длины строки
5 };
```

Ці дві змінні пов'язані: `m_length` завжди повинен відповідати довжині рядка, утримуваної `m_string`. Якби `m_length` був би відкритим, то будь-хто міг би змінити довжину рядка без зміни `m_string` (або навпаки). Це точно призвело б до проблем. Роблячи як `m_length`, так і `m_string` закритими, користувачі змушені використовувати методи для взаємодії з класом.

Ми також можемо самі поліпшити захист нашого класу від неправильного використання. Наприклад, розглянемо клас з відкритою змінною-членом у вигляді масиву :

```
1 class IntArray
2 {
3 public:
4     int m_array[10];
5 };
```

Якби користувачі могли б безпосередньо звертатися до масиву, то вони могли б використовувати неприпустимий індекс:

```
1 int main()
2 {
3     IntArray array;
4     array.m_array[16] = 2; // некорректный индекс, вследствие чего перезаписываем память, которой мы
5 }
```

Однак, якщо ми зробимо масив закритим, то зможемо змусити користувача використовувати функцію, яка, в першу чергу, перевіряє коректність індексу:

```
1 class IntArray
2 {
3 private:
4     int m_array[10]; // пользователь не имеет прямого доступа к этому члену
5
6 public:
7     void setValue(int index, int value)
8     {
9         // Если индекс недействителен, то не делаем ничего
10        if (index < 0 || index >= 10)
11            return;
12
13        m_array[index] = value;
14    }
15 };
```

Таким чином, ми захистимо цілісність нашої програми. Розглянемо наступний простий приклад:

```

1  #include <iostream>
2
3  class Values
4  {
5  public:
6      int m_number1;
7      int m_number2;
8      int m_number3;
9  };
10
11 int main()
12 {
13     Values value;
14     value.m_number1 = 7;
15     std::cout << value.m_number1 << '\n';
16 }

```

Хоча ця програма працює нормально, але що станеться, якщо ми вирішимо перейменувати `m_number1` або змінити тип цієї змінної? Ми б зламали не тільки цю програму, а й більшу частину програм, які використовують клас `Values`!

Інкапсуляція надає можливість зміни способу реалізації класів, не порушуючи при цьому роботу всіх програм, які їх використовують. Ось інкапсульована версія класу вище, але яка використовує методи для доступу до `m_number1`:

```

1  #include <iostream>
2
3  class Values
4  {
5  private:
6      int m_number1;
7      int m_number2;
8      int m_number3;
9
10 public:
11     void setNumber1(int number) { m_number1 = number; }
12     int getNumber1() { return m_number1; }
13 };
14
15 int main()
16 {
17     Values value;
18     value.setNumber1(7);
19     std::cout << value.getNumber1() << '\n';
20 }

```

Тепер давайте змінимо реалізацію класу:

```

1 #include <iostream>
2
3 class Values
4 {
5 private:
6     int m_number[3]; // здесь изменяем реализацию этого класса
7
8 public:
9     // Нам нужно обновить переменные методов, чтобы заработала новая реализация
10    void setNumber1(int number) { m_number[0] = number; }
11    int getNumber1() { return m_number[0]; }
12 };
13
14 int main()
15 {
16     // Но наша программа продолжает работать как прежде
17     Values value;
18     value.setNumber1(7);
19     std::cout << value.getNumber1() << '\n';
20 }

```

Зверніть увагу, оскільки ми не змінювали прототипи яких-небудь функцій у відкритому інтерфейсі нашого класу, наша програма, яка використовує клас, продовжує працювати без будь-яких змін або проблем.

### Функції доступу

Залежно від класу, може бути доречним мати можливість отримувати / встановлювати значення закритих змінних-членів класу.

**Функція доступу** - це коротка відкрита функція, завданням якої є отримання або зміна значення закритою змінної-члена класу, наприклад:

```

1 class MyString
2 {
3 private:
4     char *m_string; // динамически выделяем строку
5     int m_length; // используем переменную для отслеживания длины строки
6
7 public:
8     int getLength() { return m_length; } // функция доступа для получения значения m_length
9 };

```

Тут `getLength()` є функцією доступу, яка просто повертає значення `m_length`. Функції доступу зазвичай бувають двох типів :

- **Геттери** - це функції, які повертають значення закритих змінних-членів класу.
- **Сетери** - це функції, які дозволяють присвоювати значення закритих змінних-членів класу.

Ось приклад класу, який використовує геттери і сеттери для всіх своїх закритих змінних-членів:

```
1 class Date
2 {
3     private:
4         int m_day;
5         int m_month;
6         int m_year;
7
8     public:
9         int getDay() { return m_day; } // геттер для day
10        void setDay(int day) { m_day = day; } // сеттер для day
11
12        int getMonth() { return m_month; } // геттер для month
13        void setMonth(int month) { m_month = month; } // сеттер для month
14
15        int getYear() { return m_year; } // геттер для year
16        void setYear(int year) { m_year = year; } // сеттер для year
17    };
```

**Правило:** Надавайте функції доступу тільки в тому випадку, коли потрібно, щоб користувач мав можливість отримувати або привласнювати значення членам класу.

**Правило:** Геттера повинні використовувати тип повернення за значенням або за константною посиланням.