**Лабораторна робота №6.** **Розподіл поведінки та реалізації**.

**Мета:** Набуття навичок в розробці найпростіших класів та роботі з об’єктами класів, встановлення селекторів та модифікаторів.

**Порядок виконання роботи**

1. Повторити синтаксис оголошення класу, специфікатори доступу, опис функцій та їх прототипів, використання файлів.
2. Ознайомитися з теоретичними відомостями.
3. Продовжити працювати над програмою минулої лабораторної роботи. Доповнити клас *Person* селекторами та модифікаторами, які зробити вбудованими в клас.
4. Надати коментарі до всіх членів класу.
5. В першому рядку програми та заголовкового файлу повинні бути записаними в коментарі номер групи та прізвище, а також номер ЛР (через кому до попередньої).
6. Результати надсилати на електронну адресу викладача

[**t.i.lumpova@gmail.com**](mailto:t.i.lumpova@gmail.com)у вигляді заголовкового файлу та cpp-файлу з іменем у форматі

Для заголовкового файлу

**<Прізвище англійською>.h**

Наприклад, Ivanov.h

Для cpp-файлу:

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

Наприклад, 31-01Ivanov.cpp.

Тему в заголовку листа записати

**ООП<Номер групи>-ЛР <Номер лабораторної>-<Прізвищеанглійською>**

**Строк відсилки ЛР ІПЗ-31 15.10.2021**

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

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

**Що повинно бути зроблено за попередню ЛР №5.**

1. Опис інтерфейсу, що визначає поведінку, надати в заголовковому файлі (h-файлі), причому методи-члени класу для операцій заповнення членів-даних значеннями по замовчуванню (для початкового тестування), консольне введення, де код особи задається програмно, і виведення значень екземплярів класу *Person* в інтерфейсі надаються у вигляді прототипу, а реалізація надається в cpp-файлі.
2. В головній програмі надати реалізацію методів, створити меню, аналогічне розробленим в попередніх ЛР з позиціями:

"interactive data entry about the person" (інтерактивне введення даних про особу) (1),

"interactive data entry about the teacher" (інтерактивне введення даних про викладача) (2),

«interactive data entry about the student» (інтерактивне введення даних про студента) (3),

"reading from a data file about persons " (читання з файлу даних про осіб) (4),

"reading from the teacher data file " (читання з файлу даних про викладачів) (5),

"reading from a data file about students " (читання з файлу даних про студентів) (6),

«calculation of the average score» (розрахунок середнього балу) (7),

"sorting teacher data by phone number" (сортування за номером телефону даних про викладачів) (8),

"sorting student data by phone number" (сортування за номером телефону даних про студентів) (9).

"***Exit***" ("Вихід").

По незадіяним позиціям меню надати повідомлення "Not designed".

1. Встановити перевірки щодо виклику позицій меню, наприклад, виконання позицій щодо обчислення середніх балів та сортування інформації по студентах виконуються при виконанні позицій щодо введення відповідних даних. При невиконанні умов надати підказку щодо необхідності введення даних.

Це повинна бути НОВА програма, в якій нема функціональності та структур попередніх ЛР. Алгоритми збережених попередніх ЛР будуть використані в процесі реалізації можливостей ООП.

При реалізації п.3 можливі алгоритмічні труднощі, що визначення алгоритму аналізу. Для вирішення цього питання доцільно скористатися таблицями рішень, які використовуються також і для тестування алгоритмів обрання потрібної гілки розгалуження в програмі.

**Таблиця прийняття рішень** (**Decision Table -** таблиця рішень) - спосіб компактного представлення моделі зі складною логікою. Аналогічно умовних операторів мови програмування, вони встановлюють зв'язок між умовами і діями. Але, на відміну від традиційних мов програмування, таблиці рішень можуть представляти зв'язок між безліччю незалежних умов і дій в елегантній формі.

Таблиця рішень використовується для представлення умовної логіки шляхом створення списку завдань із зображенням правил бізнес-рівня. Таблиці рішень можуть бути використані, коли є послідовна кількість умов, які необхідно оцінити та призначити певний набір дій, які слід використовувати, коли умови нарешті виконані. У таблиці прийняття рішень перераховані причини (стан бізнес-правил) та наслідки (дія ділових правил та очікувані результати), які представлені за допомогою використання матриці, де кожен стовпець являє собою унікальну комбінацію.

Таблиці прийняття рішень, як правило, поділяються на чотири квадрати, як показано нижче.

|  |  |
| --- | --- |
| Умови | Варіанти виконання умов |
| Дії | Необхідність дій |

У найпростішому випадку тут Умови - список можливих умов, Варіанти виконання умов - комбінація з виконання та/або невиконання умов цього списку. Дії - список можливих дій, Необхідність дій - вказівка треба чи не треба виконувати відповідну дію для кожної з комбінацій умов. Наприклад для ситуації **"світло згасло"**

|  |  |  |  |
| --- | --- | --- | --- |
| Світло в сусідній кімнаті горить | Так | Немає | Немає |
| Світло у сусідів горить | - | Так | Немає |
| Поміняти лампочку | Х |  |  |
| Перевірити пробки |  | Х |  |
| Зателефонувати електрику |  | Х | Х |
| Зателефонувати диспетчеру |  |  | Х |

Дії можуть бути елементарними або посилатися на інші таблиці прийняття рішень. Необхідність виконання дій може бути невпорядкованою, як в даному прикладі, або впорядкованій. В останньому випадку якщо при певній комбінації виконання умов можливе виконання декількох дій, то в таблиці рішень вказується їх пріоритет.

Як складати таблицю

* По горизонталі — виписуємо умови, які впливають на результат. А нижче — сам результат, далі – дія (Action), яку потрібно виконати.
* По вертикалі — правила: конкретна комбінація вхідних умов.

Тобто ми вказуємо значення умови та результату

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Правило 1** | **Правило 2** | **...** | **Правило N** |
| **Умови** |  |  |  |  |
| Умова 1 |  |  |  |  |
| Умова 2 |  |  |  |  |
| ... |  |  |  |  |
| Умова N |  |  |  |  |
|  |  |  |  |  |
| **Дія/Результат** |  |  |  |  |
| Дія 1 |  |  |  |  |
| Дія 2 |  |  |  |  |
| ... |  |  |  |  |
| Дія N |  |  |  |  |

**Приклад 1. (російськомовне джерело) Страховка на автомобиль (один результат)**

Я прихожу в страховую компанию и заполняю анкету, где есть 2 вопроса:

1. Есть ли 5 лет стажа вождения?
2. Была ли в авариях?

Ответить можно либо да, либо нет.

Получается 2 условия по 2 возможных варианта, итого 4 варианта пересечения условий, 4 правила. На каждое правило свой результат:

* Если у меня небольшой стаж и я часто бываю в авариях — придется заплатить по максимуму, иначе страховать такого водителя будет невыгодно.
* Если нет стажа, но нет аварий — плачу поменьше, но не сильно. Знаете как бывает — первое время катаются очень осторожно, а потом начинают думать «да я царь и бог, не попаду в аварию». И понеслось...
* Если я опытный водитель, но бываю в авариях — ценник еще чуть ниже. Ведь бывать в авариях — это нормально. Иногда ты просто стоишь на светофоре, а в тебя влетает дурак, ну что тут поделаешь? Но если аварий мало, а опыта много — это хороший знак.
* Если опытный, да еще и без аварий — меньше всего. Очень аккуратный водитель, платить скорее всего не придется!

А теперь то же самое, только в виде таблички:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | **Правило 1** | **Правило 2** | **Правило 3** | **Правило 4** |
| **Условия** |  |  |  |  |
| Стаж 5 лет | Нет | Нет | Да | Да |
| Был в авариях? | Да | Нет | Да | Нет |
|  |  |  |  |  |
| **Результат** |  |  |  |  |
| Страховка | 200 руб | 100 руб | 50 руб | 10 руб |

**Приклад 2. (російськомовне джерело)**

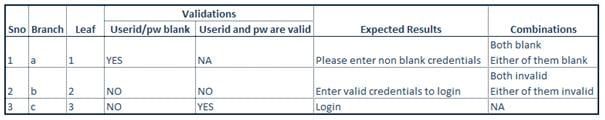
Рассмотрим *таблицу принятия решений* на примере страницы регистрации нового пользователя сервиса KUKU.io

Используем понятия “корректные” и “некорректные” данные.

Чтобы регистрация прошла успешно, необходимо заполнить корректными оба поля. Если поля заполняются некорректными данными, то система должна выдать ошибку: “Введены невалидные данные”.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Условие** | **Значения 1** | **Значения 2** | **Значения 3** | **Значения 4** |
| Ввод корректных данных в поле E-mail | + | — | + | — |
| Ввод корректных данных в поле Password | + | — | — | + |
| Ввод некорректных данных в поле E-mail | — | + | — | + |
| Ввод некорректных данных в поле Password | — | + | + | — |
| **Действия** |  |  |  |  |
| Регистрация прошла успешно | + | — | — | — |
| Выдается ошибка: “Введены невалидные данные” | — | + | + | + |

Значения 2, 3, 4 приводят к одному и тому же результату с разными входными значениями.



**ТЕОРЕТИЧНІ ВІДОМОСТІ**

Як розглядалося раніше, розділення інтерфейсу та реалізації визначено концепцією інкапсуляції, що заснована на розділенні того, як об’єкт виглядає (його інтерфейс), і того, як він в дійсності працює (його реалізації).

Відповідно, визначення деяких функцій-членів класу варто розміщати за межами його оголошення, залишаючи всередині лише їх прототипи. Остання вимога обов'язкова! Саме це робить конкретну функцію членом класу.

Для того щоб зв'язати визначення функції-члена з класом, якому вона належить, перед її ім'ям вказується ім'я класу й оператор дозволу області видимості ::. Розглянемо програму, де надане визначення функції**-**члена поза класом:

#include <iostream>

#include <stdio.h>

#include <malloc.h>

using namespace std;

class TComplex // Інтерфейс

{system("color F0");

double Re;

double Im;

public:

void print();

void init();

};

int main()

{

TComplex\* p = (TComplex\*)malloc(sizeof(TComplex));

// Z.Re = 1; Помилка! Член Re знаходиться в закритому розділі!

// Z.Im = 2; Помилка! Член Im знаходиться в закритому розділі!

p->init(); // Виклик функції-члена init()

p->print(); // Виклик функції-члена print()

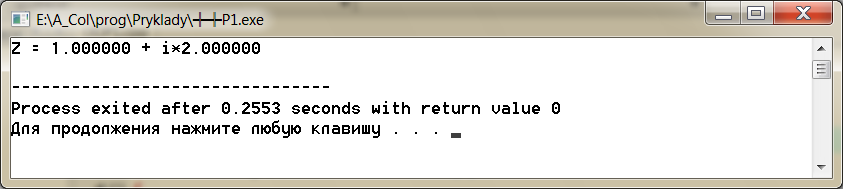
return 0;

}

// Реалізація

void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}

void TComplex::init() {Re = 1; Im = 2;}



**void \* malloc( size\_t sizemem );**

Функція malloc виділяє блок пам'яті, розміром sizemem байт, і повертає покажчик на початок блоку.

Зміст виділеного блоку пам'яті не инициализируется, воно залишається з невизначеними значеннями.

Для відділення інтерфейсу від реалізації перенесемо інтерфейс в заголовковий файл Untitled11-1.h, тоді для використання програма повинна виглядати так:

#include <stdio.h>

#include <malloc.h>

#include "Untitled11-1.h"

void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}

void TComplex::init() {Re = 1; Im = 2;}

int main()

{

TComplex\* p = (TComplex\*)malloc(sizeof(TComplex));

// Z.Re = 1; Помилка! Член Re знаходиться в закритому розділі!

// Z.Im = 2; Помилка! Член Im знаходиться в закритому розділі!

p->init(); // Виклик функції-члена init()

p->print(); // Виклик функції-члена print()

return 0;

}

Для того, щоб зробити функції вбудованими, якщо вони винесені за межі опису інтерфейсу, достатньо їм надати специфікатор inline:

**inline** void TComplex::print() {printf("Z = %lf + i\*%lf \n",Re,Im);}

**inline** void TComplex::init() {Re = 1; Im = 2;}

Зауважимо, якщо функції доступу (селектори та модифікатори) складаються з одного рядка присвоювання, то їх можна залишити в тілі класу (в інтерфейсі).

**Правило одного визначення** (**One Definition Rule, ODR**) — важлива концепція в мові програмування C++, що визначена в ISO C++ Standard(ISO/IEC 14882) 2003, в розділі 3.2.

Коротко ODR стверджує:

1. В окремій одиниці трансляції (файлі, якщо конкретна реалізація зберігає тексти програм в файлах, після обробки препроцесором) шаблон, клас, функція, об'єкт, або перерахування можуть мати не більше одного визначення. Хоча деякі можуть мати яку завгодно кількість оголошень.
2. В програмі об'єкт або невбудована (не-inline) функція не можуть мати більш ніж одне визначення.
3. Деякі сутності, наприклад, класи, шаблони або вбудовані (inline) функції можуть мати більше ніж одне визначення тільки якщо:
   1. вони знаходяться в різних одиницях трансляції;
   2. вони ідентичні лексема за лексемою;
   3. значення лексем однаково в обох одиницях трансляції;

Компілятори не завжди знаходять порушення ODR. Багато з них виявляються вже [компонувальником](https://uk.wikipedia.org/wiki/%D0%9A%D0%BE%D0%BC%D0%BF%D0%BE%D0%BD%D1%83%D0%B2%D0%B0%D0%BB%D1%8C%D0%BD%D0%B8%D0%BA).

Класи — це користувальницькі типи даних, які звільнюються від оголошення тільки в одному місці. Саме тому клас, визначений в заголовковому файлі, можна підключати в інші файли.

Методи, визначені всередині тіла класу, вважаються неявно вбудованими (inline). Вбудовані функції також звільнюються від правила оголошення тільки в одному місці. А це означає, що проблем з визначенням простих методів (таких як функції доступу) всередині самого класу виникати не повинно.

Методи, визначені поза тіла класу, розглядаються, як звичайні функції, та повинні підкорятися правилу одного визначення, тому ці функції повинні бути визначені в файлі .cpp, а не всередині .h. Єдиним виключенням є шаблони функцій.

**Параметри по замовчуванню** для методів повинні бути оголошені в тілі класу (в заголовковому файлі), де вони будуть видні всім, хто підключає цей заголовковий файл з класом.

Розділення інтерфейсу (оголошення) класу та його реалізації дуже поширене в бібліотеках, які використовуються для розширення можливостей програми, наприклад, заголовкові файли зі Стандартної бібліотеки С++ (iostream, string, vector, array тощо). Програма потребує тільки оголошення з заголовкових файлів, щоб компілятор зміг перевірити коректність коду у відповідності до правил синтаксису мови C++. Реалізації класів зі Стандартної бібліотеки С++ знаходяться в попередньо скомпільованому файлі, який додається компонувальником.

Чому недоцільно вносити в інтерфейс реалізацію методів:

1. Це приведе до перенавантаженню визначення класу.
2. Функції, визначені всередині тіла класу, вважаються неявно вбудованими, що сприяє "розбуханню" коду.
3. Якщо в заголовковий файл вносяться зміни, то потрібно перекомпілювати всі програми, що містять цей заголовок. Якщо змінено код в файлі .cpp, то потрібно перекомпілювати тільки цей файл .cpp!

**Тому** **рекомендується**:

  Класи, що використовуються тільки в одному файлі, і які повторно не використовуються, визначати безпосередньо в файлі .cpp, де вони використовуються.

  Класи, що використовуються в декількох файлах або призначені для повторного використання, визначайте в заголовковому файлі с тим же ім’ям, що у класу.

   Тривіальні методи (звичайні конструктори або [**деструктори**](https://ravesli.com/urok-120-destruktory/), функції доступу тощо) визначайте всередині тіла класу.

   Нетривіальні методи визначайте в файлі .cpp з тим же іменем, що у класу.

**Контрольні запитання**

* 1. Що таке екземпляр класу?
  2. Що таке інтерфейс класу та його реалізація?
  3. Що відносять до даних-членів класу?
  4. Які специфікатори доступу використовують для даних-членів класу?
  5. Вкажіть способи опису класів.
  6. Що таке закриті члени класу?
  7. Що таке відкриті члени класу?
  8. Як організувати доступ до закритих членів класу?
  9. Скільки об'єктів класу може бути використано в програмі, яка застосовує клас?
  10. Які специфікатори доступу використовують для даних-членів класу?
  11. Чим відрізняються функції – члени класу від функцій, не пов’язаних з будь-яким класом?