**Міністерство освіти і науки України**
**Національний університет “Львівська політехніка”**
**Інститут комп’ютерних наук та інформаційних технологій**

***Кафедра ПЗ***

**Звіт**

до лабораторної роботи №9

на тему «Принцип поліморфізму»

з курсу «Об’єктно-орієнтоване програмування»

**Виконав:**

студент групи ПЗ-11

Ясногродський Нікіта

**Перевірила**:

доц. Коротєєва Т.О.

Львів

**2022**

**Тема.** Наслідування. Створення та використання ієрархії класів.

**Мета.** Навчитись створювати списки об’єктів базового типу, що включають об’єкти похідних типів. Освоїти способи вирішення проблеми неоднозначності при множинному наслідуванні. Вивчити плюси заміщення функцій при множинному наслідуванні. Навчитись використовувати чисті віртуальні функції, знати коли варто використовувати абстрактні класи.

**Теоретичні відомості:**

Поліморфізм – одна з трьох основних парадигм ООП. Якщо говорити коротко, поліморфізм – це здатність об'єкта використовувати методи похідного класу, який не існує на момент створення базового.

В ооп часто створюються ієрархії логічно пов’язаних класів. Наприклад уявимо клас Shape, від якого наслідуються класи Rectangle і Circle. Пізніше від класу Rectangle наслідується клас Square, як окремий вид прямокутників. В класі Shape немає змісту визначати функції знаходження площі чи знаходження периметру, оскільки в похідних класах вони будуть перевизначені. Всі методи повинні функціонувати нормально в похідних класах а не в базовому класі Shape, оскільки неможливо створити екземпляр форми як такий. Також програма повинна бути захищена від спроби користувача створити об’єкт цього класу.

C++ підтримує створення абстрактних типів даних з чистими віртуальними функціями. Чисті віртуальні функції це такі, які ініціалізуються нульовим значенням, наприклад

virtual void Draw() = 0;

Клас, який містить чисті віртуальні функції є абстрактним. Неможливо створити об’єкт абстрактного класу. Поміщення в клас чистої віртуальної функції означає наступне:

- неможливо створити об’єкт цього класу;
- необхідно замістити чисту віртуальну функцію в похідному класі.

Будь-який клас, наслідуваний від абстрактного класу, наслідує від нього чисту віртуальну функцію, яку необхідно замістити щоб отримати можливість створювати об’єкти цього класу.

Зазвичай чисті віртуальні функції оголошуються в абстрактному базовому класі і не виконуються. Оскільки неможливо створити об’єкт абстрактного базового класу, як правило, нема необхідності і в виконання чистої віртуальної функції.

Тим не менше, інколи виникає необхідність виконання чистої віртуальної функції. Вона може бути викликана з об’єкта, який наслідує абстрактний клас,

наприклад, щоб забезпечити загальну функціональність для всіх заміщених функцій

**Завдання для лабораторної роботи:**

1. Розробити ієрархію класів відповідно до варіанту.
2. Використати множинне наслідування, продемонструвати вирішення проблеми з неоднозначністю доступу до членів базових класів за допомогою віртуального наслідування, за допомогою явного звертання до членів класу та за допомогою заміщення функцій в похідному класі (при потребі).
3. Створити списки об’єктів базового типу, в них помістити об’єкти похідного типу. Продемонструвати виклик функцій з об’єктів – елементів списку. Використати опратор dynamic_cast (при потребі).
4. Створити абстрактний клас, використати чисто віртуальну функцію, що містить реалізацію в базовому класі.
5. Для вивільнення динамічної пам’яті використовувати віртуальні деструктори.
6. Сформувати звіт до лабораторної роботи. Відобразити в ньому діаграму наслідування класів.

Варіант 13.

## Звичайний масив(однозвязний список)

**Результати:**

“Array.cpp”

```
#ifndef __ARRAY_CPP
#define __ARRAY_CPP


#include "array.h"


template <typename T>
std::string ArrayInfoPrinter<T>::get_info_abstract() const {
  return "General Info";
};


template <typename T>
std::string ArrayInfoPrinter<T>::get_info() const {
  return "Simple info";
};
```

```
template <typename T>
int BasicArray<T>::length() const {
  return this->elements.size();
}

template <typename T>
bool BasicArray<T>::includes(T const val) const {
  for (T i : this->elements) {
    if (i == val) {
      return true;
    }
  }
  return false;
}

template <typename T>
void BasicArray<T>::push(T const& val) {
  this->elements.push_back(val);
}

template <typename T>
std::string BasicArray<T>::get_info_abstract() const {
  return ArrayInfoPrinter<T>::get_info_abstract() + " from BasicArray";
}

template <typename T>
SinglyLinkedArray<T>::SinglyLinkedArray() : head(nullptr) {}

template <typename T>
SinglyLinkedArray<T>::~SinglyLinkedArray() {
  Node* i = this->head;
  while (i != nullptr) {
    i = i->next;
    delete i;
    delete i->data;
```

```
  }
}

template <typename T>
int SinglyLinkedArray<T>::length() const {
  int count = 0;
  for (auto i = this->head; i != nullptr; i = i->next) {
    count++;
  }
  return count;
}

template <typename T>
void SinglyLinkedArray<T>::push(T const& val) {
  auto newNode = new Node(val);

  if (head == nullptr) {
    head = newNode;
    return;
  }

  auto i = head;
  while (i->next != nullptr) i = i->next;
  i->next = newNode;
}

template <typename T>
bool SinglyLinkedArray<T>::includes(T const val) const {
  auto i = head;
  while (i != nullptr) {
    if (i->data == val) return true;
  }
  return false;
}

template <typename T>
```

```
std::string SinglyLinkedArray<T>::get_info_abstract() const {
  return ArrayInfoPrinter<T>::get_info_abstract() + " from SinglyLinkedArray";
}


#endif
```

“widget.cpp”

```
#include "widget.h"

#include <QFile>
#include <QGridLayout>
#include <QTextStream>

#include "array.h"

void Widget::on_output() {
  const int count = 5;

  Array<double> *arrays[count];

  for (auto i = 0; i < count; i++) {
    if (i % 2) {
      arrays[i] = new BasicArray<double>();
    } else {
      arrays[i] = new SinglyLinkedArray<double>();
    }
  }

  this->class_names_output->setMarkdown(
      QString("### Basic Array:\n\n"
              "* %1\n"
              "* %2")
          .arg(QString::fromStdString(arrays[1]->get_info()))
          .arg(QString::fromStdString(arrays[1]->get_info_abstract())));

  this->results_output->setMarkdown(
      QString("### SinglyLinkedArray\n"
              "* %1\n"
              "* %2")
          .arg(QString::fromStdString(arrays[0]->get_info()))
          .arg(QString::fromStdString(arrays[0]->get_info_abstract())));
}

Widget::Widget(QWidget *parent) : QWidget(parent) {
  auto *main_layout = new QGridLayout;

  this->output_btn = new QPushButton("Print output");

  this->class_names_output = new QTextEdit;
  this->class_names_output->setReadOnly(true);

  this->results_output = new QTextEdit;
  this->results_output->setReadOnly(true);

  main_layout->addWidget(this->class_names_output, 0, 0);
  main_layout->addWidget(this->output_btn, 0, 1);
  main_layout->addWidget(this->results_output, 0, 2);
```

```
  connect(this->output_btn, &QPushButton::released, this, &Widget::on_output);

  setLayout(main_layout);
}
```

“main.cpp”

```
#include "mainwindow.h"

#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();
    return a.exec();
}
```

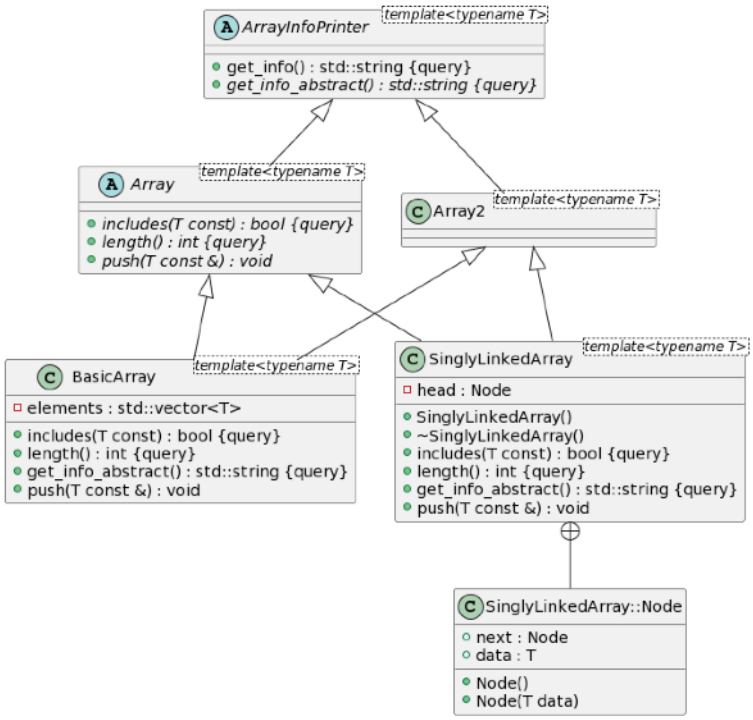


*Рис. UML-діаграма класів*

**Висновок:** Виконуючи лабораторну роботу №9 я навчився створювати списки об’єктів базового типу, що включають об’єкти похідних типів. Освоїв способи вирішення проблеми неоднозначності при множинному наслідуванні. Вивчив плюси заміщення функцій при множинному наслідуванні. Навчився використовувати чисті віртуальні функції, знати коли варто використовувати абстрактні класи.