МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ
Озерский технологический институт — филиал НИЯУ МИФИ

Кафедра прикладной математики

Вл. Пономарев

ПРАКТИКУМ

по объектно-ориентированному программированию

Учебно-методическое пособие

Часть 2. Наследование и полиморфизм

Озерск, 2019 г.

УДК 681.3.06
П 56

Вл. Пономарев. Практикум по объектно-ориентированному программированию. Учебно-методическое пособие. Часть 2. Наследование и полиморфизм. Озерск: ОТИ НИЯУ МИФИ, 2019. — 44 с.

В пособии излагается, как выполнять практические работы по дисциплине «Объектно-ориентированное программирование».

В качестве основного материала при выполнении практических работ пособие предназначено для студентов, обучающихся по направлению подготовки 09.03.01 «Информатика и вычислительная техника» и специальности 09.05.01 «Применение и эксплуатация автоматизированных систем специального назначения».

Рецензенты:
1.
2.

УТВЕРЖДЕНО
Редакционно-издательским
Советом ОТИ НИЯУ МИФИ

ОТИ НИЯУ МИФИ, 2019

Вл. Пономарев © 2019

Содержание

Общие цели занятий ... 4

1. Работа ООП-201. Основы наследования ... 5
   - 1.1. Подготовка проекта ... 5
   - 1.2. Наследование ... 5
   - 1.3. Наследство ... 7
   - 1.4. Отношение обобщения ... 9
   - 1.5. Гирлянда ... 10
   - 1.6. Множественное наследование ... 11
   - 1.7. Тип объекта указателя ... 12
   - 1.8. Вопросы и упражнения ... 13
2. Работа ООП-202 (4 часа). Полиморфизм ... 14
   - 2.1. Подготовка шаблона ... 14
   - 2.2. Приложение Paint ... 14
   - 2.3. Структуры данных ... 15
   - 2.4. Добавление объекта ... 17
   - 2.5. Вывод объектов ... 17
   - 2.6. Раннее связывание ... 19
   - 2.7. Определение типа объекта указателя ... 19
   - 2.8. Виртуальные методы и полиморфизм ... 21
   - 2.9. Механизм позднего связывания ... 22
   - 2.10. Не перегруженные виртуальные методы ... 24
   - 2.11. Не виртуальные методы ... 25
   - 2.12. Виртуальные методы и раннее связывание ... 26
   - 2.13. Механизм RTTI ... 26
   - 2.14. Динамическое приведение типов ... 28
   - 2.15. Вопросы и упражнения ... 30
3. Работа ООП-203 (4 часа). Абстрактные классы ... 31
   - 3.1. Подготовка проекта ... 31
   - 3.2. Абстрактный класс ... 31
   - 3.3. Чистая виртуальная функция ... 31
   - 3.4. Конструктор производного класса ... 33
   - 3.5. Методы базового класса ... 34
   - 3.6. Приложение ... 35
   - 3.7. Интерфейсы и отношение реализации ... 37
   - 3.8. Интерфейс для фигур ... 38
   - 3.9. Сила абстракций ... 40
   - 3.10. Вопросы и упражнения ... 44

Общие цели занятий

В ходе практических работ изучаются основы использования классов в рамках методологии объектно-ориентированного программирования.

В этой части работ рассматриваются следующие темы:

1) простое наследование реализации;
2) отношение обобщения.
3) множественное наследование;
4) раннее и позднее связывание;
5) виртуальные функции и полиморфизм;
6) определение типа объекта во время выполнения;
7) чистая виртуальная функция;
8) абстрактный класс;
9) интерфейс;
10) отношение реализации.

К практическим работам приписаны контрольные вопросы и упражнения. Контрольные вопросы могут быть заданы преподавателем в ходе защиты работы, однако преподаватель может задавать и другие вопросы, не указанные в списке.

Для защиты знаний и навыков, полученных в ходе выполнения практических работ студент должен иметь тетрадь (12-18 листов). Для каждой выполненной работы в тетрадь записывается отчет.

Отчет по работе начинается с заголовка:

1. Фамилия Имя Отчество
2. Группа
3. Дата начала выполнения работы
4. Код работы
5. Название работы
6. Цели работы
7. Задачи работы

При необходимости при выполнении работы в отчет записываются контрольные значения.

Во время защиты тетрадь используется для вопросов преподавателя и ответов студента.

1. Работа ООП-201. Основы наследования

Цели:
- изучение механизма наследования реализации.

Задачи:
- исследование наследования;
- исследование наследства;
- исследование отношения обобщения;
- исследование множественного наследования;
- определение типа объекта во время выполнения.

1.1. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска С:. Переименуйте каталог в OOP201. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию.

Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// OOP-201. Основы наследования
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

1.2. Наследование

Модуль class.h.

По ходу наших работ требуется цветная лампочка.

У нас же есть лампочка просто.

Она вот такая:

```
// просто лампочка
class lamp {
    int state;
public:
    lamp() : state(1) {}
};
```

Для цветной лампочки мы описываем новый класс:

```
// цветная лампочка
class colorlamp {
    int state, color;
public:
    colorlamp() : state(1), color(2) {}
};
```

В функции main описываем представителя первого и второго класса:

```
int main(void) {
    lamp a;
    colorlamp b;
    int x = 0; // точка остановки
}
```

Выполняем эти две строчки кода и останавливаемся.

В окно Watch наблюдаем переменные a и b:

Watch 1

Name	Value	Type
⊟ a	{state=1}	lamp
state	1	int
⊟ b	{state=1 color=2}	colorlamp
state	1	int
color	2	int

Как видим, два объекта похожи в том, что они одинаково начинаются: сначала элемент state.

Остановим выполнение проекта Shift+F5.

Цветная лампочка *это* лампочка. Или:

Цветная лампочка *is a* лампочка.

Очевидно, что между цветной и простой лампочками есть отношение *обобщения*. А отношению обобщения в ООП соответствует *наследование*.

Поэтому описываем новый класс цветной лампочки, *производный* от класса простой лампочки (модуль class.h). При этом класс лампочки lamp становится *базовым* классом класса cola:

```
// цветная лампочка -
// это специальная лампочка
class cola : public lamp {
    int color;
public:
    cola() : color(2) {}
};
```

В функции main описываем экземпляр специальной лампочки:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
}
```

Выполняем эти строчки кода и останавливаемся.

В окне Watch наблюдаем переменные a и c:

Watch 1

Name	Value	Type
⊟ a	{state=1}	lamp
state	1	int
⊟ c	{color=2}	cola
⊞ lamp	{state=1}	lamp
color	2	int

Видим, что цветная лампочка cola состоит из просто лампочки lamp и элемента color.

Остановим выполнение проекта Shift+F5.

В чем различие между colorlamp и cola?

В том, что изменение класса lamp не влияет на класс colorlamp, но влияет на класс cola. Это следствие установления отношения обобщения.

С точки же зрения физического устройства объектов типа cola и типа colorlamp разницы нет. Оба объекта состоят из элемента state сначала, и элемента col затем.

В этом можно убедиться.

Удалите комментарий строки "using namespace std;" в модуле class.h, затем добавьте следующий *жульнический* код в функции main:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
    cout << sizeof(b) << endl;
    cout << *((int*)&b + 0) << endl;
    cout << *((int*)&b + 1) << endl;
    cout << sizeof(c) << endl;
    cout << *((int*)&c + 0) << endl;
    cout << *((int*)&c + 1) << endl;
}
```

Для каждого из объектов здесь сначала выводится его размер, затем значение первого (нулевого по смещению) элемента и значение второго элемента. Убеждаемся, что оба объекта показывают одни и те же значения.

Должен возникнуть вопрос «А что лучше?». Или вопрос «А как правильно?». Есть однозначный ответ: если вам нужно *отношение обобщения*, следует использовать наследование. Нужно отношение обобщения или нет, — это вопрос, который мы рассмотрим позже.

Если отношение обобщения не нужно, ответ неоднозначный. Иногда лучше colorlamp, иногда лучше cola. Например, вариант cola может замедлить выполнение, если создается много объектов. Вариант colorlamp также лучше, если класс lamp как таковой не используется (не требуется).

1.3. Наследство

Далее следует уточнить, что входит в наследство при наследовании одного класса другим. Описываем метод set класса cola. Модуль class.h:

```
class cola : public lamp {
    int color;
public:
    cola() : color(2) {}
    // включает/выключает
    void set(int value) {
        state = value;
    }
};
```

Увы, компилятор не разрешает использовать элемент state, доставшийся в наследство от класса lamp. Это должно быть понятно: закрытый элемент state по определению виден только самому классу и его друзьям.

Для случая наследования существует модификатор доступа protected, который по-прежнему закрывает элемент для посторонних, но открывает его для производных классов, создавая защищенную секцию.

Мы должны переместить элемент state в эту секцию:

```
class lamp {
protected:
    int state;
public:
    lamp() : state(1) {}
};
```

Как только мы это сделаем, класс cola станет правильным.

Добавляем в класс cola метод get для определения состояния:

```
class cola : public lamp {
    int color;
public:
    . . .
    // состояние
    int get() {
        return state;
    }
};
```

В основной функции убеждаемся, что лампочка с работает:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
    cout << "c.get =" << c.get() << endl;
    c.set(0);
    cout << "c.get =" << c.get() << endl;
}
```

Правильно ли мы сделали?

С точки зрения цветной лампочки все логично. Ее нужно включать и выключать. А вот просто лампочку как включить или выключить?

Поэтому копируем методы класса cola в класс lamp и проверяем:

```
int main(void) {
    lamp a;
    cout << "a.get =" << a.get() << endl;
    a.set(0);
    cout << "a.get =" << a.get() << endl;
    colorlamp b;
    cola c;
    cout << "c.get =" << c.get() << endl;
    c.set(0);
    cout << "c.get =" << c.get() << endl;
}
```

Все замечательно должно работать.

Однако вспомним, что в наследство производному классу достается все, что есть в базовом классе, вопрос только в доступе к элементам.

Являются ли доступными в производном классе методы базового?

Попробуем другой вариант класса cola:

```
class cola : public lamp {
    int color;
public:
    cola() : color(2) {}
    // включает/выключает
    void set(int value) {
        lamp::set(value);
    }
    // состояние
    int get() {
        return lamp::get();
    }
};
```

Этот код показывает, как можно вызвать метод базового класса.

Удостоверяемся, что все по-прежнему работает.

Но на самом деле в классе cola эти методы не нужны. Поскольку они достаются в наследство и находятся в открытой секции, то они будут вызываться с объектом типа cola, хотя принадлежат классу lamp.

Мы не удаляем эти методы, а помещаем их в комментарий.

Снова удостоверяемся, что все по-прежнему работает.

На самом деле правильно так, как получилось в последнем варианте.

По сути, цветная лампочка отличается от просто лампочки лишь тем, что обладает цветом. Поэтому, если используется наследование, то в производный класс нужно добавлять лишь то, что отличает его от базового класса. Это экономит код, делает его проще, и, самое главное, — избавляет код от дублирования.

1.4. Отношение обобщения

К чему нам отношение обобщения?

Да вот, например.

Мы хотим сделать гирлянду из разных лампочек.

Как это сделать?

Наверное, нужен массив.

Массив какого типа?

Если взять тип cola или тип colorlamp, то только заявленный тип в массив и можно поместить.

Если же взять тип lamp, то в массив можно поместить как простые лампочки, так и цветные лампочки типа cola. Просто потому что тип cola связан с типом lamp отношением обобщения, и приводится к нему.

Попробуем сначала создать массив.

Функция main:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
    lamp g[] = {c, b, a};
}
```

Действительно, эти типы не соединяются в один массив, компилятор сопротивляется.

Удалим в массиве переменную b.

Теперь должно быть корректно.

Иначе говоря, объекты типа cola трактуются как объекты типа lamp.

Но так, как мы сейчас сделали, делать *нельзя*.

На самом деле объекты типа cola и lamp различны. Цветные лампочки имеют размер 8 байт, а простые 4 байта. Поскольку массив имеет тип lamp, объекты в нем размещаются каждые 4 байта, и объект a на самом деле записался поверх объекта c (переменная state объекта a записалась поверх переменной col объекта c), в чем можно убедиться.

Нужно запомнить раз и навсегда: *если есть отношение обобщения*, лучше не использовать статические объекты. Нужны либо ссылки, либо динамические объекты. Отношение обобщения наиболее полно проявляется при использовании только *ссылок* или *указателей*.

Поэтому правильно будет так:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
    lamp * g[] = {&c, &a};
}
```

Можно убедиться, что объекты массива выполняют свои функции.

Для этого дописываем в конце функции main следующий код:

```
int main(void) {
    lamp a;
    colorlamp b;
    cola c;
    lamp * g[] = {&c, &a};
    g[0]->set(5);
    g[1]->set(6);
    cout << "g[0]->get=" << g[0]->get() << endl;
    cout << "g[1]->get=" << g[1]->get() << endl;
}
```

Все работает, поэтому можно приступить к созданию гирлянды.

1.5. Гирлянда

Модуль class.h.

Поскольку гирлянда будет представлена массивом, нужно объявить константу MAX_GARLAND со значением, равным, например, четырем:

```
#define MAX_GARLAND 4
```

Описываем класс garland.

Закрытые элементы класса — это массив g типа lamp* и размерностью MAX_GARLAND, и счетчик лампочек count.

Конструктор и методы размещаем в открытой секции.

Конструктор по умолчанию устанавливает нулевое значение count.

1. Метод void add(lamp * a) добавляет в массив новую лампочку.

Сначала нужно проверить значение счетчика:

```
assert(count < MAX_GARLAND);
```

Затем в элемент массива g[count] записать переданный объект a.

2. Метод должен увеличивать значение счетчика *после* добавления a.

Метод void set(int value) устанавливает значения всех лампочек.

Для этого он формирует цикл по переменной i, которая изменяется от нуля *до* значения счетчика, и в итерации использует метод set элемента массива.

3. Метод void show выводит гирлянду в консоль.

Формирует такой же цикл, как и метод set, но в итерации выводит в консоль значение элемента массива при помощи метода get элемента.

Конец строки endl в цикле не выводим. Выводим endl после цикла.

После конструирования класса в main пробуем зажечь гирлянду:

```
int main(void) {
    garland g;
    g.add(new lamp);
    g.add(new lamp);
    g.add(new cola);
    g.add(new cola);
    g.set(0);
    g.show();
    g.set(1);
    g.show();
}
```

Если все правильно, в консоль сначала выводится строка из четырех нулей, а затем строка из четырех единиц.

1.6. Множественное наследование

Модуль class.h.

Опишем новый класс color, состоящий из элемента данных col, конструктора по умолчанию, методов int get(void) и void set(int).

Опишем еще один класс цветной лампочки с названием micola, наследующий классы lamp и color, тело класса пустое.

В основной функции создадим экземпляр micola m.

Исследуем экземпляр m, чтобы понять, как он устроен.

Пробуем использовать метод m.set(7);.

Компилятор говорит, что возникла *неоднозначность*, — в классе две одинаково называющиеся функции. Действительно, одна функция set наследуется от класса lamp, а вторая, — от класса color.

Разрешается эта неоднозначность использованием операции разрешения видимости: m.color::set(7);. Удостоверьтесь в том, что вызывается правильный метод, — метод класса color.

Измените названия методов класса color на getc и setc.

Разрешение неоднозначности больше не требуется.

Попробуйте изменить в классе color название элемента col на state.

Удостоверьтесь, что неоднозначность в классе micola из-за этого не возникает.

Попробуйте поменять в описании класса micola порядок базовых классов. Исследуйте экземпляр класса micola, сравните его с тем, что было до изменения порядка.

1.7. Тип объекта указателя

Осталось разобраться с тем, что есть в цветной лампочке такого, чего нет в простой лампочке, а именно, — с ее цветом.

Для управления цветом лампочки нужны два метода: getc, возвращающий цвет, и setc, устанавливающий цвет.

Для этого просто копируем методы get и set класса lamp в класс cola, и меняем некоторые идентификаторы.

Сами по себе эти методы наверняка работают, когда их вызывают с объектом типа cola или с объектом типа cola*. Нас же больше интересует, можно ли их вызывать с объектом типа lamp*.

Для этой цели можно попробовать приведение типа, например, так:

```
int main(void) {
    lamp * x = new cola;
    cola * y = (cola*)x;
    y->setc(3);
}
```

Можно убедиться, что данный код работает, отслеживая изменение объекта y в окне Watch.

Однако не все так просто. Например, нам бы хотелось, чтобы при выводе гирлянды выводилось 1, если горит лампочка типа lamp, и значение цвета, если горит лампочка типа cola. Это значит, что при выводе значения некоторой лампочки, которая представлена в гирлянде типом lamp*, сначала нужно узнать, какого она типа. Если она lamp, выводим значение get, а если она типа cola, выводим значение getc.

Эта задача не простая.

Чтобы в этом убедиться, стоит исследовать следующий код:

```
int main(void) {
    lamp * x = new lamp;
    cola * y = (cola*)x;
    int z = y->getc();
}
```

Здесь объект типа lamp* приводится к типу cola*. У объекта типа cola нет метода getc, но вызвать его можно, поскольку переменная y имеет тип cola*. Естественно, в ответ мы получим что-то неопределенное.

Иначе говоря, нет никакой возможности определить, какой тип имеет переменная y на самом деле, чтобы решить задачу вывода цвета лампочки.

Вам предлагается самим подумать над ее решением, поэкспериментировать, поискать решение в сети Интернет.

Решение точно существует и оно простое.

1.8. Вопросы и упражнения

1. Как синтаксически описывается наследование?

2. Как называются классы, участвующие в наследовании?

3. Какое отношение между классами возникает при наследовании реализации?

4. Что наследуется? Какой доступ в наследующем классе имеют элементы наследуемого класса?

5. Что произойдет, если доступ элемента state базового класса lamp поменять на закрытый? Как решить проблему, которая возникнет при этом, если она возникнет?

6. Что произойдет, если доступ базового класса lamp поменять на защищенный? Как решить проблему, которая возникнет при этом, если она возникнет?

7. Как синтаксически вызывается метод наследуемого класса в методе наследующего класса?

8. Как конструируется объект производного класса?

9. Когда наследование необходимо для конструирования класса цветной лампочки, а когда в этом особой необходимости не возникает?

10. Что означает множественное наследование? Какие неоднозначности при этом могут возникнуть и как они могут быть разрешены.

11. Сконструируйте классы и отношения, описывающие следующие объекты: транспортное средство как таковое vehicle, велосипед bicycle, автомобиль как таковой car, автомобиль mersedes, автомобиль lada. Для каждого базового класса найдите хотя бы один общий для производных классов элемент данных. Нарисуйте диаграмму классов UML.

12. Сконструируйте оригинальный класс «летающая лампочка» flyla, используя множественное наследование от классов lamp и point.

2. Работа ООП-202 (4 часа). Полиморфизм

Цели:
- изучение механизма позднего связывания и полиморфизма.

Задачи:
- исследование раннего связывания;
- исследование позднего связывания;
- исследование таблицы виртуальных функций;
- исследование механизма RTTI;

2.1. Подготовка шаблона

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP202. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию.

Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// OOP-202. Полиморфизм
// 01.01.2000
```

Модуль class.h предназначен для описания классов.

2.2. Приложение Paint

Будем проектировать приложение Paint, — графический редактор.

Приложение Paint — это класс, название класса paintapp.

Для начала будет достаточно, если приложение сможет рисовать кружочки (*circle*) и квадратики (*square*).

Кружочки и квадратики — это классы circle и square соответственно.

Для того, чтобы разместить объекты в разных местах рисунка, нужно добавить к ним точку отсчета объекта, — левый верхний угол.

Для указания точек потребуется класс точки с двумя координатами.

Уместно назвать этот класс point.

Рисовать будем так. Кружочек будет выводить в консоль "c(x,y)", а квадратик, соответственно, выводить "s(x,y)", где x и y — какие-то числа.

Приложение может добавлять новые объекты и рисовать их.

Метод paintapp::add добавляет объект, а метод paintapp::draw выводит в консоль рисунок целиком, от первого до последнего объекта.

Объекты рисуют себя сами, для этого у них есть метод class::draw.

В соответствии с этим можно начать описание классов с такой схемы (модуль class.h):

```
// точка
struct point {
    int x, y;
    // конструктор по умолчанию
    point() : x(0), y(0) {}
};

// кружочек
class circle : public figure {
protected:
    point tl;
public:
    // конструктор по умолчанию
    circle() {}
    // рисует объект
    void draw(void) {
        printf("c(%d,%d)\n", tl.x, tl.y);
    }
};

// квадратик
class square : public figure {
protected:
    point tl;
public:
    // конструктор по умолчанию
    square() {}
    // рисует объект
    void draw(void) {
        printf("c(%d,%d)\n", tl.x, tl.y);
    }
};

// приложение
class paintapp {
public:
    // конструктор по умолчанию
    paintapp() : count(0) {}
    // добавляет фигуру
    void add() {
    }
    // рисует объекты
    void draw(void) {
    }
};
```

2.3. Структуры данных

Дальше следует придумать, как мы будем сохранять информацию об объектах рисунка. Есть множество вариантов.

Например:

Вариант 1. Для хранения кружочков один массив, для хранения квадратиков другой массив.

Вариант 2. Для хранения всех объектов один массив.

Критика варианта 1: если потом потребуется хранить объекты других типов, потребуется добавить неопределенное количество массивов и переделывать половину класса paintapp.

Критика варианта 2: как мы выяснили в предыдущей работе, возникают проблемы с определением типа объекта.

Все-таки вариант 2 кажется более предпочтительным по следующим соображениям:

- для вывода всех объектов, очевидно, потребуется цикл;
- если использовать несколько массивов, потребуется несколько циклов, а при добавлении нового типа объекта нужно будет дописывать в процедуре вывода новые циклы. Это кажется неразумным.

Остается проблема определения типа объекта.

Эту проблему мы как-нибудь решим, это точно.

Остается определиться с типом массива.

Если использовать тип circle*, то нельзя будет добавить объект square и наоборот. Из этого следует, что нужен какой-то более общий тип.

Рассматривая объекты более абстрактно, можно предположить, что все объекты, которые можно нарисовать, являются какими-то фигурами.

Можно предложить ввести такой более общий тип «фигура», который позволит обращаться к производным от него типам как к однотипным.

Естественно, то, что предлагается, — это *отношение обобщения*.

Буквально:

- кружочек *это* фигура,
- квадратик *это* фигура, и т.д.

Следовательно, нам нужен более абстрактный класс, — класс figure.

Рассматривая абстракцию геометрической фигуры, можно понять, что ее местоположение присуще всем фигурам вообще.

Это приводит нас к пониманию того, что фигура как тип обладает как минимум положением, то есть точкой верхнего левого угла (top-left).

Могут быть и другие общие характеристики фигур, например, цвет, площадь, периметр. Однако сейчас нам эти характеристики не важны.

Учитывая вышеизложенное, добавляем в проект класс figure.

Единственный элемент данных фигуры, — точка tl. Поскольку эта точка должна быть доступна в производных классах, помещаем ее в защищенную секцию.

Пока в этом классе больше ничего нет.

Поскольку возникло отношение обобщения, классы кружочка и квадратика *должны быть производными* от класса фигуры. Соответственно, изменяем эти классы, убеждаемся, что модификация не разрушает проект.

Поскольку мы определились с более абстрактным классом, его можно использовать как тип для массива фигур. Естественно, тип может быть только указательным, то есть figure*.

Теперь в классе приложения pointapp можно объявить элемент данных в виде массива g (от *graphics*) типа figure*, с размерностью MAX_OBJECT.

Для подсчета количества объектов нужен элемент данных count.

Этот элемент должен обнуляться конструктором.

Доступ к этому элементу видимо, закрыт.

Константа MAX_OBJECT должна быть объявлена в модуле class.h, например, в начале модуля, ее значение может быть равным четырем, для отладки больше не требуется.

2.4. Добавление объекта

Теперь мы можем реализовать метод для добавления нового объекта в рисунок, то есть метод paintapp::add.

Добавляемый объект является параметром метода add, тип добавляемого объекта должен быть figure*, чтобы можно было добавлять разные по типу объекты. Наименование переменной параметра может быть a.

Сначала нужно удостовериться, что значение элемента count меньше MAX_OBJECT при помощи функции assert. Это обязательно, иначе возможно нарушение памяти.

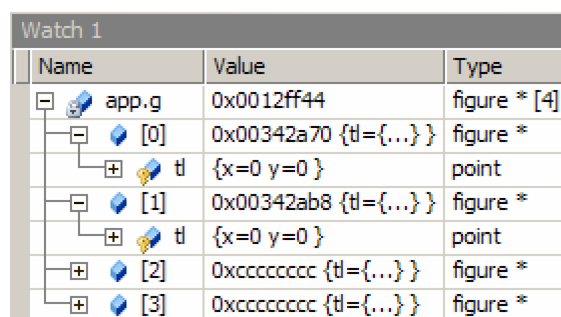
Затем в элемент массива g[count] записываем объект a.

Метод add должен увеличивать счетчик объектов count *после* добавления объекта.

Испытание этой части проекта выглядит следующим образом:

```
int main(void) {
    paintapp app;
    app.add(new circle());
    app.add(new square());
}
```

Следует проверить, что содержит массив app.g.



Watch 1

Name	Value	Type
app.g	0x0012ff44	figure * [4]
[0]	0x00342a70 {tl={...} }	figure *
tl	{x=0 y=0 }	point
[1]	0x00342ab8 {tl={...} }	figure *
tl	{x=0 y=0 }	point
[2]	0xcccccccc {tl={...} }	figure *
[3]	0xcccccccc {tl={...} }	figure *

К сожалению, типы объектов здесь не показываются.

Если все работает, продолжаем.

2.5. Вывод объектов

Наша основная задача заключается в рисовании всех объектов.

Как говорилось, объекты рисуют себя сами.

Для этого у них есть метод void draw.

Сейчас самое время определить этот метод в классах circle и square.

Возьмем класс circle.

Метод circle::draw(voi d) может иметь следующий вид:

```
void draw(void) {
    printf("c(%d,%d)\n", tl.x, tl.y);
}
```

Следует проверить, как объект рисует себя, функция main:

```
int main(void) {
    circle c;
    c.draw();
    paintapp app;
    app.add(new circle());
    app.add(new square());
}
```

Убеждаемся, что в консоль выводится правильная строчка.

Копируем метод в класс square, изменяем название объекта в строке.

Удостоверяемся, что объект типа square рисует себя правильно.

Давайте запишем в отчет размер объекта типа cicrle.

Выведем результат функции sizeof(circle) в функции main:

```
cout << "sizeof(circle)=" << sizeof(circle) << endl;
```

Результат должен быть равен восьми (int x + int y).

Теперь нужно реализовать метод paintapp::draw.

Нам хочется, чтобы метод paintapp::draw рисовал все объекты в одном и том же цикле, то есть вот так:

```
void draw(void) {
    for (int i = 0; i < count; i++) {
        g[i]->draw();
    }
}
```

Описываем этот метод в классе paintapp, и убеждаемся, что метод draw недоступен в классе figure, который является базовым.

Поэтому добавляем в класс figure метод draw, аналогичный методам производных классов. Как только мы опишем метод figure::draw, метод paintapp::draw станет правильным, и мы можем посмотреть, что получилось. Основная функция, вызываем метод paintapp::draw:

```
int main(void) {
    paintapp app;
    app.add(new circle());
    app.add(new square());
    app.draw();
}
```

Увы, выводится два объекта типа figure:

```
f(0,0)
f(0,0)
```

2.6. Раннее связывание

Связывание, упрощено, — это определение адреса метода объекта.

Когда во время трансляции программы обнаруживается, что для объекта o типа A вызывается метод f, то есть транслируется запись o.f, то формируется машинный код, который:

- проталкивает в стек адрес объекта o машинной инструкцией push,
- вызывает метод A::f машинной инструкцией call адрес(A_f).

На языке Си это соответствует записи A_f(&o), при условии, что A_f — условное, декорированное имя функции метода A::f.

Такое связывание называется *ранним* (*early binding*), и это означает, что в момент трансляции *известно*, что объект имеет тип A, и, соответственно, вызывать нужно метод A::f.

В нашей программе объектом является элемент массива g[i], относительно которого можно только сказать, что этот элемент указывает на объект какого-то типа, но какого конкретно, неизвестно, и не будет известно до момента выполнения программы. Тем не менее, инструкция call адрес должна быть сформирована в любом случае, так как вызов метода есть.

Поскольку нет дополнительных указаний, как выполнить связывание, во время трансляции объект g[i] связывается с методом figure::draw, потому что объект g[i] имеет тип figure*.

Иначе говоря, при раннем связывании объект связывается с методом не на основании *типа объекта*, а на основании *типа указателя*.

Следовательно, возникает проблема.

Есть три метода, адреса которых известны. Есть объект g[i] типа указателя, но тип объекта, на который указывает g[i], неизвестен. Нужен механизм, который определит тип объекта указателя g[i].

2.7. Определение типа объекта указателя

Мы можем решить проблему определения типа объекта сами.

Сначала посмотрим, как можно заставить программу вызвать другой метод, нежели figure::draw.

Немного изменим код метода paintapp::draw:

```
void draw(void) {
    for (int i = 0; i < count; i++) {
        ((circle*)g[i])->draw();
    }
}
```

Здесь мы используем приведение указателя g[i] к типу circle*.

Запускаем программу и убеждаемся, что произошло раннее связывание объекта g[i] с другим методом, а именно с методом circle::draw.

В консоль выводится:

```
c(0,0)
c(0,0)
```

Чтобы использовать приведение, нужно знать тип объекта указателя.

Добавим в класс paintapp еще один массив:

```
class paintapp {
    figure * g[MAX_OBJECT];
    int t[MAX_OBJECT];
    int count;
public:
    . . .
};
```

Будем записывать в массив t тип объекта.

По-хорошему, для каждого типа нужно было бы задать константу перечисления, но мы для простоты примем, что значение 1 обозначает объект типа circle, а значение 2, — объект типа square.

Добавим также два метода для включения в рисунок разных объектов:

```
class paintapp {
    . . .
    // добавляет кружочек
    void add_c(circle * a) {
        assert(count < MAX_OBJECT);
        t[count] = 1;
        g[count++] = a;
    }
    // добавляет квадратик
    void add_s(square * a) {
        assert(count < MAX_OBJECT);
        t[count] = 2;
        g[count++] = a;
    }
    . . .
};
```

Чтобы не портить имеющийся метод paintapp::draw, добавим в класс paintapp еще один метод для рисования:

```
class paintapp {
    . . .
    // рисует, выясняя тип объекта
    void drawt(void) {
        for (int i = 0; i < count; i++) {
            if (t[i] == 1) {
                ((circle*)g[i])->draw();
            } else if (t[i] == 2) {
                ((square*)g[i])->draw();
            } else {
                ((figure*)g[i])->draw();
            }
        }
    }
    . . .
};
```

Здесь мы при помощи массива t определяем тип записанного в элементе g[i] объекта и выполняем соответствующее приведение.

Остается немного изменить код функции main:

```
int main(void) {
    paintapp app;
    app.add_c(new circle());
    app.add_s(new square());
    app.drawt();
}
```

Запускаем программу и убеждаемся, что мы смогли решить проблему, используя раннее связывание. В консоль выводится:

```
c(0,0)
s(0,0)
```

2.8. Виртуальные методы и полиморфизм

Трудно не согласиться с тем, что полученный код класса paintapp не особенно изящен. Мы решили проблему, что называется, «в лоб», не особенно задумываясь о компактности кода или его производительности.

В частности, очень смущают множественные операторы if метода paintapp::drawt. Практически код можно бесконечно улучшать, используя разные технические приемы, и в конечном итоге избавиться от множественного разбора (заменив его, например, операцией сложения, что значительно повысит производительность), но проблема кода не в этом.

В сущности цель программирования, — получение такого кода, который бы зависел от возможно меньшего числа внешних факторов, то есть был бы в большей степени *абстрактным*, нежели *конкретным*.

Образно говоря, мы спроектировали класс, сложность которого $O(n)$, в то время как сложность задачи является $O(1)$! (n — число разных фигур).

На самом деле методология ООП предполагает, что объекты типа указателя могут проявлять себя разным образом, вследствие того, что в реальном мире такое проявление имеет место быть. Например, живое существо как таковое умеет издавать звуки, при этом конкретное живое существо может мяукать, крякать или тявкать.

Это проявление многообразия в ООП называется *полиморфизмом*.

Заметим, что полиморфизм имеет отношение только к поведению объектов, и не относится к их характеристикам. Иначе говоря, полиморфными являются методы, классы и объекты, но не свойства.

В приложении к нашей программе, метод figure::draw фактически является полиморфным, потому что конкретные фигуры, являющиеся фигурами как таковыми, рисуются по-разному.

Для реализации полиморфизма в ООП есть механизм, позволяющий связать объект с методом не на основании *типа указателя*, а на основании *типа объекта*.

Этот механизм называется *поздним связыванием* (*late binding*), и нам его нужно всего лишь включить. Включается позднее связывание просто. Полиморфный метод базового класса должен быть помечен специальным модификатором virtaul, то есть, обозначен как *виртуальный* метод.

Заметим, что сам по себе модификатор не ведет к появлению полиморфизма, — он может быть, а может и не быть в результирующей программе. Полиморфное поведение возможно только при использовании указателя или ссылки, и только если они указывают на базовый класс. При соблюдении еще некоторых условий раскрывается настоящая мощь ООП.

Сейчас давайте уточним метод paintapp::draw:

```
void darw(void) {
    for (int i = 0; i < count; i++) {
        g[i]->draw();
    }
}
```

Кроме того, уточним код функции main:

```
int main(void) {
    paintapp app;
    app.add(new circle());
    app.add(new square());
    app.draw();
}
```

Описываем метод figure::draw как виртуальный:

```
class figure {
    . . .
    virtual void draw(void) {
        printf("f(%d,%d)\n", tl.x, tl.y);
    }
};
```

Запускаем программу и убеждаемся, что *механизм позднего связывания* включился и *полиморфизм* появился.

В консоль выводится:

```
c(0,0)
s(0,0)
```

2.9. Механизм позднего связывания

Образно, механизм позднего связывания, — это переключатель, который устанавливается в объект, и указывает путь к виртуальному методу в зависимости от типа объекта.

Этот переключатель называется *виртуальным указателем* и обозначается сокращением vptr, а путь к виртуальному методу находит *таблица виртуальных методов* (*virtual method table*, VMT), обозначаемая vtable.

Сейчас запишем в отчет размер объекта типа circle.

Для этого снова используем sizeof, как было показано ранее.

Как бы это ни казалось странным, но как только мы включили механизм позднего связывания, размер объекта типа circle изменился. Ранее мы записали, что размер был равен 8, теперь же он стал равен 12.

Объект изменился, потому что в него добавился новый элемент данных, тот самый переключатель vptr.

Поскольку это указатель, то есть адрес, его размер в Win32 равен 4, 8 + 4 = 12. Если не 12, а 16, то СТОП! Целевая машина не x86.

Заметим, что vptr всегда размещается в начале объекта, и имеет смещение, равное нулю. Это упрощает вычисление пути, то есть адреса виртуального метода (здесь мы немного нивелируем ситуацию, потому что в реальности таких указателей у объекта может быть несколько).

Виртуальный указатель *указывает на таблицу* виртуальных методов.

Таблица виртуальных методов класса A, — это таблица *указателей*, то есть адресов *фактических* методов класса A. Поскольку каждый адрес занимает в памяти 4 байта в Win32, таблица на самом деле небольшая.

В момент создания объекта класса A виртуальный указатель vptr устанавливается на таблицу виртуальных методов vtable класса A, обеспечивая связь объекта с методом в зависимости от *типа объекта*. Если тот же объект заменить другим полиморфным объектом, то его переключатель vptr будет установлен на таблицу vtable другого класса.

Каким образом вычисляется адрес виртуального метода. Машинная инструкция call требует адрес, и его нужно иметь. Поскольку в момент трансляции тип объекта неизвестен, нужен способ вычисления адреса вне зависимости от того, каков будет объект.

Все просто. Адрес вычисляется не в момент трансляции, а в момент *вызова* метода, который происходит во время выполнения. В момент выполнения тип объекта известен, поскольку его создали, и тогда vptr указывает на правильную vtable.

Формула вычисления, которая записывается в программу во время ее трансляции, *примерно* такова: взять адрес объекта, по адресу объекта находится vptr, взять адрес vtable (vptr — это адрес vtable) и прибавить к нему номер виртуального метода (в единицах адреса). Тогда по полученному адресу находится адрес фактического метода. В момент трансляции все составляющие формулы известны, собственно, это только адрес объекта и смещение (номер) метода, а изменение указателя vptr по ходу работы программы в разные моменты ее выполнения приводит к разным методам.

Все это не так сложно, как кажется.

Определим перед функцией main функцию showaddr, которая вычислит все адреса и выведет их в консоль:

```
// выводит адреса
void showaddr(figure * o) {
    // адрес объекта o
    printf("obj  = 0x%p\n", o);
    // vptr - элемент 0
    int vptr = ((int*)o)[0];
    printf("vptr = 0x%p\n", vptr);
    // адрес метода 0
    int meth = ((int*)vptr)[0];
    printf("draw = 0x%p\n", meth);
}
```

В основной функции будем изменять объект f:

```
int main(void) {
    figure * f = new figure;
    showaddr(f);
    f = new circle;
    showaddr(f);
    f = new square;
    showaddr(f);
}
```

Выполняем программу строчка за строчкой, сравниваем выводимые значения с тем, что показывает среда.

Для этого в окне Watch введем для наблюдения переменную f:

Watch 1

Name	Value	Type
f	0x00346758 {tl={...} }	figure *
__vfptr	0x00417838 const figure::`vftable'	*
[0]	0x004110e6 figure::draw(void)	*
tl	{x=0 y=0 }	point

Для виртуального указателя в моей среде разработки (Microsoft Visual Studio 2008) используется переменная с именем __vfptr, а таблица виртуальных методов обозначается `vftable'.

Убеждаемся, что вычисляемые адреса совпадают с теми, что показывает окно наблюдения Watch.

2.10. Не перегруженные виртуальные методы

Перегруженным (*overloaded*) называется метод производного класса, который заменяет соответствующий виртуальный метод базового класса.

Методы circle::draw и square::draw являются перегруженными. То есть они проявляют полиморфизм метода figure::draw.

Однако мы *не обязаны* перегружать виртуальный метод.

Что происходит в этом случае?

Давайте добавим в базовый класс figure еще один виртуальный метод, который, например, выводит в консоль координаты:

```
class figure {
    . . .
    // выводит координаты
    virtual void location(void) {
        printf("(%d,%d)", tl.x, tl.y);
    }
};
```

В основной функции создадим объект какого-нибудь производного класса, например, класса circle, используя переменную типа figure*:

```
int main(void) {
    figure * f = new circle;
}
```

Исполним строчку кода функции main и исследуем переменную f в окне наблюдения Watch:

Watch 1

Name	Value	Type
f	0x00346758	figure *
[circle]	{...}	circle
__vfptr	0x0041786c const circle::`vftable'	*
[0]	0x0041100a circle::draw(void)	*
[1]	0x00411307 figure::location(void)	*
tl	{x=0 y=0 }	point

Видим, что в таблице виртуальных методов класса circle есть указатель на второй метод (со смещением 1 в единицах адреса), который на самом деле указывает на метод figure::location.

Следовательно, если метод не перегружается (иначе говорят — не замещается), вызывается метод базового класса.

Этого и следовало ожидать.

2.11. Не виртуальные методы

Наконец, что происходит, если в базовом или производном классе есть обычные, не виртуальные методы?

Сначала добавим обычный метод в класс figure:

```
class figure {
    . . .
    // координата X
    int getX(void) {
        return tl.x;
    }
};
```

Опять выполняем строчку кода и исследуем объект f в окне *Watch*.

Видим, что никаких изменений не произошло.

Теперь добавим аналогичный обычный метод в класс circle:

```
class circle : public figure {
    . . .
    // координата Y
    int getY(void) {
        return tl.y;
    }
};
```

Снова выполняем строчку кода и исследуем объект f в окне Watch.

Видим, что никаких изменений также не наблюдается.

Иначе говоря, таблица виртуальных методов содержит адреса только виртуальных методов, а не всех методов класса.

Для не виртуальных методов может быть использовано только раннее связывание.

Выше говорилось: «Такое связывание называется *ранним* (*early binding*), и это означает, что в момент трансляции *известно*, что объект имеет тип A, и, соответственно, вызывать нужно метод A::f».

Действительно, если мы попытаемся сейчас в функции main вызвать не виртуальный метод circle::getY, компилятор скажет нам, что в базовом классе figure нет метода getY. Потому что в момент компиляции тип объекта, на который указывает f, неизвестен. Вы можете сказать, — как неизвестен, строчкой же выше мы создали объект типа circle, и компилятор успешно это транслировал! Да, это так, но только компилятор не занимается таким глубоким семантическим анализом, чтобы определить, что мы только что создали объект. Например, мы могли создать этот объект в какой-нибудь функции, которая вызывается через несколько других.

Сказанное не означает, что мы не можем вызвать метод circle::getY если есть указатель типа figure*. Практически всегда можно использовать приведение одного типа (figure*) к другому типу (circle*). Но только после этого связывание будет ранним по определению.

2.12. Виртуальные методы и раннее связывание

Механизм позднего связывания включается *только* тогда, когда тип объекта неизвестен. Это подразумевает, что используется указатель (неважно какого типа). Если тип объекта известен во время трансляции, то даже если метод виртуальный, будет применено раннее связывание.

В качестве примера рассмотрим следующий код функции main:

```
int main(void) {
    circle x = *(new circle);
    // раннее связывание
    x.draw();
    circle * y = new circle;
    // позднее связывание
    y->draw();
}
```

Определить тип связывания в этом случае можно, только если анализировать генерируемый ассемблерный код. Для этого нужно включить генерацию ассемблерного кода, выбрав в настройках проекта пункты C/C++, Output Files, Assembler Output — Assembly With Source Code. Код находится в каталоге C:\OOP202\oopp\Debug.

2.13. Механизм RTTI

RTTI (*Run-Time Type Info*) расшифровывается как «информация о типах во время выполнения». Это механизм, позволяющий установить фактический тип объекта во время выполнения программы. Сделать это возможно, *только если есть виртуальные функции*. Собственно, и задача определения типа объекта чаще важна в случае, если один и тот же указатель может указывать на разные объекты.

Тип объекта в этом случае можно определить, сравнивая виртуальный указатель vptr с адресами таблиц виртуальных методов. На самом деле информация о типах может включаться в таблицы виртуальных функций. При этом, возможно, требуется включение механизма RTTI в настройках проекта среды разработки. У меня механизм включен по умолчанию.

С механизмом RTTI связаны функция определения типа объекта и динамическое приведение типа объекта к другому типу.

Сначала попробуем определить фактический тип объекта при помощи функции typeid.

Функция typeid возвращает тип const type_info&. Этот тип имеет метод name, возвращающий тип объекта в виде строки.

Функция main:

```
int main(void) {
    circle x = *(new circle);
    // раннее связывание
    x.draw();
    printf("%s\n", typeid(x).name());
    circle * y = new circle;
    // позднее связывание
    y->draw();
    printf("%s\n", typeid(y).name());
    int z = 0; // для точки остановки
}
```

Запустив этот вариант программы, мы увидим, что переменная x имеет тип circle, а переменная y — тип circle*. В консоль выводится:

```
c(0,0)
class circle
c(0,0)
class circle *
```

Интереснее случай, когда переменная имеет тип, указывающий на базовый:

```
int main(void) {
    figure x = *(new circle);
    // раннее связывание
    x.draw();
    printf("%s\n", typeid(x).name());
    figure * y = new circle;
    // позднее связывание
    y->draw();
    printf("%s\n", typeid(y).name());
    int z = 0; // для точки остановки
}
```

Здесь мы изменили типы x и y, заменив circle на figure.

Заметим, что в консоль выводится:

```
f(0,0)
class figure
c(0,0)
class figure *
```

Объект x не является объектом circle (выводит f). Раннее связывание осталось ранним связыванием, и позднее связывание осталось поздним.

Операторы typeid выводят теперь соответственно figure и figure*.

Как же быть со вторым типом, объект ведь имеет тип circle.

Надо просто разыменовать указатель y:

```
printf("%s\n", typeid(*y).name());
```

Теперь вывод программы в консоль следующий:

```
f(0,0)
class figure
c(0,0)
class circle
```

То есть переменная y указывает на объект типа circle, являясь при этом переменной типа figure*. Что и требовалось показать.

2.14. Динамическое приведение типов

Динамическое приведение типа выполняет специальная синтаксическая конструкция вида dynamic_cast<тип>(объект).

В отличие от обычного приведения, результат динамического приведения может оказаться равным нулю (NULL), если тестируемый объект не может быть использован как указанный тип (например, объект типа circle не может быть приведен к типу square* и наоборот).

Говорят, что динамическое приведение позволяет выполнить понижающее приведение типа (*downcast*, от базового типа к производному типу), имея в виду положение типов на диаграмме UML (рисунок 1).

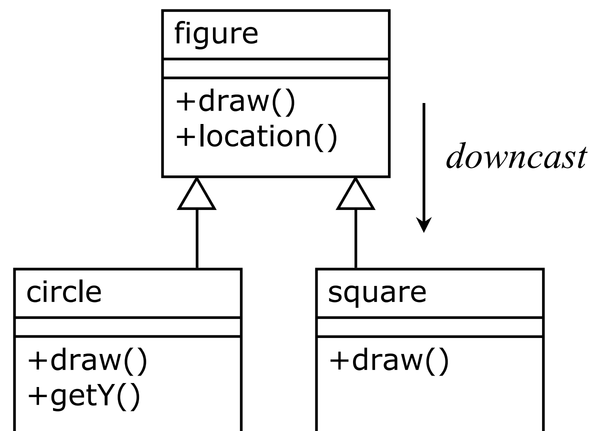


Рисунок 1 — Понижающее приведение

Заметим, что повышающее приведение, от производного типа к базовому типу, в большинстве случаев выполняет обычное приведение.

Добавим в класс paintapp еще один метод для рисования, который будет использовать динамическое приведение:

```
class paintapp {
    . . .
    // рисует, используя RTTI
    void drawrtti(void) {
        for (int i = 0; i < count; i++) {
            circle * a = dynamic_cast<circle*>(g[i]);
            if (a) {
                a->draw();
                continue;
            }
            square * b = dynamic_cast<square*>(g[i]);
            if (b) {
                b->draw();
                continue;
            }
        }
    }
};
```

Для тестирования нового метода используем следующий код:

```
int main(void) {
    paintapp app;
    app.add(new square());
    app.add(new circle());
    app.add(new circle());
    app.add(new square());
    app.drawrtti();
}
```

Запускаем программу и убеждаемся, что определение типа объекта во время выполнения действительно работает, в консоль выводится:

```
s(0,0)
c(0,0)
c(0,0)
s(0,0)
```

Полезно также выполнить код метода drawrtti шаг за шагом, чтобы уточнить, как работает динамическое приведение.

Мы ввели метод drawrtti исключительно для демонстрации, нет никакой необходимости использовать динамическое приведение в данном случае, поскольку полиморфизм и так обеспечивает правильный вывод.

Однако посмотрим на диаграмму UML (рисунок 1).

Классы circle и square не эквивалентны. В классе circle есть метод, которого нет в классе square. Динамическое приведение позволит в этом случае определить, можно или нельзя вызывать метод getY.

Добавим в класс paintapp еще один метод для рисования.

Для этого продублируйте метод draw и переименуйте в drawy.

В новом методе после вывода объекта методом draw попробуйте динамически привести объект к типу circle*, и если приведение успешно, то дополнительно для этого объекта выведите в консоль строчку со значением, возвращаемым методом getY. В функции main замените вызов метода drawrtti на вызов метода drawy.

В заключение заметим, что в случае, когда нет виртуальных функций, а механизм RTTI требуется, нужно включить в класс или классы пустую виртуальную функцию, единственное назначение которой заключается в том, чтобы включить RTTI. Вспомните, что в самом начале нам не хватало именно способа определения типа объекта указателя.

2.15. Вопросы и упражнения

1. Что называется ранним связыванием?
2. Что называется поздним связыванием?
3. Что такое виртуальный указатель vptr?
4. Что такое таблица виртуальных методов vtable?
5. Как работает механизм позднего связывания?
6. В чем заключается полиморфизм?
7. Что может быть полиморфным?
8. Назовите условия, при которых полиморфизм возможен.
9. Назовите условия, при которых полиморфизм присутствует.
10. Назовите условия, при которых используется позднее связывание.
11. Какие методы включаются в таблицу виртуальных методов?
12. Какой метод называется перегруженным?
13. Как связываются не перегруженные виртуальные методы?
14. Как связываются не виртуальные методы?
15. Что такое RTTI? Как включается RTTI?
16. В чем заключается динамическое приведение типов?
17. Для каждого из приведенных ниже классов опишите содержание таблицы виртуальных методов.

```
struct A {
    void am1() {}
    virtual void am2() {}
    virtual void am3() {}
};
struct B : public A {
    void am1();
    virtual void am2() {}
};
struct C : public A {
    virtual void am2() {}
    virtual void cm1() {}
};
struct D : public C {
    void am1();
    virtual void am3() {}
};
```

Нарисуйте для этой иерархии диаграмму классов UML.

3. Работа ООП-203 (4 часа). Абстрактные классы

Цели:
- изучение абстрактных классов.

Задачи:
- исследование чистой виртуальной функции;
- исследование абстрактных классов;
- исследование отношения реализации;
- построение иерархий классов.

3.1. Подготовка проекта

Скачайте архив работы OOP100. Извлеките каталог OOP100 из архива в корневой каталог диска C:. Переименуйте каталог в OOP203. Откройте проект. Убедитесь, что целевой платформой является x86 или Win32.

Модуль main.cpp содержит основную функцию.

Укажите в начале этого модуля сведения об организации, о себе, о проекте, тему и дату начала работы:

```
// ОТИ НИЯУ МИФИ
// 1ПО-00Д
// Студент Имя Отчество
// Объектно-ориентированное программирование
// OOP-203. Абстрактные классы
// 01.01.2000
```

3.2. Абстрактный класс

Абстрактным называется класс, в котором есть хотя бы одна *чистая виртуальная функция* (C++). При этом создать экземпляр абстрактного класса нельзя *по определению*. Иначе говоря, абстрактный класс, — это абстрактный тип данных в чистом виде.

Роль абстрактных классов в современном ООП исключительно высока. Абстрактные классы описывают *базовые* функциональности сущностей и являются *базовыми* классами в иерархиях.

3.3. Чистая виртуальная функция

Чистой виртуальной функцией называется виртуальная функция, тело которой объявлено как *чистый спецификатор* (*pure specifier*).

Сам по себе спецификатор выглядит как присвоение нуля " = 0". В таблице виртуальных функций для чистой виртуальной функции вместо адреса записывается специальная переменная __purecall (*чистый вызов*).

Чистый спецификатор означает, что у функции *нет, и не будет тела*.

Это, в свою очередь, значит, что нельзя создать экземпляр класса, в котором есть хотя бы одна чистая виртуальная функция.

Модуль class.h, описываем обобщенный класс фигуры:

```
// фигура
class figure {
    int x, y;
public:
    figure(int a = 0, int b = 0) : x(a), y(b) {}
    // чистая виртуальная функция
    virtual void draw(void) = 0;
};
```

Модуль main.cpp, функция main:

```
int main(void) {
    figure * f = new figure;
}
```

Здесь мы пробуем создать объект типа figure. Увы, компилятор сообщает нам «*cannot instantiate abstract class*», что означает, что невозможно создать экземпляр абстрактного класса.

Модуль class.h, описываем класс кружочка:

```
// кружок
class circle : public figure {
public:
    void draw(void) {
        printf("circle\n");
    }
};
```

Функция main, создаем объект типа circle:

```
int main(void) {
    figure * f = new circle;
    f->draw();
}
```

Объект успешно создается и выводит «*circle*».

Стоит остановиться и обдумать, что есть что.

Во-первых, мы выяснили, что тип figure является *абстрактным*. Это значит, что объект этого типа создать нельзя. Это должно быть понятно. Если у метода объекта нет тела, то вызов такого метода приведет к краху, поэтому компилятор сопротивляется.

Во-вторых, использовать тип figure как таковой нельзя, но использовать указательный тип figure* можно! (и нужно!).

Остается понять, а зачем вообще нужен чистый виртуальный метод?

А давайте посмотрим так.

Для полиморфизма требуется производный класс. Он у нас есть. Указатель на абстрактный тип figure* объявить можно. Полиморфизм нам требуется для того, чтобы через указатель на базовый класс можно было вызвать виртуальный метод circle::draw.

Что произойдет, если в классе circle не будет метода draw?

А то, что класс circle тоже станет абстрактным!

Ведь в его таблице виртуальных методов должен быть записан указатель, а поскольку метод *не замещен* (*не перегружен*), то в таблице записывается метод базового класса, который определен как __purecall.

Иначе говоря, чтобы класс circle не стал абстрактным, *совершенно необходимо*, чтобы абстрактный метод figure::draw был замещен на не абстрактный метод circle::draw.

Единственное назначение чистого виртуального метода figure::draw заключается в том, чтобы *гарантировать наличие метода* draw в производных от figure классах (в каком-то из производных классов).

Модуль class.h, описываем класс квадратика аналогичным образом и убеждаемся в его работоспособности.

3.4. Конструктор производного класса

Временно изменим конструктор базового класса следующим образом:

```
class figure {
    int x, y;
public:
    figure(int a, int b) : x(a), y(b) {}
    . . .
};
```

Мы убрали значения по умолчанию параметров конструктора.

Таким образом, в базовом классе конструктора по умолчанию нет.

Проект сразу стал неработоспособным, компилятор сообщает, что в базовом классе нет конструктора по умолчанию: «*'figure': no appropriate default constructor available*».

Если базовый класс не определяет конструктор по умолчанию (но определяет конструктор с параметрами), производный класс *обязан* иметь какой-нибудь конструктор, который вызывает конструктор базового класса в списке инициализации.

Поэтому нужно определить конструкторы в классах circle и square, пусть это будут конструкторы с двумя параметрами по умолчанию, как это было вначале в классе figure. Вот пример конструктора класса circle:

```
class circle : public figure {
public:
    circle(int a = 0, int b = 0) : figure(a, b) {}
    . . .
};
```

После определения конструкторов производных классов проект снова должен стать работоспособным. Функция main:

```
int main(void) {
    figure * f = new circle(1, 1);
    f->draw();
    f = new square;
    f->draw();
}
```

3.5. Методы базового класса

Для вывода положения объекта определим в классе figure виртуальный метод virtual void location(void). Этот метод выводит в консоль строку вида "(x, y)", где x и y — числа, соответствующие координатам.

После определения этого метода в классе figure в методах draw производных классов circle и square вызовем метод location *перед* выводом названия класса. Если все сделано правильно, программа должна выводить в консоль следующий результат:

```
(1,1)circle
(0,0)square
```

Определим в базовом классе figure пару не виртуальных методов, возвращающих значения координат x и y:

```
class figure {
    . . .
    // возвращает x
    int getX() { return x; }
    // возвращает y
    int getY() { return y; }
};
```

Переопределим метод location в классе square следующим образом:

```
class square : public figure {
    . . .
    // выводит положение
    void location(void) {
        printf("[%d,%d]", getX(), getY());
    }
};
```

Теперь вывод в консоль должен измениться:

```
(1,1)circle
[0,0]square
```

Собственно, так и должно было быть.

Но иногда требуется, чтобы вместо замещенного метода вызывался именно метод базового класса. В этом случае при вызове метода location нужно использовать разрешение видимости базового класса figure:

```
class square : public figure {
    . . .
    void draw(void) {
        figure::location();
        printf("square\n");
    }
};
```

Вывод программы вернулся к прежнему выводу:

```
(1,1)circle
(0,0)square
```

Наконец, нам потребуется также метод для изменения положения.

Это не виртуальный метод void move(int a, int b) в классе figure. Определяем этот метод и проверяем работоспособность в функции main:

```
    f->move(2, 2);
    f->draw();
```

Удостоверяемся, что в консоль выводится:

```
(1,1)circle
(0,0)square
(2,2)square
```

3.6. Приложение

Модуль class.h, описываем приложение paintapp:

```
// приложение Paint
#define BASECLASS figure
class paintapp {
    BASECLASS * first;
public:
    paintapp() : first(0) {}
};
```

Функция main содержит только объявление переменной app:

```
int main(void) {
    paintapp app;
}
```

Чтобы можно было сформировать список объектов, добавим в базовый класс figure указатель next на следующий объект. Этот указатель нужно обнулить в конструкторе:

```
class figure {
    int x, y;
public:
    figure * next;
    figure(int a, int b) : next(0), x(a), y(b) {}
    . . .
};
```

Описываем метод paintapp::add для добавления нового объекта:

```
class paintapp {
    . . .
    // добавляет объект
    void add(BASECLASS * ob) {
        if (first == 0) {
            first = ob;
        } else {
            BASECLASS * temp = first;
            while (temp->next) {
                temp = temp->next;
            }
            temp->next = ob;
        }
    }
};
```

Описываем метод paintapp::draw(void), который рисует объекты:

```
class paintapp {
    . . .
    // рисует объекты
    void draw(void) {
        BASECLASS * temp = first;
        while (temp) {
            temp->draw();
            temp = temp->next;
        }
    }
};
```

В функции main добавляем пару объектов и рисуем их:

```
int main(void) {
    paintapp app;
    app.add(new circle(1, 1));
    app.add(new square(2, 2));
    app.draw();
}
```

Деструктор класса paintapp определите сами, если будет время.

Добавим метод paintapp::item, который позволит получить доступ к произвольному объекту по его порядковому номеру:

```
class paintapp {
    . . .
    // возвращает объект по номеру
    BASECLASS * item(int index) {
        index = abs(index);
        assert(first != NULL);
        BASECLASS * temp = first;
        while (temp) {
            if (--index == 0) return temp;
            temp = temp->next;
        }
        int index_overflow = 0;
        assert(index_overflow);
    }
};
```

Здесь мы убеждаемся, что в списке есть объекты, при помощи первого вызова функции assert. Второй вызов assert нужен для того, чтобы остановить программу, если будет задан слишком большой индекс. Переменная index_overflow введена для того, чтобы получить внятное пояснение в случае остановки программы.

Абсолютная величина индекса делает его положительным на случай непредвиденного задания отрицательного значения. Все эти проверки при работе с указателями существенно необходимы.

В основной функции используем новый метод, убеждаемся, что второй добавленный объект перемещается:

```
int main(void) {
    paintapp app;
    app.add(new circle(1, 1));
    app.add(new square(2, 2));
    app.item(2)->move(3, 3);
    app.draw();
}
```

3.7. Интерфейсы и отношение реализации

Будем называть *интерфейсом* класс, состоящий только из чистых виртуальных функций. Как и всякий абстрактный класс, интерфейс может служить только в качестве базового класса. В отличие от абстрактного класса, интерфейс ничего не передает в наследство производному классу, кроме чистой функциональности. Интерфейс *обязывает* производный класс иметь определенный набор функций, то есть обладать определенным поведением.

При наследовании интерфейса возникает не отношение *обобщения*, а отношение *реализации*. Оно обозначается как заключение контракта: производный класс обязуется *реализовать* интерфейс. В отличие от наследования реализации, при множественном наследовании поведения не возникает неоднозначности.

При множественном наследовании для каждого наследуемого класса используется своя таблица виртуальных методов, а объект имеет столько указателей vptr, сколько классов наследуется. Если два базовых класса предписывают реализовать один и тот же метод, таблицы vtable для этих классов содержат разные указатели, в конечном итоге ведущие к одной реализации.

При этом объект производного класса может быть приведен к любому из наследуемых типов обычным или статическим приведением.

Например, если тип C наследует два типа A и B, то допустимо:

```
C * c = new C;
A * a = (A*)b;
B * b = (B*)a;
A * aa = static_cast<A*>(c);
B * bb = static_cast<B*>(c);
```

Модуль class.h, описываем данную иерархию:

```
struct A {
    virtual void f(void) = 0;
};

struct B {
    virtual void f(void) = 0;
};

struct C : public A, public B {
    void f(void) {}
};
```

В основной функции main описываем приведенную выше последовательность приведений. Выполняем строчки кода и исследуем объекты:

Watch 1

Name	Value	Type
a	0x00342a70	A *
[C]	{...}	C
__vfptr	0x00406790 const C::`vftable'{for `A'}	*
[0]	0x0040109b C::f(void)	*
b	0x00342a74	B *
[C]	{...}	C
__vfptr	0x00406740 const C::`vftable'{for `B'}	*
[0]	0x00401096 [thunk]:C::f`adjustor{4}' (void)	*

Видим, что если объект с трактуется как A*, то используется таблица виртуальных методов для класса A, а если как B*, то таблица для класса B.

При этом в таблице виртуальных методов класса B указан переключатель (преобразователь) [thunk], который переключает (преобразует) вызов B::f() в A::f(), фактически выполняя приведение указателя к типу A, на что указывает adjustor{4}. 4 — разница адресов указателей vptr.

Объект типа C состоит из двух указателей: сначала указатель на vtable for A, затем указатель на vtable for B. Указатель a указывает на объект, то есть на vtable for A, а указатель b указывает на второй указатель, то есть имеет адрес a + 4.

Так примерно выглядит таблица vtable for A (ассемблер):

```
??_7C@@6BA@@@ DD FLAT:??_R4C@@6BA@@@            ; C::`vftable'
    DD   FLAT:?f@C@@UAEXXZ
```

Так примерно выглядит таблица vtable for B:

```
??_7C@@6BB@@@ DD FLAT:??_R4C@@6BB@@@            ; C::`vftable'
    DD   FLAT:?f@C@@W3AEXXZ
```

Если посмотреть на реализацию метода ?f@C@@W3AEXXZ, который соответствует методу B::f(), то увидим:

```
?f@C@@W3AEXXZ PROC                ; [thunk]:C::f`adjustor{4}', COMDAT
    sub ecx, 4
    jmp ?f@C@@UAEXXZ              ; C::f
```

Здесь регистр ecx — это указатель this. Из кода видно, как из this вычитается 4, он начинает указывать на vtable for A, после чего происходит перенаправление (jump) на адрес ?f@C@@UAEXXZ, который соответствует методу A::f(), что в конечном итоге является методом C::f().

3.8. Интерфейс для фигур

Перед классом figure опишем интерфейс рисуемых объектов:

```
// интерфейс рисуемый
class drawable {
public:
    virtual void draw(void) = 0;
};
```

Объекты, реализующие интерфейс drawable, должны иметь метод для рисования draw. Давайте добавим наследование интерфейса drawable в классы circle и square:

```
// кружок
class circle : public figure, public drawable {
    . . .
};

// квадрат
class square : public figure, public drawable {
    . . .
};
```

В работе программы ничего не должно меняться.

Однако сами объекты изменились.

Исследуем объект класса circle.

Для этого создадим объект класса c и рассмотрим его по частям.

Функция main:

```
int main(void) {
    circle * c = new circle(3, 7);
    printf("0 = 0x%x\n", *((int*)c + 0));
    printf("1 = 0x%x\n", *((int*)c + 1));
    printf("2 = 0x%x\n", *((int*)c + 2));
    printf("3 = 0x%x\n", *((int*)c + 3));
    printf("4 = 0x%x\n", *((int*)c + 4));
}
```

Выполняем программу, но не завершаем ее, и исследуем объект:

Watch 1

Name	Value	Type
c	0x00342a70	circle *
figure	{x=3 y=7 next=0x00000000 }	figure
__vfptr	0x0040674c const circle::`vftable'{for `figure'}	*
[0]	0x00401014 figure::location(void)	*
[1]	0x00401005 circle::draw(void)	*
x	3	int
y	7	int
next	0x00000000 {x=??? y=??? next=??? }	figure *
drawable	{...}	drawable
__vfptr	0x00407194 const circle::`vftable'{for `drawable'}	*
[0]	0x00401118 [thunk]:circle::draw`adjustor{16}' (void)	*

Программа выводит числа:

```
0 = 0x40674c
1 = 0x3
2 = 0x7
3 = 0x0
4 = 0x407194
```

Первое число, — это указатель на vtable for figure.

Второе число, — это x, третье, — это y, четвертое, — это next.

Четвертое число, — это указатель на vtable for drawable.

Поскольку базовые классы circle обязывают иметь один и тот же метод, vtable for drawable содержит переключатель [thunk], использующий приведение circle* к drawable*, при этом разница адресов соответствующих указателей vptr составляет 16, потому что объект c имеет следующее устройство:

Смещение	*Элемент*
0	vptr vtable for figure
4	x
8	y
12	next
16	vptr vtable for drawable

Если, например, изменить тип данных x и y на short, то смещение составит не 16, а 12 байт, убедиться в чем несложно.

3.9. Сила абстракций

Мы не случайно определили константу BASECLASS.

Одна из целей программирования заключается в проектировании возможно более абстрактного кода.

Рассмотрим приложение класса paintapp.

Во-первых, это приложение, которое умеет рисовать, в смысле использовать метод void draw(void) для этой цели. Иначе говоря, объекты для приложения должны быть *рисуемыми*, в смысле должны определять метод void draw(void).

Во-вторых, это приложение, которое умеет составлять динамический список объектов. Иначе говоря, объекты для приложения должны быть *связываемыми*, то есть обладать указателем next.

Возникает вопрос: насколько абстрактно приложение типа paintapp?

Сейчас оно может коллекционировать только объекты, производные от базового класса figure, которые *являются* рисуемыми и связываемыми.

Можно ли использовать это же приложение для включения объектов иерархии других классов, производных не от класса figure?

Сейчас ответ — нет.

Наверное, paintapp хорошее приложение, но степень его абстракции не очень высока. И если думать в направлении проектирования более абстрактного кода, то использование типа figure в качестве базового для этого приложения ограничивает его возможности. Буквально, если завтра нам потребуется включить в список допускаемых объектов другие объекты, производные не от класса figure, нам потребуется полностью переделывать приложение.

И это неправильно.

Нам нужно использовать силу абстракций в прямом смысле: приложению должны быть доступны любые объекты, которые являются *рисуемыми* и *связываемыми*, где два последних термина обозначают наследование соответствующих интерфейсов, абстрактных классов без реализации.

Для начала наши объекты *являются* рисуемыми.

Только нужно устранить двойственность методов.

Для этого сделаем производным от интерфейса drawable не классы circle и square, а базовый класс figure.

С помощью кода функции main можно убедиться, что объекты класса circle снова изменились, у них исчез указатель на vtable for drawable, он переместился в объект типа figure.

Облагородим класс square.

Во-первых, исправим метод square::draw, убрав из него разрешение видимости класса figure::. Во-вторых, удалим метод location, который мы использовали для своих исследований.

Кроме того, удалим классы A, B, C, они не понадобятся больше.

Далее нужно думать, как сделать приложение более абстрактным.

Попробуем переопределить константу BASECLASS:

```
#define BASECLASS drawable
```

Как только мы сделаем это, приложение рассыплется, потому что объекты типа drawable не имеют указателя next.

Как поступить в этом случае? Это сложный вопрос и ответ во многом зависит от системы программирования.

Можно, например, в данном конкретном случае ввести в интерфейс drawable методы для управления указателем next, которые автоматически сделают рисуемые объекты связываемыми. Но с точки зрения логики, это не одно и то же. Некоторые другие приложения могут потребовать только связываемые объекты, и тогда подставляемые в него иерархии, основанные на интерфейсе drawable, не будут подходить.

Разная функциональность должна определяться разными интерфейсами, — вот важный принцип, который следует поддерживать.

Нас бы вполне устроил такой подход: определить два интерфейса drawable и linkable, и на их основе создать интерфейс paintable, который можно затем использовать как базовый тип для paintapp. Немного, правда, сложно и запутано, но вполне реализуемо.

Мы так делать не будем.

Нам интересно, чтобы интерфейсы использовались сами по себе, а не комбинировались в какие-то невообразимые и трудно понимаемые структуры.

Для начала определим интерфейс связываемых объектов.

Модуль class.h, перед интерфейсом drawable:

```
// интерфейс связываемый
class linkable {
public:
    virtual void set_next(linkable *) = 0;
    virtual linkable * get_next(void) = 0;
};
```

Наследуем класс figure от интерфейса linkable тоже:

```
// фигура
class figure : public drawable, public linkable {
    . . .
};
```

Класс figure должен определять методы linkable:

```
class figure : public drawable, public linkable {
    . . .
    // устанавливает next
    void set_next(linkable * a) {
        next = static_cast<figure*>(a);
    }
    // возвращает next
    linkable * get_next(void) {
        return static_cast<linkable*>(next);
    }
};
```

И остается открытым вопрос, какой тип должен быть базовым для приложения paintapp?

Ответ зависит от того, в каком контексте используется тип.

Если тип требуется для хранения, то подойдет тип void*. Любой указательный тип приводится к нему.

Если тип требуется для связывания, то он должен быть linkable*.

Если тип требуется для рисования, он должен быть drawable*.

Глядя на код методов paintapp, можно заметить, что чаще всего требуется тип для связывания. Поэтому можно предложить linkable в качестве базового типа:

```
#define BASECLASS linkable
```

Далее нужно избавиться от указателя next в коде всех методов.

Это можно сделать следующими заменами:

"->next = ob"	заменяем на	"->set_next(ob)"
"->next"	заменяем на	"->get_next"

После этого программа не станет более работоспособной, потому что тип linkable не имеет метода draw. Нам требуется приведение.

Использовать здесь можно только *динамическое* приведение.

Чтобы во время рисования не нужно было проверять, что приведение успешно, нужно проверку сделать в методе add.

Метод paintapp::add:

```
    void add(BASECLASS * ob) {
        assert(dynamic_cast<drawable*>(ob));
        . . .
    }
```

Если будет подставлен объект, не являющийся drawable, программа будет остановлена функцией assert.

Метод paintapp::draw:

```
    void draw(void) {
        BASECLASS * temp = first;
        while (temp) {
            dynamic_cast<drawable*>(temp)->draw();
            temp = temp->get_next();
        }
    }
```

Код функции main:

```
int main(void) {
    paintapp app;
    app.add(new circle(1, 1));
    app.add(new square(2, 2));
    //app.item(2)->move(3, 3);
    app.draw();
}
```

Теперь программа должна работать нормально, за исключением того, что попытка применить метод move к объекту, возвращаемому методом paintapp::item, окажется безуспешной. Это должно быть понятным: связываемый объект не имеет метода move. Эта проблема также решается при помощи динамического приведения.

Предлагаю добавить еще один интерфейс, moveable (*перемещаемый*), с единственным виртуальным методом void move(int a, int b).

После этого класс figure объявим наследником moveable тоже.

Перемещение объекта можно сделать тогда следующим образом:

```
int main(void) {
    paintapp app;
    app.add(new circle(1, 1));
    app.add(new square(2, 2));
    moveable * m = dynamic_cast<moveable*>(app.item(2));
    if (m) m->move(3, 3);
    app.draw();
}
```

Заметим, что можно было бы также метод move внести в интерфейс drawable, но так мы более дискретно разделили функциональность.

Можно также избавиться от константы BASECLASS, заменив ее текст на linkable, после чего вообще удалить. Текст станет более понятным.

Теперь можно убедиться в том, что приложение позволяет добавлять произвольные объекты, являющиеся *рисуемыми* и *связываемыми*, но не обязательно являющиеся *перемещаемыми*.

Модуль class.h. Новый класс:

```
// текст
struct text : public drawable, public linkable {
    linkable * next;
    text() : next(0) {}
    // устанавливает next
    void set_next(linkable * a) {
        next = a;
    }
    // возвращает next
    linkable * get_next(void) {
        return next;
    }
    // рисует
    void draw(void) {
        printf("text\n");
    }
};
```

Добавить объект типа text в приложение paintapp можно, можно получить ссылку на этот объект при помощи метода paintapp::item, но преобразовать ее к типу moveable не удастся, объекты этого типа не являются перемещаемыми.

Приложение paintapp в конечном итоге получилось в достаточной степени абстрактным.

3.10. Вопросы и упражнения

1. Что называется чистой виртуальной функцией?
2. Какой класс называется абстрактным?
3. Что называется интерфейсом?
4. Чем интерфейс отличается от абстрактного класса?
5. Чем отношение реализации отличается от отношения обобщения?
6. Поясните устройство объекта класса D для следующей иерархии:

```
struct A {
    virtual void fa(void) = 0;
};
struct B {
    int b;
    virtual void fa(void) = 0;
};
struct C {
    int c;
    virtual void fb(void) = 0;
};
struct D : public C, public B, public A {
    int d;
    virtual void fa(void) {};
    virtual void fb(void) {};
};
```

7. Можно ли в классе figure заменить тип указателя next, если можно, то на какой тип?