Тема 2.1 Абстрактные классы

Абстрактный класс в объектно-ориентированном программировании — базовый класс, который не предполагает создания экземпляров. Абстрактные классы реализуют на практике один из принципов ООП — полиморфизм. Абстрактный класс может содержать (и не содержать) абстрактные методы и свойства. Абстрактный метод не реализуется для класса, в котором описан, однако должен быть реализован для его неабстрактных потомков. Абстрактные классы представляют собой наиболее общие абстракции, то есть имеющие наибольший объём и наименьшее содержание.

Когда виртуальная функция не переопределена в производном классе, то при вызове ее в объекте производного класса вызывается версия из базового класса. Однако во многих случаях невозможно ввести содержательное определение виртуальной функции в базовом классе. В таких случаях необходим метод, гарантирующий, что производные классы действительно определят все необходимые функции. Язык C++ предлагает в качестве решения этой проблемы чисто виртуальные функции.

Чисто виртуальная функция (pure virtual function) является функцией, которая объявляется в базовом классе, но не имеет в нем определения. Поскольку она не имеет определения, то есть тела в этом базовом классе, то всякий производный класс обязан иметь свою собственную версию определения. Для объявления чисто виртуальной функции используется следующая общая форма:

virtual тип имя_функции(список параметров) = 0;

При введении чисто виртуальной функции в производном классе обязательно необходимо определить свою собственную реализацию этой функции. Если класс не будет содержать определения этой функции, то компилятор выдаст ошибку. Например, если попытаться откомпилировать программу figure, в которой удалено определение функции snow_area() из класса circle, то будет выдано сообщение об ошибке:

```
#include <iostream.h>
class figure {
protected:
double x, y;
public:
void set_dim(double i, double j) {
x = i;
y = j;
}
virtual void show_area() = 0; // pure
};
class triangle: public figure {
public:
void show_area() {
cout << "Triangle with height ";
cout << x << " and base " << y;
cout << " has an area of ";
cout << x * 0.5 * y << ". \ n";
```

```
}
};
class square: public figure {
public:
void show_area() {
cout << "Square with dimensions ";
cout << x << "x" << y;
cout << " has an area of ";
cout << x * y << " . \n";
}
};
class circle: public figure {
// определение show_area() отсутствует и потому выдается ошибка
};
int main ( )
{
figure *p; // создание указателя базового типа
circle c; // попытка создания объекта типа circle - ОШИБКА
triangle t; // создание объектов порожденных типов
square s;
p = &t;
p->set_dim(10.0, 5.0);
p->show_area ();
p = &s;
p->set_dim(10.0, 5.0);
p->show_area();
return 0;
}
```

На абстрактный класс можно объявить ссылку, но нельзя создать экземпляр (объект). Ссылку можно использовать для того, чтобы ссылаться на любой класс, производный от абстрактного.

Абстрактные классы активно используются для создания т.н. пользовательского интерфейса. Ссылаясь через абстрактный класс можно “прятать” реализацию функции, предоставляя доступ к функционалу. То есть, наследование от абстрактных классов также помогает в осуществлении важной парадигмы объектно-ориентированного программирования – инкапсуляции.

Любой класс всегда неявно объявляет свой интерфейс — то, что доступно при использовании класса извне. Если у нас есть класс Ключ и у него публичный метод Открыть, который вызывает приватные методы Вставить, Повернуть и Вынуть, то интерфейс класса Ключ состоит из метода Открыть. Когда мы унаследуем какой-то класс от класса Ключ, он унаследует этот интерфейс. Кроме этого интерфейса, у класса есть также реализация — методы Вставить, Повернуть, Вынуть и их вызов в методе Открыть. Наследники Ключа наследуют вместе с интерфейсом и реализацию.
И вот здесь таятся проблемы. Предположим, у нас есть некая модель, которая предполагает использование ключа для открытия двери. Она знает интерфейс Ключа и поэтому вызывает метод Открыть.

Но, предположим, некоторые двери открываются не таким вот поворот-

ным ключом, а магнитной карточкой — которая ведь тоже по своей сути ключ. Интерфейс этой карточки никак принципиально не отличается от интерфейса обычного ключа — можно Открыть ключом, а можно Открыть карточкой. И мы хотим сделать класс Магнитную Карточку, который тоже будет содержать интерфейс Ключа. Для этого мы унаследуем Магнитную Карточку от Ключа. Но вместе с интерфейсом унаследовалась и реализация, которая в методе Открыть вызывает методы Вставить, Повернуть и Вынуть — а это совершенно не подходит для Магнитной Карточки.
Нам придётся самое меньшее перегружать в Магнитной Карточке реализацию метода Открыть, используя уже последовательность Вставить, Провести и Вынуть. Это уже плохо, потому что мы не знаем детали реализации класса Ключ — вдруг мы упустили какое-то очень важное изменение данных, которое должно было быть сделано — и было сделано в методе Открыть?

Объявим интерфейс Ключ, содержащий метод Открыть.
Объявим класс Поворотный Ключ, реализующий интерфейс Ключ при помощи своих методов Вставить, Повернуть и Вынуть.
Объявим класс Магнитная Карточка, тоже реализующий интерфейс Ключ, но уже по-своему — и без каких-либо неприятных пересечений с реализацией Поворотного Ключа. Этого помогло нам достичь отделение интерфейса от реализации.

Всегда помните об общем принципе: нам нужно использовать не класс, а интерфейс. Нам не важно, что это за штука — поворотный ключ или магнитная карточка, — нам важно, что им можно открыть дверь. То есть, вместо того, чтобы задумываться о природе объекта, мы задумываемся о способах его использования.

Абстрактные классы являются важной частью современного объектно-ориентированного программирования, особенно касательно создания API. В более современных языках, как C# и Java, идея абстрактных классов как интерфейсов получила продолжение в так называемы интерфейсах, классах, только объявляющих методы, но не реализующих их.