19. ПОЛИМОРФИЗМ

Одно из самых коротких и выразительных определений полиморфизма таково: полиморфизм – это функциональная возможность, позволяющая старому коду вызвать новый. Это свойство дает возможность расширять и совершенствовать программную систему, не затрагивая существующий код. Осуществляется такой подход с помощью механизма виртуальных функций.

19.1. Раннее и позднее связывание

К механизму виртуальных функций обращаются в тех случаях, когда в базовый класс необходимо поместить функцию, которая должна по-разному выполняться в производных классах. Точнее, по-разному должна выполняться не единственная функция из базового класса, а в каждом производном классе требуется свой вариант этой функции.

Предположим, необходимо написать функцию-член CalculatePay() (Расчет), которая подсчитывает для каждого объекта класса Employee (Служащий) ежемесячные выплаты. Все просто, если зарплата рассчитывается одним способом: можно сразу вставить в вызов функции тип нужного объекта. Проблемы начинаются с появлением других форм оплаты. Допустим, уже есть класс Employee, реализующий расчет зарплаты по фиксированному окладу. А что делать, чтобы рассчитать зарплату контрактников? Ведь это уже другой способ расчета! В процедурном подходе пришлось бы переделать функцию, включив в нее новый тип обработки, так как в прежнем коде такой обработки нет. Объектно ориентированный подход, благодаря полиморфизму, позволяет производить различную обработку.

В таком подходе надо описать базовый класс Employee, а затем создать производные от него классы для всех форм оплаты. Каждый производный класс будет иметь собственную реализацию метода CalculatePay().

Другой пример: базовый класс figure может описывать фигуру на экране без конкретизации её вида, а производные классы triangle (треугольник), ellipse (эллипс) и т. д. однозначно определяют её форму и размер. Если в базовом классе ввести функцию void show() для изображения фигуры на экране, то выполнение этой функции будет возможно только для объектов каждого из производных классов, определяющих конкретные изображения. В каждый из производных классов нужно включить свою функцию void show() для формирования

изображения на экране. Доступ к функции show() производного класса возможен с помощью явного указания ее полного имени, например triangle::show();
или с использованием конкретного объекта:

triangle t; t.show();

Однако в обоих случаях выбор нужной функции выполняется при написании исходного текста программы и не изменяется после компиляции. Такой режим называется ранним или статическим связыванием.

Большую гибкость, особенно при использовании уже готовых библиотек классов, обеспечивает так называемое позднее, или отложенное, или динамическое связывание, которое предоставляется механизмом виртуальных функций.

19.2. Виртуальные функции

Рассмотрим сначала, как ведут себя при наследовании не виртуальные функции-члены с одинаковыми именами, сигнатурами и типами возвращаемых значений. struct base{ void fun (int i){

cout << "base::i = " << i << "\n";}

}; struct der: public

base{ void fun (int i){

cout << " der::i = " << i << "\n";}

}; void main(){ base B, *bp = &B; der D, *dp = &D; base *pbd = &D; // Неявное преобразование от der* к base*.

bp -> fun(1); dp -> fun(5); pbd -> fun(8); }

Результат:

base::i = 1 der::i = 5 base::i = 8 Здесь указатель pbd имеет тип base*, но его значение – адрес объекта D класса der. При вызове

функции-члена по указателю на объект выбор функции зависит **только от типа указателя**, но не от его значения, что и иллюстрируется выводом base::i = 8. Настроив указатель базового класса на объект производного класса, **не удается** с помощью этого указателя вызвать функцию из производного класса. Таким способом не удается достичь **позднего** или **динамического** связывания.

Динамическое связывание обеспечивается механизмом виртуальных функций. Любая нестатическая функция базового класса может быть сделана виртуальной, если в ее объявлении использовать спецификатор **virtual**: class base{ public: int i;

virtual void print(){

```
cout << i << "  внутри base\n";}
}; class D: public
base{ public:
void print(){
cout << i << "  внутри D\n";}
}; void main(){
base b; base
*pb = &b;
D f;
f.i = 1 + (b.i = 1); pb -> print (); pb = &f;              // Неявное
преобразование D* к Base*. pb -> print(); }
```

Результат:

```
1 внутри base
2 внутри D
```

Здесь в каждом случае выполняется различная версия функции print(). Выбор динамически зависит от типа объекта, **на который указывает указатель**. Служебное слово virtual означает, что функция print() может иметь свои версии для различных порожденных классов. Указатель на базовый класс может указывать или на объект базового класса, или на объект порожденного класса. Выбранная функция-член зависит от класса, на объект которого указывается, но не от типа указателя. При отсутствии члена производного типа по умолчанию используется виртуальная функция базового класса. Отметим различие между выбором соответствующей переопределенной виртуальной функции и выбором перегруженной функции-члена (не виртуальной): перегруженная функция-член выбирается во время компиляции алгоритмом, основанным на правиле сигнатур. При перегрузке функции-члены могут иметь разные типы возвращаемого значения. Если же функция объявлена как virtual, то все её переопределения в порожденных классах должны иметь ту же сигнатуру и тот же тип возвращаемого значения. При этом в производных классах слово virtual можно и не указывать.

В производном классе нельзя определять функцию с тем же именем и с той же сигнатурой, но с другим типом возвращаемого значения, чем у виртуальной функции базового класса. Отметим, что конструктор не может быть виртуальным, а деструктор – может.

Рассмотрим пример вычисления площадей различных фигур.

Различные фигуры будем порождать от базового класса figure.

```
class figure{ protected: double x, y; virtual double area(){
return 0;       // Площадь по умолчанию.
}
};

class rectangle: public figure{ private:
double height, width;
. . . public:
rectangle(double h, double w){
height = h;  width
= w;
}  double  area(){return  height  *
width;}
. . .
};

class circle: public figure{
const double pi; double
radius;
. . . public: circle (double r): pi(3.14159265358979),
radius(r){} double area(){ return pi * radius *
radius; }
. . .
};
```

Код пользователя может выглядеть так:

```
const  int  N  =  30;
figure          *p[N];
double tot_area = 0;
. . .       // Здесь устанавливаются указатели p[i], например,
. . .       // rectangle r(3, 5); p[0] = &r; circle c(8); p[1] = &c; и т. д.
for (int i = 0; i < N; i++)  tot_area += p[i] -> area ();
// Код пользователя.
```

Главное преимущество состоит в том, что код пользователя не нуждается в изменении, даже если к системе фигур добавляются новые.

19.3. Абстрактные классы

Снова рассмотрим пример с вычислением площадей фигур. В этой программе использована виртуальная функция area(). Эта функция должна была быть первый раз определена в классе figure. Поскольку при нормальной работе не должно существовать объектов типа figure, а только объекты производных от него типов, то версия area () была определена так:

```
double figure::area{return 0;}
```

Если бы тип возвращаемого значения у функции был void (например, при рисовании фигуры void show ( )), можно было бы написать:

```
void figure::show(){}
```

В обоих случаях эти функции фиктивны. Такого рода виртуальные функции можно было бы использовать для контроля ошибок, связанных с созданием объектов типа figure:

```
double figure::area(){
cout <<"Ошибка: попытка вычислить площадь ";
cout << "несуществующего объекта!\n";
exit(1); return 1;}
```

В C++ существует более удобный и надежный способ. Версия виртуальной функции, которая, с одной стороны, должна быть определена, а с другой – никогда не должна использоваться, может быть объявлена как **чисто виртуальная функция**: class figure{. . . virtual double area() = 0; };

В классах, производных от figure, при наличии своей версии виртуальной функции area() она должна либо быть определена, либо объявлена как чисто виртуальная функция. Во время выполнения программы при обращении к чисто виртуальной функции выдается сообщение об ошибке, и программа аварийно завершается. Класс, содержащий хотя бы одну чисто виртуальную функцию, называется **абстрактным** классом. Запрещено создание объектов таких классов. Это позволяет установить контроль со стороны компилятора за ошибочным созданием объектов фиктивных типов, подобных figure. Заметим, что можно создавать указатели и ссылки на абстрактные классы.