НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ ЯДЕРНЫЙ УНИВЕРСИТЕТ «МИФИ»
Кафедра информатики и процессов управления (№17)

Лекции по информатике и программированию

Лекция 6

Наследование и полиморфизм (часть 1)

Содержание

9. Наследование и полиморфизм адресов

9.1. Иерархические предметные области

Часто при разработке крупных проектов объекты предметной области объединяют в категории и подкатегории, создавая ***иерархию типов объектов***.

Например, в моделирующей дорожное движение программе может рассматриваться следующая иерархия ***категорий*** транспортных средств:

Каждая категория объектов обладает специфичными для нее ***свойствами*** объектов. Если категория сама разделяется на подкатегории, то ее свойствами обладают и все объекты каждой из ее подкатегорий.

Например, такими свойствами как «текущая скорость», «направление движения» обладают объекты всех категорий рассматриваемой иерархии.

Категория «Автомобили» и все ее подкатегории имеют свойства «объем двигателя», «тип топлива», «объем топливного бака», «количество топлива в баке». Для автомобилей должна быть определена операция «заправка топливом». Но всеми перечисленными свойствами не обладают объекты категорий «Трамвай», «Велосипедный транспорт», «Прицеп».

Объекты категории «Грузовой автомобиль» имеют свойство «объем кузова», которого нет у других подкатегорий «Автомобилей». Поэтому только для грузовиков должны быть определены операции «погрузка» и «разгрузка».

Для описания приведенной иерархии необходимы объекты, которые, с одной стороны, имеют различный набор полей и методов, но, с другой стороны, содержат некоторые общие поля и методы. В объектно-ориентированном программировании подход к такому описанию иерархии объектов называется ***наследованием***.

Наследование от ***объектов-предков*** к ***объектам-потомкам*** рассматривается как ***переход об общего к частному***.

Объекты-потомки выступают как частные случаи объектов-предков. При таком направлении количество информации увеличивается: в объектах-потомках содержится больше информации, чем в объектах-предках.

Если в программе применяется только **инкапсуляция** и **методы** объектов, то такой подход называется ***object-based programming***. Полноценное ООП – ***object-oriented programming*** – возникает только при наличии **наследования**, которое может сопровождаться **динамическим полиморфизмом**.

9.2. Наследование структур

Вначале рассмотрим упрощенный случай: пусть объект является структурой (`struct`), которая не имеет функций-членов (методов).

Например, имеется структура, описывающая человека (персону):

```
struct Person {
    char name[64];       // имя
    char sex;            // пол, значения: 'm' или 'f'
    int year_of_birth;   // год рождения
};
```

Теперь опишем студента, который также является персоной:

```
struct Student {
    char name[64];         // имя
    char sex;              // пол, значения: 'm' или 'f'
    int year_of_birth;     // год рождения
    int spec_code;         // код специальности
    int year_of_study;     // курс обучения в вузе
    double average_score; // средний балл
};
```

Первые три поля структур `Person` и `Student` совпадают, поэтому структура `Student` может **унаследовать** их от структуры `Person`. Для этого изменим описание структуры `Student`:

```
struct Student : Person {     // Student унаследована от Person
    int spec_code;         // код специальности
    int year_of_study;     // курс обучения в вузе
    double average_score; // средний балл
};
```

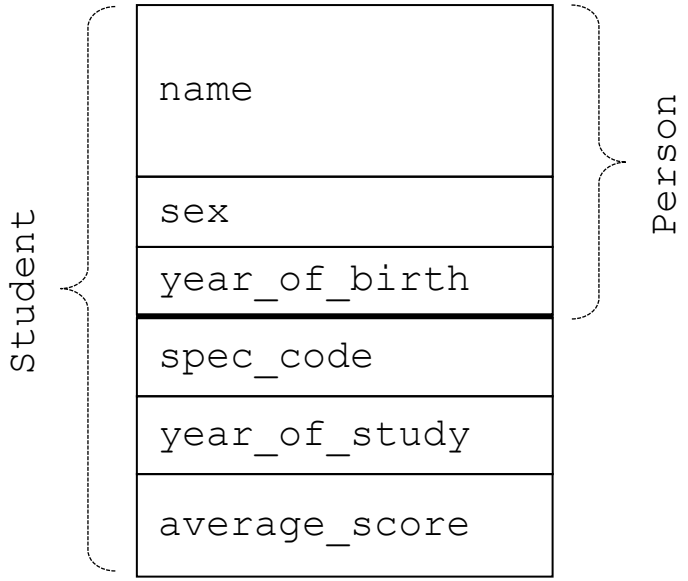
Теперь `Student` является структурой-**потомком** от `Person`, ставшей структурой-**предком**. Для обозначения отношений наследования может использоваться следующая терминология:

Person	Student
предок	***потомок***
базовый	***унаследованный***
родительский	***порожденный***
родительский	***дочерний***

Терперь можно создать экземпляр (объект) структуры-потомка – переменную типа `Student`, заполнив значениями все ее поля:

```
Student stud;
strcpy(stud1.name, "Ivanov Ivan");
stud.sex = 'm';
stud.year_of_birth = 2004;
stud.spec_code = 140501;
stud.year_of_study = 3;
stud.average_score = 4.56;
```

Поля структуры `Student`, унаследованные от структуры `Person`, располагаются на тех же местах (то есть с тем же смещением относительно начала), что и в структуре `Person`. Поэтому при необходимости **с переменной типа** `Student` **можно работать точно так же, как если бы она имела тип** `Person`. Обратное неверно: в структуре `Person` отсутствуют поля, имеющиеся в структуре `Student`.



Расположение полей переменной `stud`.

9.3. Полиморфизм адресов

На основе описанного принципа расположения полей действует следующее правило, определяющее ***полиморфизм адресов*** в язык Си ++:

> В языке Си++ для указателей и ссылок разрешено **неявное преобразование** адреса ***объекта-потомка*** в адрес ***объекта-предка***.
>
> ***Полиморфизм адресов***: *указатель* или *ссылку* на *объект-потомок* можно использовать **везде**, где требуется *указатель* или *ссылка* на *объект-предок*.

В рассмотренном примере адрес типа «`Student *`» можно неявно преобразовать в адрес типа «`Person *`». Например:

```
Student stud;    // stud - объект типа Student.
Person *ppers;   // ppers - указатель на объект типа Person.
...
ppers = &stud;   // В указатель ppers записываем адрес объекта stud.
Person &rpers = stud;  // В ссылку rpers записываем адрес объекта stud.
```

Такое неявное преобразование так же происходит при передаче параметров в функцию:

```
void func(Person &rpers) {  // Формальный параметр функции func -
    ...                     // - ссылка на объект типа Person.
}
Student stud;  // stud - объект типа Student.
...
func(stud);  // Передача адреса объекта stud в функцию func по ссылке.
```

Преобразование в обратную сторону не выполняется:

```
Person pers;      // pers - объект типа Person.
Student *pstud;   // pstud - указатель на объект типа Student.
...
pstud = &pers;   // ошибка компиляции!
Student &rstud = pers;   // ошибка компиляции!
```

Обратное преобразование не выполняется и при передаче параметров в функцию:

```
void func(Student &rstud) {   // Формальный параметр функции func -
    ...                       // - ссылка на объект типа Student.
}
Person pers;   // stud - объект типа Person.
...
func(pers);   // ошибка компиляции!
```

Ранее описывались проявления полиморфизма: полиморфизм, встроенный в язык; перегрузка функций; перегрузка операций. **Полиморфизм адресов** принципиально отличается от описанных ранее проявлений полиморфизма и относится к объектно-ориентированному программированию.

> ***Полиморфизм*** состоит в том, что *операция* или *действие*, **обозначаемые одинаково** (одним и тем же символом), корректно выполняются для *операндов* и *параметров* **различных типов**.

9.4. Отличие классов от структур

И структуры (`struct`), и классы (`class`) могут иметь функции-члены (методы).

В языке Си++ структуры (`struct`) отличаются от классов (`class`) только моделью защиты (`private` / `public`), используемой по умолчанию.

Примеры эквивалентных структур (`struct`) и классов (`class`):

Структура (`struct`)	Класс (`class`)

`struct My {  // по умолчанию`
`  int a;`
`  double x;`
`  My(int a0, double x0);`
`  ~My();`
`  void print();`
`  int calc(double y);`
`};`
	`class My {`
	`public:`
	`  int a;`
	`  double x;`
	`  My(int a0, double x0);`
	`  ~My();`
	`  void print();`
	`  int calc(double y);`
	`};`

`struct My {`
`private:`
`  int a;`
`  double x;`
`public:`
`  My(int a0, double x0);`
`  ~My();`
`  void print();`
`  int calc(double y);`
`};`
	`class My {  // по умолчанию`
	`  int a;`
	`  double x;`
	`public:`
	`  My(int a0, double x0);`
	`  ~My();`
	`  void print();`
	`  int calc(double y);`
	`};`

В силу такой эквивалентности в Си++ **классы** (`class`) **можно наследовать от структур** (`struct`) **и наоборот**.

Термин «*класс*» относится к *объектно-ориентированному программированию* (ООП). «***Класс***» – это ***множество объектов, устроенных одинаково***. В языке Си++ для описания типа этих объектов (то есть внутреннего устройства «*класса*») можно использовать ключевые слова `class` или `struct`.

В практике программирования на Си++ для описания типов объектов чаще стараются использовать классы (`class`), а не структуры (`struct`), показывая тем самым, что программист действует в парадигме ООП. Структуры (`struct`) в основном используются для описания *абстрактных типов данных* (АТД) с открытыми (незащищенными) полями.

Обычно термин «*класс*» используется для обозначения *типа объектов*, независимо от того каким ключевым словом языка Си++ (`class` или `struct`) он описан.

9.5. Защита при наследовании

В языке Си++ имеется два вида наследования: открытое (`public`) и закрытое (`private`). Модель защиты (`private` / `public`) указывается в заголовке класса-потомка после двоеточия и перед названием класса-предка.

При любом из видов наследования объектам класса-потомка **доступны только открытые поля и методы** (`public`) класса-предка. Причем при открытом наследовании (`public`) они доступны **отовсюду**, при закрытом наследовании (`private`) они доступны **только из методов класса-потомка**.

То есть при открытом наследовании (`public`) поля и методы класса-предка доступны для класса-потомка так, как если бы у класса-потомка они были описаны в разделе `public`. А при закрытом наследовании (`private`) поля и методы класса-предка доступны для класса-потомка так, как если бы у класса-потомка они были описаны в разделе `private`.

Закрытые (`private`) поля и методы класса-предка для класса-потомка **не доступны**.

Пример наследования классов и схожих с ними классов без наследования:

Класс-предок и классы-потомки	Схожие классы без наследования

class **Parent** { // *класс-предок*
  int n;
public:
  double x;
  void output(int m);
private:
  void calc(double y);
};
class **ChildPubl** : public **Parent** {
  ... // *закрытые поля потомка*
public:
  ... // *открытые поля и методы потомка*
  void some();
private:
  ... // *закрытые методы потомка*
};
 	
class **ChildPriv** : private **Parent** {
  ... // *закрытые поля потомка*
public:
  ... // *открытые поля и методы потомка*
  void some();
private:
  ... // *закрытые методы потомка*
};
	class **Parent** {
	  int n;
	public:
	  double x;
	  void output(int m);
	private:
	  void calc(double y);
	};
	class **ChildPubl** {
	  ... // *закрытые поля*
	public:
	  double x;
	  void output(int m);
	  ... // *открытые поля и методы*
	  void some();
	private:
	  ... // *закрытые методы*
	};
	class **ChildPriv** {
	  ... // *закрытые поля*
	  double x;
	public:
	  ... // *открытые поля и методы*
	  void some();
	private:
	  void output(int m);
	  ... // *закрытые методы*
	};

Класс-предок и классы-потомки	Схожие классы без наследования

`void ChildPubl::some() {`
`  n = 123;  // не доступно`
`  x = 4.56;  // доступно`
`  output(7);  // доступно`
`  calc(0.89); // не доступно`
`}`
`void ChildPriv::some() {`
`  n = 123;  // не доступно`
`  x = 4.56;  // доступно`
`  output(7);  // доступно`
`  calc(0.89); // не доступно`
`}`
`int main(void) {`
`  ChildPubl a;`
`  a.n = 123;  // не доступно`
`  a.x = 4.56;  // доступно`
`  a.output(7);  // доступно`
`  a.calc(0.89); // не доступно`
`  ...`
`  ChildPriv b;`
`  b.n = 123;  // не доступно`
`  b.x = 4.56;  // не доступно`
`  b.output(7);  // не доступно`
`  b.calc(0.89); // не доступно`
`  ...`
`}`
	`void ChildPubl::some() {`
	`  n = 123;  // не существует!`
	`  x = 4.56;  // доступно`
	`  output(7);  // доступно`
	`  calc(0.89); // не существует!`
	`}`
	`void ChildPriv::some() {`
	`  n = 123;  // не существует!`
	`  x = 4.56;  // доступно`
	`  output(7);  // доступно`
	`  calc(0.89); // не существует!`
	`}`
	`int main(void) {`
	`  ChildPubl a;`
	`  a.n = 123;  // не существует!`
	`  a.x = 4.56;  // доступно`
	`  a.output(7);  // доступно`
	`  a.calc(0.89); // не существует!`
	`  ...`
	`  ChildPriv b;`
	`  b.n = 123;  // не существует!`
	`  b.x = 4.56;  // не доступно`
	`  b.output(7);  // не доступно`
	`  b.calc(0.89); // не существует!`
	`  ...`
	`}`

9.6. Наследование методов

В приведенном примере при наследовании для объекта-потомка `a` (который является экземпляром класса-потомка **`ChildPubl`**) можно вызвать метод `output`, описанный в классе-предка **`Parent`**. Причем этот метод будет работать так же, как и для объектов класса-предка, ничего не зная о том, что его вызвали для объекта-потомка.

В ООП *потомок* умеет отвечать **на все виды сообщений** (то есть **выполнять все доступные методы**), предусмотренные для его *предков*.

Здесь действует описанный ранее **полиморфизм адресов** в применении к указателю `this`. В методе `output` указатель `this` имеет тип «**`Parent *`**» – указатель на объект класса-предка **`Parent`**. При вызове `output` для объекта класса-потомка **`ChildPubl`** указатель `this` должен иметь тип «**`ChildPubl *`**», но он **неявно** преобразовывается к типу «**`Parent *`**» по закону полиморфизма.

С точки зрения **логики проектирования**: *объект класса-потомка* – это **частный случай** *объекта класса-предка* (то есть *объект класса-потомка* **является также** и *объектом класса-предка*).

С точки зрения **средств реализации**: *объект класса-потомка* **содержит в себе** *объект класса-предка* в качестве своей **части**.

9.7. Наследование по умолчанию

Аналогично классам (`class`) правила наследования действуют и для структур (`struct`). При этом классы (`class`) и структуры (`struct`) различаются только видом наследования **по умолчанию:**

Наследование по умолчанию	Эквивалентное наследование

`struct Person {`
`  ...`
`};`
`struct Student : Person {`
`  ...`
`};`
	`struct Person {`
	`  ...`
	`};`
	`struct Student : public Person {`
	`  ...`
	`};`

`class Person {`
`  ...`
`};`
`class Student : Person {`
`  ...`
`};`
	`class Person {`
	`  ...`
	`};`
	`class Student : private Person {`
	`  ...`
	`};`

9.8. Открытое наследование

На практике закрытое наследование (`private`) используется **крайне редко**. Поэтому при описании наследуемых классов **почти всегда** указывают ключевое слово `public`.

Пример наследования для классов `Person` и `Student`:

```
class Person {
    ...     // закрытые поля (не доступны потомкам)
public:
    ...     // открытые поля и методы (доступны потомками)
private:
    ...     // закрытые методы (не доступны потомками)
};
class Student : public Person {     // Student унаследован от Person
    ...     // закрытые поля потомка
public:
    ...     // открытые поля и методы потомка
private:
    ...     // закрытые методы потомка
};
```

При этом класс-потомок `Student` наследует **только открытые** (`public`) **поля и методы** класса-предка `Person` (унаследованные поля и методы повторно описывать в классе `Student` не нужно). **Закрытые** (`private`) поля и методы класса-предка `Person` для класса-потомка `Student` **не доступны**.

9.9. Защищенные поля и методы (protected)

Часто возникает необходимость в таких полях и методах класса, которые предназначены исключительно для его потомков. Для таких случаев имеется ***защищенный*** режим наследования с ключевым словом protected.

Поля и методы, помеченные словом protected, доступны:

- методам самого класса,
- дружественным функциям,
- методам классов, которые являются непосредственными потомками, а также потомками потомков только при открытом наследовании (public).

Во всех остальных местах программы доступ к ним запрещен.

Защищенные поля и методы (protected) нельзя в полной мере отнести к закрытой части класса, они считаются **особым видом открытой части класса**. Детали закрытой части класса (private) можно безболезненно изменять, исправляя при этом только методы самого класса и дружественные функции (перечисленные в заголовке класса). А поля и методы protected могут использоваться там, где их появление невозможно предсказать заранее.

Поэтому поля и методы protected должны **обязательно *документироваться*** (например, **комментариями в тексте программы**), а к их изменению необходимо подходить также осторожно, как к изменениям в открытой части класса (public).

Например, создадим класс **Parent** с защищенными (protected) полем ch и методом erase, описание которых необходимо обязательно документировать. С помощью открытого наследования (public) от класса **Parent** создадим новый класс-потомок **Child**. Тогда защищенные (protected) поле ch и метод erase объекта-потомка доступны только из его методов (например, из метода some), но не доступны из других частей программы:

```
class Parent {
    int n;
protected:
    char ch;              // обязательно документировать поле!
    void erase(int k);    // обязательно документировать метод!
public:
    double x;
    ...
};
class Child : public Parent {
    ...
public:
    void some();
    ...
};
void Child::some() {
    n = 123;    // не доступно
    ch = 'w';   // доступно
    erase(5);   // доступно
    x = 4.56;   // доступно
}
int main(void) {
    Child a;
    a.n = 123;    // не доступно
    a.ch = 'w';   // не доступно
    a.erase(5);   // не доступно
    a.x = 4.56;   // доступно
    ...
}
```

9.10. Конструктор и деструктор потомка

В момент создания объекта-потомка создается и объект-предок, а при уничтожении объекта-потомка исчезает и объект-предок. Такой объект-предок не является отдельным объектом, а создается как **часть** объекта-потомка и становится его **другой ипостасью**.

При этом конструктор объекта-предка должен выполниться **перед началом** работы конструктора объекта-потомка. А деструктор объекта-предка должен выполниться **после окончания** работы деструктора объекта-потомка.

Деструктор.

Для корректного уничтожения объектов-потомков компилятор автоматически вставляет вызов деструктора класса-предка в самый конец кода деструктора класса-потомка.

Конструктор.

Для корректного создания объекта-потомка компилятор должен сначала провести инициализацию объекта-предка с помощью его конструктора, а затем использовать конструктор класса-потомка для инициализации оставшейся части объекта-потомка. Но для этого конструктор объекта-потомка может требовать **входных параметров**.

В этом случае поступают так же, как при инициализации членов класса в его конструкторе. После описания заголовка конструктора класса-потомка (после двоеточия) помещают список инициализаторов, **начинающийся с инициализатора класса-предка**, в котором указывают значения параметров конструктора класса-предка. Например:

```
class Parent {  // Класс-предок.
    ...
public:
    Parent(int x, int y);  // Конструктор класса-предка с параметрами.
    ...
};
```

```
class Child : public Parent {  // Класс-потомок.
    int k;
    ...
public:
    Child();  // Конструктор класса-потомка.
    ...
};
void Child::Child() : Parent(12, 34), k(5) {  // Заголовок конструктора
    ...                       // класса-потомка со списком инициализаторов.
}
```