18. ПРОИЗВОДНЫЕ КЛАССЫ

18.1. Построение производного класса

Рассмотрим класс с конструктором и деструктором:

```
class Base{ int
*bmember;
public:
Base(int arg = 0){ bmrmber
= new int(arg);
}
~Base(){   delete
bmember;
}
};
```

Предположим, что нам нужно изменить этот класс так, чтобы объект такого типа содержал не один, а два указателя. Вместо того, чтобы изменять класс Base, можно поступить иначе: ничего не меняя в Base, построить на его основе новый класс Derived:

```
class Derived: public Base{
int *dmember; public:
Derived(int arg){ dmember
= new int(arg);
}
~Derived(){   delete
dmember;
}
};
```

Запись вида **class Derived: public Base** говорит о том, что класс Derived является таким заново создаваемым классом, который построен на основе класса Base. При этом класс Derived **наследует** все свойства класса Base. Говорят, что Derived является классом, **производным** от класса Base, а класс Base является **базовым** классом для Derived.

Если в программе будет создан объект типа Derived, то он будет содержать два указателя на две области динамической памяти – bmember, как подобъект типа Base, и dmember. Процесс создания объекта типа

Derived будет проходить в два этапа: сначала будет создан «подобъект» типа Base, причём это сделает конструктор класса Base.
Затем будет выполнен конструктор класса Derived. Вызов деструкторов осуществляется в обратном порядке. Поскольку конструктор класса Base может требовать наличия одного аргумента при обращении к нему, то этот аргумент необходимо передать. Чтобы передать список аргументов конструктору базового класса, этот список должен быть помещён в определении конструктора производного класса, подобно тому, как это делалось при инициализации данных абстрактного типа, являющихся членами некоторого класса:

```
Derived::Derived(int arg): Base (arg){
dmember = new int(arg); }
```

Если конструктор базового класса не имеет аргументов или использует аргументы по умолчанию, помещать пустой список в конструктор производного типа не надо.

18.2. Защищенные члены класса

Для регулирования уровня доступа к членам классов используются служебные слова public и private. Для этих же целей введено ключевое слово **protected** (защищенный).

Если класс A не служит базовым ни для какого другого класса, то его защищенные члены ничем не отличаются от личных – доступ к ним имеют только функции-члены данного класса и дружественные этому классу функции. Если же класс B является производным от A, то пользователи как класса A, так и B по-прежнему не имеют доступа к защищенным членам, но такой доступ могут иметь функции-члены класса B и функции, привилегированные в B:

```
class Base{ private:
int privatem; protected:
int protectedm;
};
class Derived: public Base{
memberF(){
cout << privatem;              // Ошибка!
cout << protectedm;            // Верно.
}
};
```

```
void main( ){
Base b;
cout << b.protectedm;// Ошибка!
Derived d;
cout <<                    // Ошибка.
d.protectedm; }
```

18.3. Управление уровнем доступа к членам класса

В предыдущих примерах базовый класс являлся общим базовым классом для производного класса:

```
class Derived: public Base{…};
```

Это означает, что уровень доступа к членам класса Base для функций-членов класса Derived и просто пользователей класса Derived остался неизменным: личные члены класса Base не доступны в классе Derived, общие и защищенные члены класса Base остались соответственно общими и защищенными в Derived. Если не указать, что базовый класс является общим, то по умолчанию он будет личным:

```
class Derived: Base{…};          // Base – личный базовый класс.
```

Если базовый класс является личным базовым классом, то его личные члены по-прежнему недоступны ни в производном классе, ни для пользователя производного класса, а защищенные и общие члены базового класса становятся личными членами производного класса.

Базовый класс не может быть защищенным базовым классом.

Если базовый класс является личным, то для некоторых его членов в производном классе можно восстановить уровень доступа базового класса. Для этого их полное имя приводится в соответствующей части определения класса: class Base{ private: int privm; protected: int protm; public: int pubm;

```
};
class Derived: Base{// Личный базовый класс.
public:
Base::pubm;     // Теперь pubm – общий член класса Derived;
Base::protm;    // ошибка – изменение уровня доступа.
protected:
Base::protm;    // Теперь protm – защищенный член класса Derived;
Base::pubm;     // ошибка – изменение уровня доступа.
```

Структуры могут использоваться подобно классам, но с одной особенностью: если производным классом является структура, то ее

базовый класс всегда является общим базовым классом. То есть объявление вида struct B: A{…};

эквивалентно class B:

public A{…};

Если же производный класс строится на основе структуры, все происходит точно так же, как и при использовании в качестве базового обычного класса. Таким образом, если и базовым, и производным классами являются структуры, то запись вида struct B: A{…};

эквивалентна class B: public A

{public: …};

18.4. Последовательность вызова конструктора и деструктора при построении производного класса на основе одного базового

Объект производного класса в качестве данных-членов класса, может содержать объекты абстрактных типов. class string{. . . public: string (char*); ~string();

…

};

class Base{…

public: Base(int);

~Base();

. . .

};

class Derived: public Base {

Base b; string s; public:

Derived(char*, int);

~Derived();

. . .

};

Перед обращением к собственно конструктору класса Derived необходимо создать, во-первых, подобъект типа Base, во-вторых, члены b и s. Поскольку для их создания нужно обратиться к конструкторам соответствующих классов, мы должны им всем передать необходимые списки аргументов:

Derived::Derived (char *st, int len): Base(len), b(len + 1), s(str){…}

В этом случае при создании объекта типа Derived сначала будет создан подобъект типа Base. При этом будет вызван конструктор Base::Base() с аргументом len. Затем будут созданы объекты b и s в том порядке, в котором они указаны в определении класса Derived. После этого будет выполнен конструктор Derived::Derived(). Деструкторы будут вызваны в обратном порядке.

18.5. Преобразования типов

Объект производного типа может рассматриваться как объект его базового типа. Обратное неверно. (Кошка есть млекопитающее, но не любое млекопитающее есть кошка.) Компилятор может неявно выполнить преобразование объекта производного типа к объекту типа базового:

```
class Base{…};
class Der: public Base{…};
Der derived;
Base b = derived;
```

Обратное преобразование – Base к Der – должно быть определено программистом: Der tmp = b; // Ошибка, если для Der
// не определён конструктор Der(Base).

Значительно чаще, чем преобразование типов, используется преобразование указателей на эти типы. Существует два типа преобразования указателей – явное и неявное. Явное преобразование будет выполнено всегда, неявное – только в определённых случаях.

Если базовый класс является общим (public) базовым классом, т. е. мы имеем дело с отношением

вида class Base{…}; class Der:
public Base{…};

то принципы преобразования очень просты:

- неявно может быть выполнено преобразование указателя типа Der* к указателю типа Base*;
- обратное преобразование обязательно должно быть явным.

Другими словам, при обращении через указатели объект производного типа может рассматриваться как объект базового типа. Обратное утверждение неверно: Der derived;

```
Base *bp = &derived;     // Неявное преобразование. Der
*dp1 = bp;               // Ошибка.
Der *dp2 = (Der*) bp;    // Явное преобразование; теперь верно.
```

То, что производный класс в некотором смысле может рассматриваться как его базовый, оказывает влияние на выбор нужной версии перегруженной функции. Сложность возникает, если для выбора одного из вариантов нужно выполнить неявное преобразование типов.

Правила здесь таковы.

Если нет точного соответствия списков формальных и фактических параметров, то наивысший приоритет среди выполняемых преобразований имеют преобразования производного типа к базовому. Это относится как к самому типу, так и к указателю на него. Только в том случае, если это невозможно, компилятор пробует выполнить другие преобразования (например, стандартные преобразования указателей). Пример: class Base{

```
…
};
class Der: public Base{
…
}; func(Base*);
func(void*);
…
Der *dp;
float *fp; func(dp);                    // Вызов
func (Base*). func(fp);                 // Вызов
func (void*).
```

Если в программе используются несколько уровней производных классов, то при выполнении неявных преобразований типа указателя ищется класс «ближайшего» уровня:

```
class Base{…}; class A:
public Base{…}; class B:
public A{…};
func(Base*); func(A*);
…
B *db;
func(db);                    // Вызов функции func (A*).
```