**Наследование**

Наследование – один из основных принципов ООП, который позволяет производным (дочерним) классам наследовать свойства и состояния базовых (родительских) классов. Это позволяет создавать иерархию типов, упростив состав производных классов.

При описании класса в его заголовке описываются все классы, являющиеся для него базовыми. Возможность обращения к компонентам (элементам) базового класса описывается с помощью ключей доступа: private, protected, public. Общий формат объявления иерархии классов следующий:

class имя\_класса : [private| protected | public] имя\_базового\_класса

{

тело класса

};

Если базовых классов несколько, они перечисляются через запятую, например,

class A{ };

class B{ };

class C{ };

class D :A, protected B, public C

{ };

Для классов по умолчанию принят ключ доступа private, для структур и объединений – public.

Ключи доступа сведены в таблицу

|  |  |  |  |
| --- | --- | --- | --- |
| Ключ доступа | Спецификатор в базовом классе | Доступ в производном классе | Примечание |
| private | private  protected  public | private  private  private | все private |
| protected | private  protected  public | private  protected  protected | все protected,  кроме private |
| public | private  protected  public | private  protected  public | не изменяется |

Доступ (заметьте, именно доступ, а не наследование) к компонентам базового класса со спецификатором private не возможен. Обращение к ним можно осуществить только через методы базового класса.

**Простое наследование**

Наследование считается простым, если у класса один базовый класс. Типовой пример простого наследования показан ниже.

#include<iostream>

using namespace std;

class Base

{

int base;

public:

Base(){};

Base(int b) :base(b){};

void Show()

{

cout << " Base: " << base << endl;

}

};

class Derived :public Base

{

int derived;

public:

Derived():Base(){};

Derived(int b, int d): Base(b), derived(d){};

void Show()

{

cout << " Derived: " << derived << endl;

}

};

int main()

{

Base bas(100);

Derived der(200,300);

bas.Show();

der.Show();

system("pause");

return 0;

}

В объявлении class Derived :public Base видно, что для класса Derived базовым классом является класс Base с ключом доступа public. В дальнейшем такой производный класс мы будем именовать классом с обобществленным базовым классом. Это будет влиять на возможность преобразований в пределах иерархии классов.

Результатом работы (вызова) программы будет:

Base: 100

Derived: 300

Здесь отработали функции Show() соответствующих классов. При необходимости объект производного класса может обратиться и к функции базового класса, то есть унаследовать ее, например, следующим образом der.Base::Show();.

Посмотрите и оцените результат работы, он будет отличаться от предыдущего вызова.

Несколько слов о функции (свойстве) Show(), она определена в обоих классах. Это говорит о том, что свойство базового класса переопределено (уточнено, конкретизировано, детализировано) в производном.

Теперь о ключах доступа, как уже было отмечен ранее, класс Derived объявлен с ключом public, то есть имеет обобществленный базовый класс. Прямой доступ к компонентам базового класса возможен только, если они расположены в областях protected и public. Если изменить ключ доступа на private в описании базового класса или вовсе не указать ключ, например, class Derived : Base, то производный класс называется классом с приватизированным базовым классом. Это означает, что все компоненты базового класса являются не доступными для объектов производного типа. При попытке вызвать функцию базового класса der.Base::Show(); компилятор выдаст сообщения следующего вида:

*error C2247: нет доступа к "Base::Show", поскольку "Derived" использует "private" для наследования из "Base"c:\users\documents\visualstudio2012\projects\test\_inherit\test\_inherit\test\_inherit.cpp.*

Это не означает, что доступа к компонентам закрытого базового класса нет совсем. Обращение возможно внутри методов производного класса к полям и методам базового класса, расположенных в областях protected и public. Это демонстрируется в модифицированном методе Show() производного класса:

void Show()

{

cout << " Derived: " << derived << endl;

Base::Show();

}

На помощь приходит операция разрешения области видимости Base::Show();.

**Общие правила наследования**

Следует заметить, что не все компоненты базового класса наследуются в производных типах, к таковым относятся:

- конструкторы;

-деструкторы;

-операция присваивания.

Отдельный вопрос относительно функций дружественных базовому классу. В стандарте языка С++ сказано, что такие функции не наследуются. И это понятно, поскольку дружественные функции не являются составными функциями какого-либо класса (они внешние по отношению к классу) и в процессе наследования не участвуют. Однако объект производного типа может воспользоваться услугами функции, дружественной по отношению к базовому типу. Объясняется это просто: по умолчанию объект (указатель или ссылка) производного типа приводится к типу объекта (указателя, ссылки) базового типа. Преобразования подобного рода относятся к стандартным и выполняются компилятором по умолчанию. Еще одна особенность преобразований состоит в том, что производный класс должен иметь обобществленный базовый класс, то есть ключ доступа должен быть public. Если наследование приватное, то компилятор подобных преобразований делать не будет и выдаст сообщение о невозможности преобразования. Пример использования дружественных функций несколько позже.

Как было отмечено, конструкторы не наследуются. Это объясняется тем, что конструкторы базового класса обязаны выделять память под свои собственные компоненты и по возможности инициализировать их. О существовании компонент производных типов они знать не обязаны. Однако не следует забывать о том, что при создании объекта производного типа, в первую очередь должен активизироваться соответствующий ему конструктор базового класса. Проверит это легко, если в тело каждого из конструкторов вставить соответствующее ему сообщение, например, для рассмотренного ранее примера:

Base()

{ cout << “ Конструктор без параметров типа Base “ << endl; }

Base(int b) :base(b)

{ cout << “ Конструктор с параметром типа Base “ << endl; }

и

Derived():Base()

{ cout << “ Конструктор без параметров типа Derived “ << endl; }

Derived(int b, int d): Base(b), derived(d)

{ cout << “ Конструктор с параметрами типа Derived “ << endl; }

Если создать объект типа Derived, например, Derived der\_1; или Derived der\_2(10,30), то порядок активизации конструкторов появится на экране монитора.

Кроме активизации конструкторов базовых классов, активизируются конструкторы объектных полей классов, если таковые имеются. Если базовых классов несколько, порядок их активизации согласно их списку в заголовке производного класса.

Еще одна особенность, касающаяся конструкторов производных классов, обязательное наличие списка инициализации, в котором указывается конструктор базового класса с необходимым количеством параметров, например, Derived():Base(){}; и Derived(int b, int d): Base(b), derived(d){};.

В первом случае будет активизирован конструктор базового класса без параметров, во втором, конструктор с параметром. Невыполнение этого требования приведет к ошибке.

Следующий не наследуемый компонент базового класса – деструктор. Данный факт также объясняется просто, освобождение памяти из-под объекта в соответствии с его фактическим типом. Если активизируется деструктор базового типа, то освободится память занимаемая компонентами базового типа, а поля производного типа останутся мусором в памяти. Здесь также необходимо сделать уточнения, в частности, порядок активизации деструкторов обратный порядку активизации конструкторов. Проверить это тоже не сложно, добавив в пример классов деструкторы с соответствующими сообщениями.

В отношении запрета на наследование операции присваивания. Этот факт связан с тем, что количество полей производного типа, как правило, превышает количество полей базового, поскольку объект производного типа наследует все поля базового и имеет свои собственные. Для большинства практических случаев это классический пример. Если бы операция присваивания наследовалась в производном классе, то присваивание касалось бы только полей базового класса, при этом свои собственные останутся без внимания. Не следует забывать о том, что операция присваивания генерируется компилятором автоматически во всех объявленных классах, если программист не предпринял никаких действий. Операцию следует переопределять в случае, если есть в этом необходимость.

**Доступ к закрытым полям базового класса**

Непосредственный доступ к полям базового класса из методов производного класса запрещен, и любая попытка сделать это будет пресечена компилятором. Рассмотрим пример модифицированной функции Show() производного класса:

void Show()

{

cout << " Base: " << base << endl;

cout << " Derived: " << derived << endl;

}

Обращение cout << " Base: " << base << endl; приведет к выводу сообщений IntelliSense: член "Base::base" (объявлено в строке 6) недоступно c:\Users\Documents\Visual Studio 2012\Projects\test\_inh\test\_inh\test\_inh.cpp и C2248: Base::base: невозможно обратиться к private член, объявленному в классе "Base"c:\users\игорь\documents\visual studio 2012\projects\test\_inh\test\_inh\test\_inh.cpp 25 1 test\_inh.

Однако обращение все-таки возможно, например, с помощью метода базового класса, расположенного в области public и возвращающего значение закрытого поля. Определим в теле базового класса функцию следующего вида:

int Get()

{

return base;

}

Тогда обращение в методе производного класса можно осуществить через данный метод следующим образом:

void Show()

{

cout << " Base: " << Get() << endl;

cout << " Derived: " << derived << endl;

}

Несложно догадаться, что с помощью методов базового класс можно не только получать значения закрытых полей базового, но и менять при необходимости.

**Защищенные поля базового класса (protected)**

До сих пор, то есть, до появления наследования в иерархии классов, области private и protected любого класса можно было рассматривать как равнозначные. Доступ к данным полям непосредственно через объект класса, указатель или ссылку недопустим. В условиях наследования прямое обращение из методов производного типа к полям базового допускается в том случае, если они расположены в области protected. В этом и проявляется отличие полей private от полей protected. Обратимся еще раз к классам и их функциям, базовый класс:

class Base

{

protected: // изменения

int base;

public:

Base(){};

Base(int b) :base(b){};

void Show()

{

cout << " Base: " << base << endl;

}

};

функция Show() производного класса:

void Show()

{

cout << " Base: " << base << endl;

cout << " Derived: " << derived << endl;

}

Заметьте, что в первой строке обращении к полю уже непосредственное.

**Множественное наследование**

Множественное наследование предполагает, что у любого класса может быть не один, а несколько базовых классов, имена которых, присутствуют в заголовке производного класса. Общий формат множественного наследования следующий:

class Имя\_производного\_класса: [private|protected|public] Имя\_базового\_класса\_1,

[private|protected|public] Имя\_базового\_класса\_2,

….

[private|protected|public] Имя\_базового\_класса\_N

{

// тело класса

}

Количество базовых классов ни чем не ограничено ( в разумных пределах), перед именем каждого обязательно необходимо указать свой ключ доступа. Еще важный момент связан с порядком следования базовых классов в описании производного класса. От этого зависит последовательность активизации конструкторов базовых классов при создании объекта производного типа. Во многих практических задачах последовательность активизации конструкторов может иметь определенное значение, в частности, от этого может зависеть объем памяти, необходимой под хранение объекта производного типа.

Любой из производных классов может в свою очередь быть базовым для одного или нескольких других классов. Иерархия классов может быть достаточно сложной, примером тому могут служить классы обслуживающие потоки, общим базовым классом (или как его часто называют в литературе дедушкой потоковых классов) является класс io (input, output). Классы, расположенные в начале иерархии, содержат в себе свойства наиболее общие для всех потомков которые, в свою очередь, уточняют, переопределяют или дополняют эти свойства в соответствии со своим типом. Это замечательное свойство (наследование) позволяет существенно уменьшить объем программного кода классов, уточняя отдельные моменты в производных классах. Некоторые свойства наследуются без изменений, например, для класса «Млекопитающие» и всех его производных классов, способ вскармливать детенышей молоком, скорее всего, переопределять не надо.

Начнем рассмотрение множественного наследования с простого примера.

#include<iostream>

using namespace std;

class Base

{

protected:

int base;

public:

Base(){};

Base(int b) :base(b){};

void Show()

{

cout << " Base: " << base << endl;

}

};

class Derived\_1 :public Base

{

int derived\_1;

public:

Derived\_1():Base(){};

Derived\_1(int b, int d): Base(b), derived\_1(d){};

void Show()

{

cout << " Derived\_1: " << derived\_1 << endl;

}

};

class Derived\_2 :public Base

{

int derived\_2;

public:

Derived\_2():Base(){};

Derived\_2(int b, int d): Base(b), derived\_2(d){};

void Show()

{

cout << " Derived\_2: " << derived\_2 << endl;

}

};

Общая структура классов графически выглядит следующим образом

Base

Derived\_2

Derived\_1

Производные классы практически одинаковые, за исключением различных имен составляющих полей. Оба класса в полном объеме наследуют все свойства базового класса.

Следующие объявления и вызовы

Base bas(10);

Derived\_1 der\_1(100,200);

erived\_2 der\_2(400,500);

bas.Show();

der\_1.Show();

der\_2.Show();

приведут к следующей реакции программы:

Base: 10

Derived\_1: 200

Derived\_2: 500

Каждый из объектов воспользовался услугами собственной функции Show(), и выведенные значения соответствуют исходным значениям, определенных при объявлении объектов. При необходимости объект каждого производного класса может унаследовать функцию базового базового класса, например, следующим образом:

der\_1.Base::Show();

der\_2.Base::Show();

Результат посмотреть и оценить самостоятельно.

Иерархии подобного вида обычно не вызывают больших проблем, ошибки их определения и использования, скорее всего, можно отнести к случайным. Однако при наличии достаточно развитой иерархии, поведение программы, в частности, ее результаты, могут оказаться неожиданными.

Для решения определённых проблем в наследовании в C++11 добавили два специальных идентификатора: override и final. Обратите внимание, эти идентификаторы (или «модификаторы») не являются ключевыми словами - это обычные идентификаторы, которые имеют особое значение в определённых контекстах. Обратимся вначале к final. В практическом программировании случаи, когда необходимо запретить переопределение виртуальной функции или наследование определённого класса. **Модификатор final** используется именно для этого. Если пользователь пытается переопределить метод или наследовать класс с модификатором final, то компилятор выдаст ошибку. Позиция модификатора видна из следующего примера:

class Derived\_1 final :public Base и

class Derived\_2 final :public Base

Это говорит о том, что классы Derived\_1 и Derived\_2 являются конечными, или по аналогии с деревьями – листьями, и попытка объявить их как базовые классы для возможных производных от них классах, будет предотвращена компилятором. Например, объявление вида:

class Test :public Derived\_1

{

};

приведет к немедленной реакции компилятора:

IntelliSense: тип класса final не может использоваться в качестве базового классаc:\User\Documents\Visual Studio 2012\Projects\test\_inh\test\_inh\test\_inh.cpp test\_inh

и

error C3246: "Test": нельзя наследовать из "Derived\_1", так как было объявление в качестве "final"c:\users\documents\visual studio 2012\projects\test\_inh\test\_inh\test\_inh.cp p

Другое использование модификатора final, а также модификатор override рассмотрим при объявлении виртуальных функций.

Следующим примером множественного наследования может служить ситуация, когда у производного класса два или более базовых классов:

#include<iostream>

using namespace std;

class Base\_1

{

protected:

int base\_1;

public:

Base\_1(int b\_1): base\_1(b\_1){};

void Show()

{

cout<< " " << base\_1 << endl;

}

};

class Base\_2

{

protected:

int base\_2;

public:

Base\_2(int b\_2): base\_2(b\_2){};

void Show()

{

cout<< " " << base\_2 << endl;

}

};

class Derived :public Base\_1, public Base\_2

{

public:

Derived(int b\_1, int b\_2): Base\_1(b\_1), Base\_2(b\_2){};

void Show()

{

cout << " Base\_1: " << base\_1 << endl;

cout << " Base\_2: " << base\_2 << endl;

}

};

int main()

{

Derived der(100, 200);

der.Show();

system("pause");

return 0;

}

Структура иерархии представлена ниже.

Base\_2

Base\_1

Derived

Отметим особенности иерархии. Для простоты базовые классы определены по аналогии друг с другом, за исключением того, что имена полей различны. Для упрощения доступа к полям базовых классов, они были помещены в области protected. Имена базовых классов в заголовке производного класса помещены в последовательности Base\_1, Base\_2. Как было уже сказано ранее, эта последовательность влияет на порядок активизации конструкторов базовых классов при объявлении объекта производного типа. Для того, чтобы в этом убедиться добавьте в конструкторы базовых классов вывод в поток информацию, идентифицирующую их срабатывание. Для полноты картины хорошо добавить вывод и в конструктор производного класса. Как уже было отмечено, конструкторы базовых классов не наследуются производными классами, тем не менее, они активизируются и связано это с тем, что они гарантированно выделят необходимый объем памяти под поля базового класса и передадут эту информацию в производный класс.

Выполните эту программу, посмотрите результат.

Подобная иерархия не также не вызывает больших проблем, за исключением того, если в базовых классах определены одноименные компоненты, например, для данного случая, это функции Show(). Вопрос решается путем применения оператора разрешении области видимости после, например,

der.Base\_1::Show();

der.Base\_2::Show();

Такой же подход применяется в случае, если осуществляется попытка обращения к одноименным полям базовых классов из метода производного класса.

Следующая проблема, связанная с множественным наследованием – наследование свойств базового класса несколькими путями. Рассмотрим пример следующей иерархии классов.

Base

Derived\_1

Derived\_2

Программа, соответствующая этой иерархии представлена ниже

class Base

{

protected:

int base;

public:

Base(){};

Base(int b):base(b){};

void Out()

{

cout << " Base: " << base << endl;

}

};

class Derived\_1 :public Base

{

char derived\_1;

public:

Derived\_1():Base(){};

Derived\_1(int b, char d):Base(b),derived\_1(d){};

void Out()

{

cout << " Derived\_1: " << derived\_1 << endl;

}

};

class Derived\_2 :public Base, public Derived\_1

{

double derived\_2;

public

Derived\_2():Base(), Derived\_1(){};

Derived\_2(int b, char d\_1, double d\_2): Base(b), Derived\_1(b,d\_1),derived\_2(d\_2){};

void Out()

{

cout << " Base: " << base << endl;

cout << " Derived\_2: " << derived\_2 << endl;

}

};

При попытке выполнить эту программу, компилятор выдаст массу сообщений вида

IntelliSense: "Derived\_2::base" не является однозначным c:\Users\Documents\Visual Studio 2012\Projects\test\_mult\test\_mult\test\_mult.cpp

Это говорит о том, что все компоненты базового класса наследуются в производном Derived\_2 двумя путями: через непосредственную связь и через класс Derived\_1. В результате чего в объекте класса Derived\_2 появятся две унаследованные копии поля int base; класса Base. Проблема решается путем объявления класса Base как виртуального базового класса в описании заголовков производных от него классов следующим образом:

class Derived\_1 :virtual public Base и

class Derived\_2 :virtual public Base, public Derived\_1

В этом случае компилятор оставляет один возможный путь наследования, закрывая остальные. Ошибки исчезнут, классы можно использовать в полном объеме.

**Полиморфизм**

Следующее важное свойство ООП – полиморфизм. Это свойство говорит о том, что каждый объект должен реагировать на любое событие (сообщение) сообразно своему типу. Предположим, что определен базовый класс Device (устройство), являющийся родителем для классов Mouse (мышь) и Keyboard (клавиатура). В классе Device должна быть объявлен функция Input, которая обеспечивает ввод информации с устройства ввода, причем реализация данной функции может и не быть. Понятно, что операции ввода от мыши и от клавиатуры имеют свои особенности, связанные со спецификой устройств. То есть их реализации в классах Mouse и Keyboard должны отличаться в соответствии с типом конкретного устройства. На практике часто встречаются ситуации, когда на момент написания программы за ранее не известен тип фактического объекта, например, как в приведенном примере с устройствами ввода. Сложно заранее предусмотреть все ситуации, в которых пользователь будет воздействовать на мышь или клавиатуру.

На помощь приходит полиморфизм, который позволяет программисту не задумываться над типом объекта, а работать с единственным указателем на класс Device. Рассмотрим упрощенный вариант реализации эти классов.

class Device

{

public:

Device(){};

void Input(unsigned int=0)

{

cout << " Class Device " << endl;

};

};

class Mouse :public Device

{

unsigned int mouse;

public:

Mouse():Device(){};

void Input(unsigned int m)

{

mouse = m;

cout << " Class Mouse " << endl;

}

};

class Keyboard :public Device

{

unsigned int keyboard;

public:

Keyboard():Device(){};

void Input(unsigned int k)

{

keyboard = k;

cout << " Class Keyboard " << endl;

}

};

Следующие объявления и вызовы приведут к тому, что в каждом случае сработают функции соответствующих классов:

Device obj\_Device;

obj\_Device.Input(0);

Mouse obj\_Mouse;

obj\_Mouse.Input(10);

Keyboard obj\_Keyboard;

obj\_Keyboard.Input(20);

Объясняется это просто – при явном объявлении объекта какого-либо класса при активизации функции, обязана отрабатывать функция именно этого класса. При необходимости обратиться к функции базового класса, необходимо указать область видимости базового класса.

Изменим несколько объявления и вызовы:

Device obj\_Device;

Device \*ptr\_Device = &obj\_Device;

ptr\_Device->Input();

Вызов Input() приведет к активизации функции класса Device, что очевидно.

Следующие объявления нуждаются в пояснениях:

Device obj\_Device;

Device \*ptr\_Device = &obj\_Device;

Mouse obj\_Mouse;

ptr\_Device = &obj\_Mouse;

ptr\_Device->Input(10);

В частности, ptr\_Device = &obj\_Mouse;, насколько правомерно подобное присваивание? Указателю на базовый класс присваивается адрес объекта производного типа. Да, это возможно, более того, преобразования указателей на производный тип к указателям на базовый относятся к стандартным и компилятор выполняет их автоматически. Точно таким же образом компилятор преобразует стандартные типы, например char -> int -> double, etc.

В некоторых случаях преобразования нужно делать явным образом, например,

ptr\_Device = (Device \*)&obj\_Mouse;

или

ptr\_Device = static\_cast<Device \*>(&obj\_Mouse);

Оба выражения эквивалентны, первое заимствовано из языка С, второе – новшество в С++, разница их в том, что второе выражение представляет собой контролируемый способ преобразований, по этой причине он предпочтительней.

Далее, вызов ptr\_Device->Input(10);

приведет к активизации функции класса Device, несмотря на то, что указателю присвоен адрес объекта производного типа. По сути, здесь объект производного типа воспользовался услугами функции базового класса, то есть унаследовал ее.

Для осуществления свойства полиморфизма в базовом классе должна присутствовать, хотя бы одна функция, описанная с ключевым словом virtual, например,

virtual void Input(unsigned int=0)

Это виртуальная функция, вызов конкретной реализации функции принимается операционной системой на этапе выполнения программы. Например, теперь вызов

ptr\_Device->Input(10);

приведет к другому результату, чем в предыдущем случае. В этом и состоит проявление полиморфизма, которое следует трактовать как проявлении многих форм одного и того же.

**Виртуальные методы**

В рассмотренном выше коде приведен пример использования виртуальных функций. В одиночных классах, не состоящих в какой-либо родственной связи с другими классами, их наличие не обязательно. Более того, их присутствие приведет к дополнительным действиям компилятора и к необходимости увеличения ресурсов, необходимых для выполнения программы.

Еще один момент, относящийся к рассмотренному примеру, работа с объектами чаще всего производится через указатели на них. Дело в том, что указателю на базовый класс можно присвоить адрес объекта любого из производных от него классов. Кроме того, дальнейшая работа будет связана в большей степени через указатели, либо через ссылки на объекты классов.

В общем случае объявить функцию как виртуальную можно по следующему шаблону:

virtual тип\_результата имя\_функции(список параметров)

{

// тело функции

}

Ключевое слово virtual может идти не только перед типом возвращаемого результата, но и после него. Компилятор об этом знает. В общем случае виртуальной может быть абсолютно любая из составных функций класса за исключением конструктора. В противоположность конструкторам, деструктор может быть виртуальным, а во многих случаях просто обязан быть таковым. Это связанно с корректным освобождением динамической области памяти от объектов иерархии классов.

Еще одна особенность, связанная с виртуальными методами состоит в том, что если в производном классе переопределен виртуальный метод базового класса, то ключевое слово virtual в его заголовке указывать не обязательно, он считается виртуальным по умолчанию.

Общие правила описания и использования виртуальных методов:

1. Если в базовом классе метод определен как виртуальный, метод, метод, определенный в производном классе с тем же именем и с тем же списком параметров, автоматически становится виртуальным, а с отличающимся списком – обычным;

2. Виртуальные методы наследуются, то есть, переопределять их в производных классах требуется только при необходимости задать отличающие действия. Права доступа при переопределении изменить нельзя;

3.Если в производном классе переопределен виртуальный метод, его объекты могут получить доступ к методу базового класса с помощью операции доступа к области видимости;

4. Виртуальный метод не может быть объявлен с модификатором static. В качестве виртуальных методов не могут выступать дружественные функции;

5. Операции new и delete не могут быть объявлены как виртуальные, поскольку по умолчанию они считаются статическими компонентами класса;

6. Любой виртуальный метод может быть объявлен как чисто виртуальный.

Чисто виртуальный метод содержит признак = 0 вместо тела, например,

virtual void Input() = 0;

Класс, имеющий в своем составе хотя бы один чисто виртуальный метод, в С++ считается абстрактным. Объявлять объекты такого класса запрещено, а указатели и ссылки – можно.

Ниже приведен пример абстрактного класса, воспроизведите его и оцените полученные результаты.

#include<iostream>

using namespace std;

class Device

{

public:

Device(){};

~Device(){};

virtual void Input() = 0;

void \*operator new(size\_t s)

{

void \*p;

p = malloc(s);

return p;

}

void operator delete(void \*p)

{

free(p);

}

};

class Mouse :public Device

{

unsigned int mouse;

public:

Mouse():Device(){};

void Input()

{

//mouse = m;

cout << " Class Mouse " << endl;

}

};

class Keyboard :public Device

{

unsigned int keyboard;

public:

Keyboard():Device(){};

void Input()

{

//keyboard = k;

cout << " Class Keyboard " << endl;

}

};

int main()

{

Device \*ptr\_Device = new Mouse;

ptr\_Device->Input();

ptr\_Device = new Keyboard;

ptr\_Device->Input();

system("pause");

return 0;

}

Здесь ответы будут получены различные, несмотря на один и тот же вызов ptr\_Device->Input();.

В этом и проявляется одно из замечательных свойств ООП – полиморфизма, реакции каждого объекта в соответствии со своим типом.