Абстрактные классы

В C++, класс в котором существует хотя бы один чистый виртуальный метод (`pure virtual`) принято считать абстрактным. Если виртуальный метод не переопределен в дочернем классе, код не скомпилируется. Также, в C++ создать объект абстрактного класса невозможно — попытка тоже вызовет ошибку при компиляции.

```
// абстрактный базовый класс
class Person {
    public:
            string name;
            int age;

            // чистый виртуальный метод
            virtual void specialization() = 0;
};

// производный класс
class Student : public Person{
     public:
            int yearOfStudy;
            void specialization() { cout << "Student" << endl;  }
}
```

Механизм виртуальных функций реализует полиморфизм времени выполнения: какой виртуальный метод вызовется будет известно только во время выполнения программы. Пример:

```
class Person {
public:
    std::string name;
    int age;
    virtual void specialization() = 0;
};

class Student : public Person{
public:
    int yearOfStudy;
    void specialization() {
        std::cout << "Student" << std::endl;
    }
};

class Programmer : public Person{
public:
    int level;
    void specialization() {
        std::cout << "Programmer" << std::endl;
    }
};

void foo( Person &person ){
    person.specialization();
}

int main(void){
    Student student;
    Programmer  programmer;
    foo(student);
    foo(programmer);
}
```

Оператор `typeid` поддерживает в языке C++ возможность идентификации динамической информации о типе. В языке C++ оператор `typeid` возвращает ссылку на объект `type_info`, описывающий тип объекта, к которому принадлежит оператор typeid. общая форма записи оператора `typeid` такова.

`typeid(`объект`)`

Класс `type_info` определяет следующие открытые члены.

```
bool operator == (const type_info &ob) const;
bool operator != (const type_info &ob) const;
bool before (const type_info &ob) const;
const char *name() const;
```

Перегруженные операторы == и != служат для сравнения типов. Функция `before()` возвращает значение `true`, если возвращающий объект в порядке сопоставления стоит перед объектом, используемым в качестве параметра. Эта функция предназначена в основном для внутреннего использования. Её значение возврата не имеет нечего общего с наследованием или иерархией классов. Функция `name()` возвращает указатель на имя типа. Если оператор `typeid` применяется к указателю полиморфного класса, он автоматически возвращает тип объекта, на который он указывает. (полиморфный класс — это класс который содержит хотя-бы одну виртуальную функцию.) Следовательно, оператор `typeid` можно использовать для определения типа объекта, адресуемого указателем на базовый класс.
В языке C оператор `typeid` не поддерживается.

Множественное наследование

В языке C++ множественное наследование позволяет одному дочернему классу иметь несколько родителей. Для использования множественного наследования нужно просто указать через запятую тип наследования и второй родительский класс:

```
class Human
{
    string name;

public:
    Human(string arg_name) : name(arg_name){}
    string getName() { return name; }
};
```

```
class Employee
{
    string employer;

public:
    Employee(string arg_employer) : employer(arg_employer){}
    string getEmployer() { return employer; }
};

// Класс Teacher открыто наследует свойства классов Human и Employee
class Teacher: public Human, public Employee
{
public:
    Teacher(string name, string employer ) : Human(name), Employee(employer){}
};
```

Множественное наследование может быть источником проблем. Может возникнуть неоднозначность, когда несколько родительских классов имеют метод с одним и тем же именем, например:

```
class USBDevice
{
    long id;
public:
    USBDevice(long arg_id) : id(arg_id){}
    long getID() { return id; }
};

class NetworkDevice
{
    long id;
public:
    NetworkDevice(long arg_id) : id(arg_id){}
    long getID() { return id; }
};

class WirelessAdapter: public USBDevice, public NetworkDevice
{
public:
    WirelessAdapter(long usbId, long networkId) : USBDevice(usbId),
NetworkDevice(networkId){}
};

…

WirelessAdapter adapter(1234, 5678);
cout << adapter.getID() << endl; // какая версия getID() будет вызвана?
```

Компилятор пытается определить, есть ли у `WirelessAdapter` метод `getID()`. Так как этого метода у него нет, поэтому компилятор двигается по цепочке наследования вверх и смотрит, есть ли этот метод в каком-либо из родительских классов. И здесь возникает проблема — `getID()` есть как у `USBDevice`, так и у `NetworkDevice`. Вызов этого метода приведет к неоднозначности, компиляция завершится ошибкой. Проблему можно обойти, если явно указать компилятору, какую версию `getID()` следует вызвать:

```
WirelessAdapter adapter(1234, 5678);
cout << adapter.USBDevice::getID();
```

Ромбовидное наследование

Ромбовидное наследование — это ситуация, когда один класс имеет 2 родительских класса, каждый из которых, в свою очередь, наследует свойства одного и того же родительского класса.

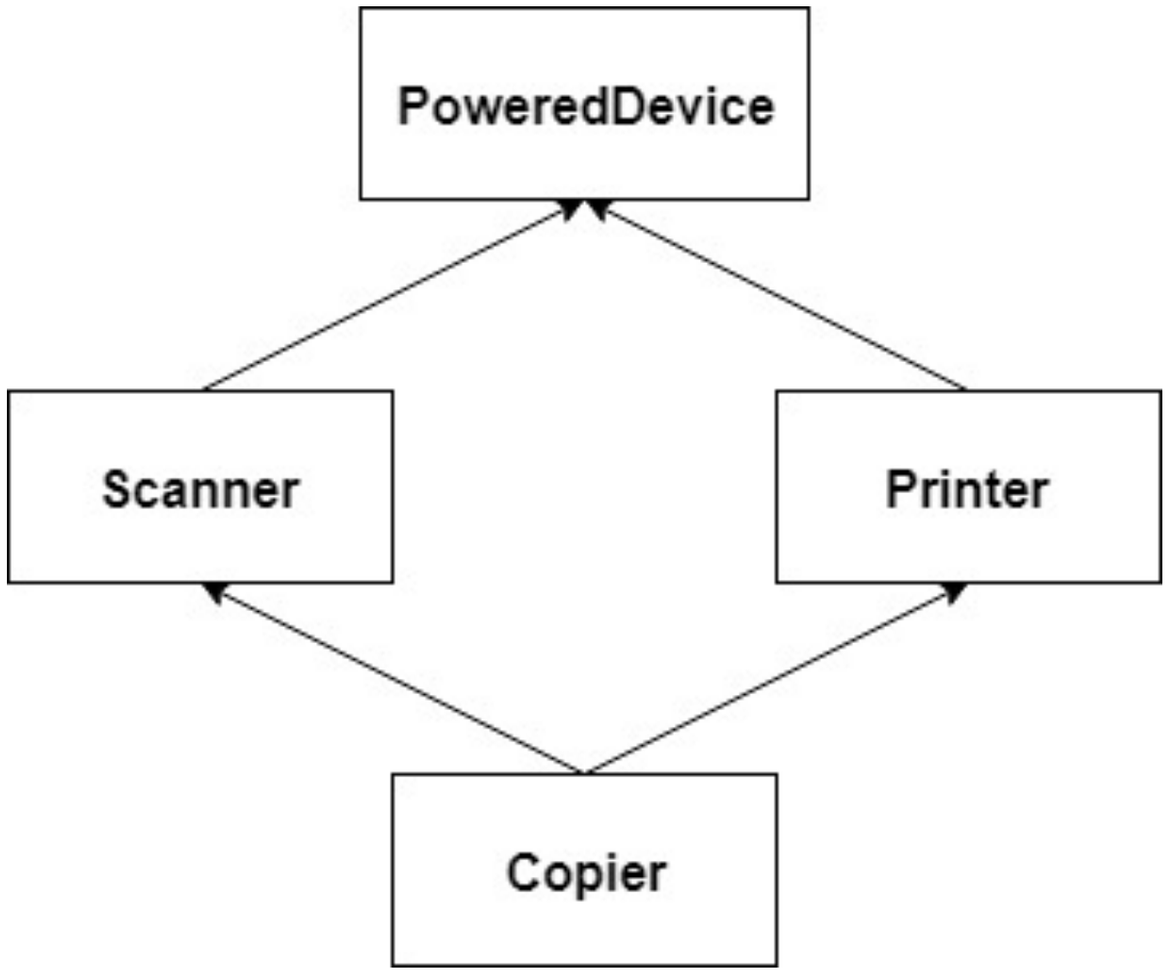
```
class PoweredDevice
{

};
class Scanner: public PoweredDevice
{

};
class Printer: public PoweredDevice
{

};
class Copier: public Scanner, public Printer
{

};
```



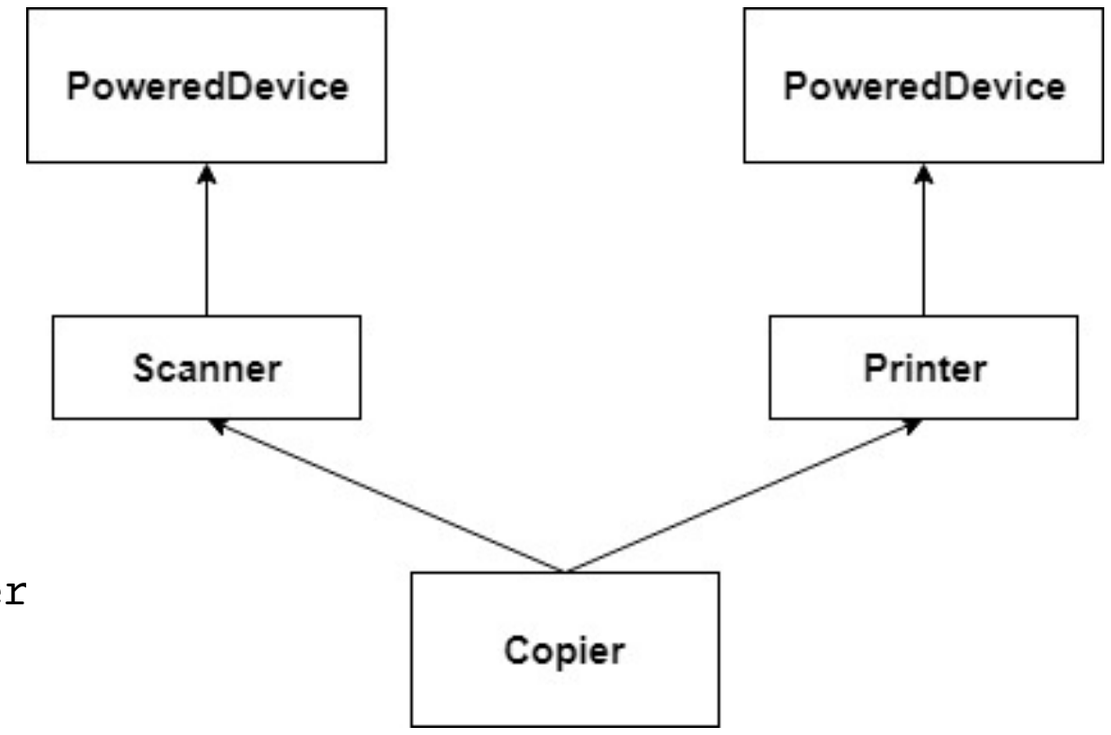
На самом деле создание объекта класса `Copier` порождает две копии класса `PoweredDevice`: одну от `Printer` и одну от `Scanner`.

```
class PoweredDevice
{
public:
    PoweredDevice(int power) { cout << "PoweredDevice: " << power << endl; }
};

class Scanner : public PoweredDevice
{
public:
    Scanner(int scanner, int power)
     : PoweredDevice(power) { cout << "Scanner: " << scanner << endl; }
};

class Printer: public PoweredDevice
{
public:
    Printer(int printer, int power)
     : PoweredDevice(power){
     cout << "Printer: " << printer << endl;
     }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power
     : Scanner(scanner, power),
     Printer(printer, power) {}
};
```



При вызове `Copier copier(1, 2, 3);` получаем результат:

```
PoweredDevice: 3
Scanner: 1
PoweredDevice: 3
Printer: 2
```

Очевидно, что конструктор `PoweredDevice` вызывается дважды. Хотя иногда так и нужно, а иногда нужно, чтобы была одна копия `PoweredDevice`: общая как для `Scanner`, так и для `Printer`. Чтобы сделать родительский класс общим, используется ключевое слово `virtual` в строке объявления дочернего класса.

Теперь, при создании класса `Copier`, мы получим только одну копию `PoweredDevice`, которая будет общей как для `Scanner`, так и для `Printer`. Пример виртуального базового класса:

```
class PoweredDevice
{
public:
    PoweredDevice(int power) { cout << "PoweredDevice: " << power << '\n'; }
};

class Scanner : virtual public PoweredDevice // PoweredDevice теперь виртуальный базовый класс
{
public:
    Scanner(int scanner, int power) :
    PoweredDevice(power) // создания объекта класса Printer в этой программе игнорируется
    { cout << "Scanner: " << scanner << '\n'; }
};

class Printer: virtual public PoweredDevice // PoweredDevice теперь виртуальный базовый класс
{
public:
    Printer(int printer, int power) :
    PoweredDevice(power) // создания объекта класса Printer в этой программе игнорируется
    { cout << "Printer: " << printer << '\n'; }
};

class Copier: public Scanner, public Printer
{
public:
    Copier(int scanner, int printer, int power) : Scanner(scanner, power),
Printer(printer, power),
PoweredDevice(power) // построение PoweredDevice выполняется здесь
{}
};
```

Вывод вышеописанного примера для вызова `Copier copier(1, 2, 3)`:

```
PoweredDevice: 3
Scanner: 1
Printer: 2
```

`PoweredDevice` теперь создается только один раз.
Виртуальные базовые классы всегда создаются перед не виртуальными базовыми классами, что обеспечивает построение всех базовых классов до построения их производных классов. Конструкторы `Scanner` и `Printer` по-прежнему вызывают конструктор `PoweredDevice`. При создании объекта `Copier` эти вызовы конструктора просто игнорируются, так как именно `Copier` отвечает за создание `PoweredDevice`, а не `Scanner` или `Printer`. Однако, если бы создавались объекты `Scanner` или `Printer`, то эти конструкторы вызывались бы и применялись обычные правила наследования.

Если класс, становясь дочерним, наследует один или несколько классов, которые, в свою очередь, имеют виртуальные родительские классы, то наиболее дочерний класс отвечает за создание виртуального родительского класса. В программе, приведенной выше, Copier наследует Printer и Scanner, которые оба имеют общий виртуальный родительский класс PoweredDevice. Copier, наиболее дочерний класс, отвечает за создание PoweredDevice. Это работает даже в случае одиночного наследования: когда Copier наследует только Printer, а Printer виртуально наследует PoweredDevice, то Copier по-прежнему ответственный за создание PoweredDevice.

Ключевое слово static

Ключевое слово static имеет разные значения в разных ситуациях, static можно применять и к переменным внутри блока, но тогда его значение будет другим. Локальные переменные имеют автоматическую продолжительность жизни, т.е. создаются, когда блок начинается, и уничтожаются при выходе из него. Использование ключевого слова static с локальными переменными изменяет их свойство продолжительности жизни с автоматического на статическое. Статическая переменная сохраняет свое значение даже после выхода из блока, в котором она определена. То есть она создается и инициализируется только один раз, а затем сохраняется на протяжении выполнения всей программы.

```
void incrementAndPrint()
{
    int value = 1; // автоматическая продолжительность жизни
    ++value;
    cout << value << endl;
} // переменная value уничтожается здесь

int main()
{
    incrementAndPrint(); // 2
    incrementAndPrint(); // 2
    incrementAndPrint(); // 2
}
```

Каждый раз, при вызове функции `incrementAndPrint()`, создается переменная `value`, которой присваивается значение `1`, далее функция `incrementAndPrint()` увеличивает значение переменной до `2`, а затем выводит его. Когда incrementAndPrint() завершает свое выполнение, переменная выходит из области видимости и уничтожается. Если переменной `value` добавить ключевое слово `static`, то пример будет работать иначе:

```
void incrementAndPrint()
{
    static int value = 1; // переменная value является статической
    ++value;
    cout << value << endl;
} // переменная value не уничтожается, но становится недоступной вне функции

int main()
{
    incrementAndPrint(); // 2
    incrementAndPrint(); // 3
    incrementAndPrint(); // 4
}
```

Класс может содержать в себе статические переменные. Если они являются открытыми, то есть возможность получить к ним доступ напрямую через имя класса и оператор разрешения области видимости.

```
class Anything
{
public:
    static int s_value;
};

// статический неконстантный член класса требует определения вне класса
int Anything::s_value = 1;

int main(){
  Anything anything;
  cout << anything.s_value << endl;  // вызов через объект
  cout << Anything::s_value << endl;  // вызов через имя
  класса и оператор разрешения области видимости
}
```

Если они являются закрытыми, то получить доступ напрямую уже не удастся. В таком случае, доступ к закрытым членам класса осуществляется через `public`-методы.

```
class Anything
{
private:
    static int s_value;
public:
    static int getValue(){ return s_value; }
};

// статический неконстантный член класса требует определения вне класса
int Anything::s_value = 1;

int main(){
   Anything anything;
   cout << anything.getValue() << endl;  // вызов через объект
   cout << Anything::getValue() << endl;  // вызов через имя
   класса и оператор разрешения области видимости
}
```

Статический член класса похож на глобальную переменную. Отличие заключается в том, что возможно ограничить к ней доступ через указание `public/private/protected` и требованием писать полное имя при доступе из-вне класса. Определение статической переменой обычно происходит вне класса, чаще всего в файле реализации т.е. в `cpp`-файле. Такой подход необходим, чтобы не было дублирования кода инициализации в других модулях, подключающих заголовочный файл. Если оставить определение в заголовочном файле, то получим ошибку множественного определения во всех модулях (кроме первого), подключающих его через `#include`. Начиная со стандарта `C++17` появилась возможность объявлять статические неконстантные члены класса встроенными `inline`. В таком случае определение вне класса не требуется.

Статический член класса может быть объявлен константой, в таком случае его определение может быть в рамках класса.

```
class Anything
{
public:
   static const int s_value = 1; // определение статического константного члена класса
};
```

```
int main(){
   Anything anything;
   cout << anything.s_value << endl; // вызов через объект
   cout << Anything::s_value << endl; // вызов через имя класса и оператор разрешения области видимости
}
```

Подобно статическим переменным-членам, статические методы не привязаны к какому-либо одному объекту класса. Их можно вызывать напрямую через имя класса и оператор разрешения области видимости, а также через объекты класса. Поскольку статические методы не привязаны к объекту, то они не имеют скрытого указателя *this и могут напрямую обращаться к другим статическим переменным или функциям, но не могут напрямую обращаться к нестатическим членам класса.

```
class Anything
{
public:
	// объявление статической переменной
	static int s_value;
	// объявление статического метода
	static int getValue();
};

// определение статической переменной
int Anything::s_value = 1;

// определение статического метода
int Anything::getValue(){ return s_value; }
```

Преобразования типов

В C++ есть четыре оператора явного преобразования (приведения) типов: `const_cast`, `static_cast`, `dynamic_cast` и `reinterpret_cast`. Кроме того, для совместимости поддерживается приведение в стиле C.

`const_cast` убирает (или добавляет, но это редко используется) так называемые cv-спецификаторы (`cv qualifiers`) — `const` и `volatile`. Спецификатор `volatile` встречается редко, так что `const_cast` обычно применяется для снятия const при обращении к некорректно написанным сторонним библиотекам. При использовании остальных операторов приведения типов cv-спецификаторы остаются неизменными.

```
double x;
const double *px = &x; // *px имеет тип const double, но переменная x не константная
*px = 10.0; // ошибка!
double *px_free = const_cast<double *> (px); // const снят
*px_free = 10.0; // ошибка!
```

Если приведение невозможно, выдаётся ошибка на этапе компиляции. Необходимо помнить, что попытка записи в изначально константный объект или переменную приводит к неопределённому поведению. Лучше `const_cast` не использовать, изменить программу так, чтобы его использование не требовалось.

`static_cast` статически (то есть на этапе компиляции) преобразует выражение одного типа к другому типу. Может быть использован везде, где допустимо неявное преобразование типов (в частности, преобразования чисел вроде `int i = 1.3`, преобразование указателя произвольного типа в нетипизированный указатель `void*`, указателя на производный класс в указатель на базовый или перечислимого типа в интегральный), а также для приведения:

- любого типа к типу `void` (допустимое, но обычно ненужное на практике преобразование);
- нетипизированного указателя `void*` к указателю произвольного типа.
- ссылку базового класса к производному. Допустимо, если объект на самом деле производного класса, но опасно, если это не так. Чтобы иметь возможность проверить корректность, для такого преобразования лучше использовать `dynamic_cast` — но применимо это только для полиморфного базового класса.
- указателя на базовый класс к указателю на производный класс (аналогично, надёжнее использовать `dynamic_cast`, если это возможно);
- интегральных типов (`int`, `char` и т. п.) к перечислимым (`enum`).

Если приведение невозможно, выдастся ошибка на этапе компиляции. Позволяет привести одно значение к другому значению. Именно оператор `static_cast` наряду с неявными преобразованиями наиболее часто используется на практике.

`dynamic_cast` динамически (на этапе выполнения) приводит полиморфный базовый класс к производному с проверкой преобразования. Таким образом, чтобы можно было воспользоваться оператором `dynamic_cast`, в базовом классе должна быть хотя бы одна виртуальная функция (таблица виртуальных функций используется для определения реального типа объекта). Если это условие не соблюдено, выдаётся ошибка на этапе компиляции.

Используется для приведения:

- указателя на базовый класс к указателю на производный класс:
  `dynamic_cast<derived_class *>(base_class_ptr_expr)`
  если приведение невозможно, будет возвращён `NULL`
- базового класса к ссылке на производный класс:
  `dynamic_cast<derived_class &>(base_class_ref_expr)`
  если приведение невозможно, будет выброшено исключение `bad_cast`. В отличие от других операторов приведения типов, `dynamic_cast` позволяет определить корректность преобразования на этапе выполнения программы и при необходимости обработать ошибку (`NULL` или исключение `bad_cast`).

`reinterpret_cast` интерпретирует память в соответствии с заданным типом без проверок. Используется для приведения указателя к указателю на другой тип, указателя к целому, целого к указателю, ссылки к ссылке, объекта к ссылке (в последнем случае фактически интерпретируется адрес объекта). `reinterpret_cast` не может привести одно значение к другому значению (для приведения значений используется `static_cast`).

```
reinterpret_cast<T2 *>(T1 *)
reinterpret_cast<integer_expression>(T *)
reinterpret_cast<T *>(integer_expression)
```

Пример `reinterpret_cast:`

```
double x = 1;
int i = -1;
char *cd = reinterpret_cast<char *>(&x);
unsigned *ui = reinterpret_cast<unsigned*>(&i);
unsigned &ui = reinterpret_cast<unsigned &>(i);
```

В примере `cd` будет указывать на первый байт переменной x.
Указатель ui будет указывать на байты i.

```
unsigned *u = reinterpret_cast<unsigned *>(&i);
unsigned *u_err = reinterpret_cast<unsigned
   *>(i);  // никогда не стоит так делать !
```

Если каст указателя будет применен к переменной, которая меньше, чем новый, то будет частично захвачено значение, которое стоит в стеке предыдущим:

```
double x = 32.3244;
// double x = 32.33434;
int i = -1;
double *u = reinterpret_cast<double *>(&i);
std::cout << *u << std::endl;
```

>> -3.90132e+233

```
//double x = 32.3244;
double x = 32.33434;
int i = -1;
double *u = reinterpret_cast<double *>(&i);
std::cout << *u << std::endl;
```

>> -7.30848e-120

Приведение в стиле C или C-style cast — самое медленное преобразование, так как последовательно перебираются следующие вызовы:

1) `const_cast`
2) `static_cast`
3) `static_cast` + `const_cast`
4) `reinterpret_cast`
5) `reinterpret_cast` + `const_cast`

Таким образом, приведение в стиле C универсально.

```
double x = 1;
int i = -i;
char *pc = (char *)(&x);
unsigned &u_ref = (unsigned &)i;
unsigned *u_ptr = (unsigned *)&i;
```

Считается, что приведение в стиле C найти в коде труднее, чем операторы XXX_cast.