Наследование

Введение

Пример базового наследования:

```
#include <iostream>
using namespace std;

class Device {
public:
    int serial_number = 12345678;
    void turn_on() {
    cout << "Device is on" << endl;
    }
private:
    int pincode = 87654321;
class Computer: public Device {};

int main() {
    Computer Computer_instance;
    Computer_instance.turn_on();
    cout << "Serial number is: " << Computer_instance.serial_number
    // cout << "Pin code is: " << Computer_instance.pincode << endl;
    // will cause compile time error
    return 0;
}
```

Типы наследования:

- public - публичные (public) и защищенные (protected) данные наследуются без изменения
- private - все унаследованные данные становятся приватными(private)
- protected - все унаследованные данные становятся защищенными (protected)

Конструкторы и деструкторы (не виртуальные) не наследуются

При инициализации дочернего класса, сначала вызываются все конструкторы до него, но при уничтожении (прекращении времени жизни) наоборот - вызываются

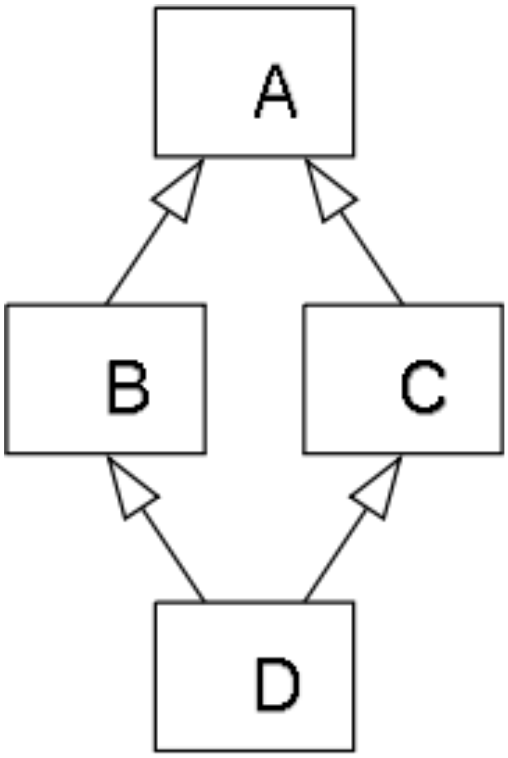
деструкторы от текущего класса, до основного предка.

Множественное наследования

```
#include <iostream>
using namespace std;

class Computer {
private:
    void turn_on() {
        cout << "Computer is on." << endl;
    }
};

class Monitor {
    public:
    void turn_on() {
        court << "Monitor is on." << endl;
    }
};

class Laptop: public Computer, public Monitor {};

int main() {
    Laptop Laptop_instance;
    // Laptop_instance.turn_on();
    // will cause compile time error
    return 0;
}
```

Множественное наследование требует тщательного проектирования, так как может привести к непредвиденным последствиям. Большинство таких последствий вызваны неоднозначностью в наследовании. В данном примере `Laptop` наследует метод `turn_on()` от обоих родителей и неясно какой метод должен быть вызван.

Проблема ромба



Проблема ромба (Diamond problem)- классическая проблема в языках, которые поддерживают возможность множественного наследования. Эта проблема возникает когда классы `B` и `C` наследуют `A`, а класс `D` наследует `B` и `C`.

К примеру, классы `A`, `B` и `C` определяют метод `print_letter()`. Если `print_letter()` будет вызываться классом `D`, неясно какой метод должен быть вызван — метод класса `A`, `B` или `C`. В C ++ решение проблемы оставлено на усмотрение программиста.

Варианты решения:

- вызвать метод конкретного суперкласса;
- обратиться к объекту подкласса как к объекту определенного суперкласса;
- переопределить проблематичный метод в последнем дочернем классе (в коде — `turn_on()` в подклассе `Laptop`).

```
#include <iostream>
using namespace std;

class Device {
public:
    void turn_on() {
        cout << "Device is on." << endl;
    }
};

class Computer: public Device {};
class Monitor: public Device {};
class Laptop: public Computer, public Monitor {
public:
    /*
    void turn_on() {
        cout << "Laptop is on." << endl;
```

```
    }
    // uncommenting this function will resolve diamond problem
    */
}

int main() {
    Laptop Laptop_instance;

    // Laptop_instance.turn_on();
    // will produce compile time error
    // if Laptop.turn_on function is commented out

    // calling method of specific superclass
    Laptop_instance.Monitor::turn_on();

    // treating Laptop instance as Monitor instance via static cast
    static_cast<Monitor&>( Laptop_instance ).turn_on();
    return 0;
}
```

Проблема ромба с конструкторами и деструкторами

Поскольку в C++ при инициализации объекта дочернего класса вызываются конструкторы всех родительских классов, возникает и другая проблема: конструктор базового класса `Device` будет вызван дважды.

Виртуальные функции

```
1. #include <cstdlib>
2. #include <iostream>

3. using std::cout;
4. using std::endl;

5. struct A
6. {
7.   virtual ~A() {}

8.   virtual void foo() const { cout << "A::foo()" << endl; }
9.   virtual void bar() const { cout << "A::bar()" << endl; }
10.   void baz() const { cout << "A::baz()" << endl; }
11. };
```

```
12. struct B : public A
13. {
14.   virtual void foo() const { cout << "B::foo()" << endl; }
15.   void bar() const { cout << "B::bar()" << endl; }
16.   void baz() const { cout << "B::baz()" << endl; }
17. };

18. struct C : public B
19. {
20.   virtual void foo() const { cout << "C::foo()" << endl; }
21.   virtual void bar() const { cout << "C::bar()" << endl; }
22.   void baz() const { cout << "C::baz()" << endl; }
23. };

24. int main()
25. {
26.   cout << "pA is B:" << endl;
27.   A * pA = new B;
28.   pA->foo();
29.   pA->bar();
30.   pA->baz();
31.   delete pA;

32.   cout << "pA is C:" << endl;
33.   pA = new C;
34.   pA->foo();
35.   pA->bar();
36.   pA->baz();
37.   delete pA;

38.   return EXIT_SUCCESS;
39. }
```

Вывод:

```
pA is B:
B::foo()
B::bar()
A::baz()

pA is C:
C::foo()
C::bar()
A::baz()
```

Вывод: *виртуальная функция становится виртуальной до конца иерархии, а ключевое слово **virtual** является «ключевым» только в первый раз, а в последующие разы оно несет в себе чисто информативную функцию для удобства программистов*.

Как работает механизм виртуальных функций ?

Раннее и позднее связывание. Таблица виртуальных функций

Связывание — это сопоставление вызова функции с вызовом. В C++ все функции по умолчанию имеют *раннее связывание*, то есть компилятор и компоновщик решают, какая именно функция должна быть вызвана, до запуска программы. Виртуальные функции имеют *позднее связывание*, то есть при вызове функции нужное тело выбирается на этапе выполнения программы.

Встретив ключевое слово virtual, компилятор помечает, что для этого метода должно использоваться позднее связывание: для начала он создает для класса таблицу виртуальных функций, а в класс добавляет новый скрытый для программиста член — указатель на эту таблицу. (На самом деле, насколько я знаю, стандарт языка не предписывает, как именно должен быть реализован механизм виртуальных функций, но реализация на основе виртуальной таблицы стала стандартом де факто.). Рассмотрим этот пруфкод:

```
1. #include <cstdlib>
2. #include <iostream>

3. struct Empty {};

4. struct EmptyVirt { virtual ~EmptyVirt(){} };

5. struct NotEmpty { int m_i; };

6. struct NotEmptyVirt
7. {
8.   virtual ~NotEmptyVirt() {}
9.   int m_i;
10. };

11. struct NotEmptyNonVirt
12. {
13.   void foo() const {}
```

```
14.    int m_i;
15. };

16. int main()
17. {
18.    std::cout << sizeof(Empty) << std::endl;
19.    std::cout << sizeof(EmptyVirt) << std::endl;
20.    std::cout << sizeof(NotEmpty) << std::endl;
21.    std::cout << sizeof(NotEmptyVirt) << std::endl;
22.    std::cout << sizeof(NotEmptyNonVirt) << std::endl;

23.    return EXIT_SUCCESS;
24. }
```

Вывод (может отличаться в зависимости от платформы):

```
1
4
4
8
4
```

Что можно понять из этого примера. Во-первых, размер «пустого» класса всегда больше нуля, потому что компилятор специально вставляет в него фиктивный член. Как пишет Эккель, «представьте процесс индексирования в массиве объектов нулевого размера, и все станет ясно» ;) Во-вторых, мы видим, что размер «непустого» класса NotEmptyVirt при добавлении в него виртуальной функции увеличился на стандартный размер указателя на void; а в «пустом» классе EmptyVirt фиктивный член, который компилятор ранее добавлял для приведения класса к ненулевому размеру, был заменен на указатель. В то же время добавление невиртуальной функции в класс на размер не влияет. Имя указателя на таблицу отличается в зависимости от компилятора. К примеру, компилятор Visual Studio 2008 называет его __vfptr, а саму таблицу ‘vftable’. В литературе указатель на таблицу виртуальных функций принято называть VPTR, а саму таблицу VTABLE.

Что представляет собой таблица виртуальных функций и для чего она нужна? Таблица виртуальных функций хранит в себе адреса всех виртуальных методов класса (по сути, это массив указателей), а также всех виртуальных методов базовых классов этого класса.

Таблиц виртуальных функций у нас будет столько, сколько есть классов, содержащих виртуальные функции — по одной таблице на класс. Объекты каждого из классов содержат именно указатель на таблицу, а не саму таблицу!

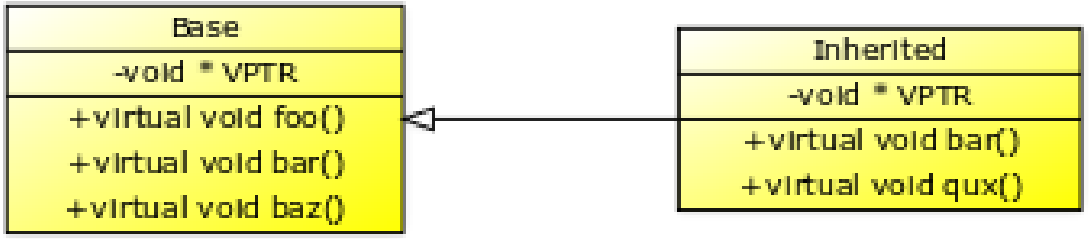
Итак, для каждого класса у нас будет создана таблица виртуальных функций. Каждой виртуальной функции базового класса присваивается подряд идущий индекс (в порядке объявления функций), по которому в последствие и будет определяться адрес тела функции в таблице VTABLE. При наследовании базового класса, производный класс «получает» и таблицу адресов виртуальных функций базового класса. Если какой-либо виртуальный метод в производном классе переопределяется, то в таблице виртуальных функций этого класса адрес тела соответствующего метода просто будет заменен на новый. При добавлении в производный класс новых виртуальных методов VTABLE производного класса расширяется, а таблица базового класса естественно остается такой же, как и была. Поэтому через указатель на базовый класс нельзя виртуально вызвать методы производного класса, которых не было в базовом — ведь базовый класс о них ничего «не знает» (дальше мы все это посмотрим на примере).

Конструктор класса теперь должен делать дополнительную операцию: инициализировать указатель VPTR адресом соответствующей таблицы виртуальных функций. То есть, когда мы создаем объект производного класса, сначала вызывается конструктор базового класса, инициализирующий VPTR адресом «своей» таблицы виртуальных функций, затем вызывается конструктор производного класса, который перезаписывает это значение.

При вызове функции через адрес базового класса (читайте — через указатель на базовый класс) компилятор сначала должен по указателю VPTR обратиться к таблице виртуальных функций класса, а из неё получить адрес тела вызываемой функции, и только после этого делать call.

Из всего вышесказанного можно сделать вывод, что механизм позднего связывания требует дополнительных затрат процессорного времени (инициализация VPTR конструктором, получение адреса функции при вызове) по сравнению с ранним.

Рассмотрим следующую иерархию:



В данном случае получим две таблицы виртуальных функций:

Base			Inherited	
0	Base::foo()		0	Base::foo()
1	Base::bar()		1	Inherited::bar()
2	Base::baz()		2	Base::baz()
			3	Inherited::qux()

Как видим, в таблице производного класса адрес второго метода был заменен на соответствующий переопределенный. Пруфкод:

```
1. #include <cstdlib>
2. #include <iostream>

3. using std::cout;
4. using std::endl;

5. struct Base
6. {
7.   Base() { cout << "Base::Base()" << endl; }
8.   virtual ~Base() { cout << "Base::~Base()" << endl; }

9.   virtual void foo() { cout << "Base::foo()" << endl; }
10.   virtual void bar() { cout << "Base::bar()" << endl; }
11.   virtual void baz() { cout << "Base::baz()" << endl; }
12. };

13. struct Inherited : public Base
14. {
15.   Inherited() { cout << "Inherited::Inherited()" << endl; }
16.   virtual ~Inherited() { cout << "Inherited::~Inherited()" << en

17.   virtual void bar() { cout << "Inherited::bar()" << endl; }
18.   virtual void qux() { cout << "Inherited::qux()" << endl; }
19. };

20. int main()
21. {
22.   Base * pBase = new Inherited;
23.   pBase->foo();
24.   pBase->bar();
25.   pBase->baz();
26.   //pBase->qux();     // Ошибка
27.   delete pBase;
```

```
28.     return EXIT_SUCCESS;
29. }
```

Что происходит при запуске программы? Вначале объявляем указатель на объект типа Base, которому присваиваем адрес вновь созданного объекта типа Inherited. При этом вызывается конструктор Base, инициализирует VPTR адресом VTABLE класса Base, а затем конструктор Inherited, который перезаписывает значение VPTR адресом VTABLE класса Inherited. При вызове pBase->foo(), pBase->bar() и pBase->baz() компилятор через указатель VPTR достает фактический адрес тела функции из таблицы виртуальных функций. Как это происходит? Вне зависимости от конкретного типа объекта компилятор знает, что адрес функции foo() находится на первом месте, bar() — на втором, и т.д. (как я и говорил, в порядке объявления функций). Таким образом, для вызова, к примеру, функции baz() он получает адрес функции в виде VPTR+2 — смещение от начала таблицы виртуальных функций, сохраняет этот адрес и подставляет в команду call. По этой же причине, вызов pBase->qux() приводит к ошибке: несмотря на то, что фактический тип объекта Inherited, когда мы присваиваем его адрес указателю на Base, происходит восходящее приведение типа, а в таблице VTABLE класса Base никакого четвертого метода нет, поэтому VPTR+3 указывало бы на «чужую» память (к счастью, такой код даже не компилируется).

Также становится понятно, почему виртуальные функции работают только при обращении по адресу объекта (через указатели либо через ссылки). Как я уже сказал, в этой строке

`Base * pBase = new Inherited;`

Происходит повышающее приведение типа: Inherited *приводится к Base*, но в любом случае указатель всего лишь хранит адрес «начала» объекта в памяти. Если же повышающее приведение производить непосредственно для объекта, то он фактически «обрезается» до размера объекта базового класса. Поэтому логично, что для вызова функций «через объект» используется раннее связывание — компилятор и так «знает» фактический тип объекта.

TODO: Виртуальные деструкторы

Источники:

- Наследование
- Виртуальные функции
- Виртуальные деструкторы