



Углубленное программирование на языке C++



Алексей Петров

Лекция №4.

Дополнительные вопросы ООП на языке C++.

Динамическая идентификация типов

1. Абстрактные классы.
2. Принципы LSP, ISP, DIP.
3. Множественное и виртуальное наследование.
4. Динамическая идентификация типов времени выполнения (RTTI) и операции приведения типов. Производительность и безопасность полиморфизма и средств поддержки RTTI.
5. Постановка задач к практикуму №4.



Наследование содействует **повторному использованию** атрибутов и методов класса, а значит, делает процесс разработки ПО более эффективным. Возникающие между классами А и В отношения наследования позволяют, например, говорить, что:

- класс А является **базовым (родительским)** классом, классом-предком, **надклассом** (англ. [superclass](#));
- класс В является **производным (дочерним)** классом, классом-потомком, **подклассом** (англ. [subclass](#)).

Отношения наследования связывают классы в **иерархию наследования**, вид которой зависит от числа базовых классов у каждого производного:

- при **одиночном наследовании** иерархия имеет вид дерева;
- при **множественном наследовании** — вид направленного ациклического графа (НАГ) произвольного вида.

В том случае если базовый и производный классы имеют общий открытый интерфейс, говорят, что производный класс представляет собой **подкласс** базового.

Отношение между классом и подклассом, позволяющее указателю или ссылке на базовый класс без вмешательства программиста адресовать объект производного класса, возникает в C++ благодаря поддержке **полиморфизма**.

Полиморфизм позволяет предложить такую реализацию ядра объектно-ориентированного приложения, которая не будет зависеть от конкретных используемых подклассов.

В рамках классического объектного подхода, — а равно и процедурного программирования, — адрес вызываемой функции (**метода класса**) определяется на этапе компиляции (**сборки**). Такой порядок связывания вызова функции и ее адреса получил название **раннего (статического)**.

Позднее (динамическое) связывание состоит в нахождении (**разрешении**) нужной функции во время исполнения кода. При этом работа по разрешению типов перекладывается с программиста на компилятор.

В языке C++ динамическое связывание поддерживается механизмом **виртуальных** методов класса, для работы с которыми компиляторы строят **таблицы виртуальных методов** (**англ. VMT, virtual method table**).

Определение наследования: пример



```
// описание производного класса
// (не включает список базовых классов!)
class Deposit;
/* ... */

// определения классов
class Account
{
    /* ... */
};
class Deposit : public Account
{
    /* ... */
};
```

Атрибуты и методы базового класса, как правило, должны быть **непосредственно доступны для производных классов** и непосредственно недоступны для прочих компонентов программы. В этом случае они помещаются в секцию `protected`, в результате чего защищенные члены данных и методы базового класса:

- доступны производному классу (**прямому потомку**);
- недоступны классам вне рассматриваемой иерархии, глобальным функциям и вызывающей программе.

Если **наличие прямого доступа** к члену класса со стороны производных классов **нежелательно**, он вводится как **закрытый**. Закрытые члены класса не наследуются потомками. Для доступа к ним класс-потомок должен быть объявлен в классе-предке как дружественный. **Отношения дружественности не наследуются.**

Перегрузка и перекрытие членов класса



Члены данных базового класса могут перекрываться одноименными членами данных производного класса, при этом их типы не должны обязательно совпадать. (Для доступа к члену базового класса его имя должно быть квалифицировано.)

Методы базового и производного классов не образуют множество перегруженных функций. В этом случае методы производного класса не перегружают (**англ. `overload`**), а перекрывают (**англ. `override`**) методы базового.

Для явного создания объединенного множества перегруженных функций базового и производного классов используется объявление **using**, которое вводит именованный член базового класса в область видимости производного.

Перегрузка и перекрытие членов класса: пример



Примечание: в область видимости производного класса попадают все одноименные методы базового класса, а не только некоторые ИЗ НИХ.

```
class Account
{ /* ... */
    void display(const char *fmt);
    void display(const int mode = 0);
};
class Deposit : public Account
{
    void display(const string &fmt);
    using Account::display;
    /* ... */
};
```

Порядок вызова конструкторов производных классов (1 / 2)



Порядок вызова конструкторов объектов-членов, а также базовых классов **при построении объекта производного класса** не зависит от порядка их перечисления в списке инициализации конструктора производного класса и является следующим:

- конструктор базового класса (если таковых несколько, конструкторы вызываются в порядке перечисления имен классов в списке базовых классов);
- конструктор объекта-члена (если таковых несколько, конструкторы вызываются в порядке объявления членов данных в определении класса);
- конструктор производного класса.

Порядок вызова конструкторов производных классов (2 / 2)



Конструктор производного класса может вызывать конструкторы классов, непосредственно являющихся базовыми для данного (прямых предков), и — без учета виртуального наследования — только их.

Примечание: правильно спроектированный конструктор производного класса не должен инициализировать атрибуты базового класса напрямую (путем присваивания значений).

Список инициализации при наследовании: пример



```
class Alpha {  
public:  
    // Alpha();  
    Alpha(int i); /* ... */  
};  
class Beta : public Alpha {  
public:  
    Beta() : _s("dictum factum") { }  
    // Beta() : Alpha(), _s("dictum factum") { }  
    Beta(int i, string s) : Alpha(i), _s(s) { }  
protected:  
    string _s; /* ... */  
};
```

Порядок вызова деструкторов производных классов



Порядок вызова деструкторов при уничтожении объекта производного класса прямо противоположен порядку вызова конструкторов и является следующим:

- деструктор производного класса;
- деструктор объекта-члена (**или нескольких**);
- деструктор базового класса (**или нескольких**).

Взаимная противоположность порядка вызова конструкторов и деструкторов является **строгой гарантией** языка C++.

Виртуальные функции (1 / 2)



Методы, результат разрешения вызова которых зависит от «реального» (**динамического**) типа объекта, доступного по указателю или ссылке, называются **виртуальными** и при определении в базовом классе снабжаются спецификатором **virtual**.

Примечание: в этом контексте тип непосредственно определяемого экземпляра, ссылки или указателя на объект называется статическим. Для самого объекта любого типа (автоматической переменной) статический и динамический тип совпадают.

По умолчанию объектная модель C++ работает с **невиртуальными** методами. Механизм виртуальных функций работает только в случае **косвенной адресации** (**по указателю или ссылке**).

Виртуальные функции (2 / 2)



Значения **формальных параметров** виртуальных функций определяются (а) на этапе компиляции (б) типом объекта, через который осуществляется вызов.

Отмена действия механизма **виртуализации** возможна и достигается статическим вызовом метода при помощи операции разрешения области видимости (::).

```
class Alpha {  
    /* ... */  
    virtual void display();  
};  
class Beta : public Alpha {  
    void display();  
}
```

Чистые виртуальные функции



Класс, где виртуальный метод объявляется впервые, должен определять его тело либо декларировать метод как не имеющую собственной реализации **чистую виртуальную функцию**.

Производный класс может **наследовать** реализацию виртуального метода из базового класса или **перекрывать** его собственной реализацией, при этом прототипы обеих реализаций обязаны совпадать.

```
virtual void display() = 0;
```

Примечание: единственное исключение C++ делает для возвращаемого значения. Значение, возвращаемое реализацией в производном классе, может иметь тип, открыто наследующий классу значения, возвращаемого реализацией в базовом.

Класс, который определяет или наследует хотя бы одну чистую виртуальную функцию, является **абстрактным**.

Экземпляры абстрактных классов **создавать нельзя**. Абстрактный класс может реализовываться только как подобъект производного, неабстрактного класса.

Чистые виртуальные функции **могут иметь** тело, вызов которого, впрочем, может производиться только статически (**при помощи операции разрешения области видимости**), но не динамически (**при помощи механизма виртуализации**).

Чистые виртуальные функции и абстрактные классы: пример



```
// class Alpha;  
// class Beta : public Alpha;  
  
virtual void Alpha::display() { /* ... */ };  
  
void Beta::display()  
{  
    // вызов чистой виртуальной функции  
    Alpha::display();  
    /* ... */  
}
```

Множественное наследование в ООП — это наследование от двух и более базовых классов, возможно, с различным уровнем доступа. Язык C++ не накладывает ограничений на количество базовых классов.

При множественном наследовании **конструкторы** базовых классов вызываются **в порядке перечисления имен классов** в списке базовых классов. Порядок вызова **деструкторов** ему прямо противоположен.

Унаследованные от разных базовых классов методы не образуют множество перегруженных функций, а потому разрешаются только по имени, без учета их сигнатур.

При множественном наследовании возможна ситуация неоднократного включения подобъекта одного и того же базового класса в состав производного. Связанные с нею проблемы и неоднозначности снимает **виртуальное наследование**.

Суть виртуального наследования — включение в состав класса **единственного разделяемого подобъекта** базового класса (виртуального базового класса).

Виртуальное наследование не характеризует базовый класс, а лишь описывает его отношение к производному.

Использование виртуального наследования должно быть взвешенным проектным решением конкретных проблем объектно-ориентированного проектирования.

Конструкция объектов при виртуальном наследовании



Виртуальные базовые классы конструируются перед неvirtуальными независимо от их расположения в иерархии наследования.

Ответственность за инициализацию виртуального базового класса несет на себе ближайший (финальный) производный класс.

В промежуточных производных классах прямые вызовы конструкторов виртуальных базовых классов автоматически подавляются.

Множественное и виртуальное наследование: пример



// множественное наследование

```
class Alpha
```

```
{ /* ... */ };
```

```
class Beta : public Alpha
```

```
{ /* ... */ };
```

```
class Gamma
```

```
{ /* ... */ };
```

```
class Delta : public Beta, public Gamma
```

```
{ /* ... */ };
```

// виртуальное наследование

```
class Alpha
```

```
{ /* ... */ };
```

```
class Beta : virtual public Alpha
```

```
// то же: class Beta : public virtual Alpha
```

```
{ /* ... */ };
```

Принцип подстановки Б. Лисков (англ. *Liskov Substitution Principle, LSP*) предполагает:

Объекты любого класса могут заменяться объектами его подклассов, и это не должно влиять на свойства самой программы

Принцип разделения интерфейсов (англ. *Interface Segregation Principle*) утверждает:

(Множество мелких интерфейсов лучше, чем единственный общий

Принцип инверсии зависимостей (англ. *Dependency Inversion Principle, DIP*) гласит:

Абстракции не должны зависеть от деталей реализации, обратное верно

Динамическая идентификация типов времени выполнения (RTTI)



Динамическая идентификация типов времени выполнения ([англ. Real-Time Type Identification](#)) обеспечивает **специальную поддержку полиморфизма** и позволяет программе **узнать реальный производный тип объекта**, адресуемого по ссылке или по указателю на базовый класс. Поддержка RTTI в C++ реализована двумя операциями:

- операция [dynamic_cast](#) поддерживает преобразование типов времени выполнения;
- операция [typeid](#) идентифицирует реальный тип выражения.

Операции RTTI — это события времени выполнения для классов с виртуальными функциями и события времени компиляции для остальных типов. Исследование RTTI-информации полезно, в частности, при решении задач системного программирования.

Операция `dynamic_cast`



Встроенная унарная операция `dynamic_cast` языка C++ позволяет:

- **безопасно трансформировать указатель** на базовый класс в указатель на производный класс (с возвратом нулевого указателя при невозможности выполнения трансформации);
- **преобразовывать леводопустимые значения**, ссылающиеся на базовый класс, в ссылки на производный класс (с возбуждением исключения `bad_cast` при ошибке).

Единственным операндом `dynamic_cast` должен являться тип класса, в котором имеется хотя бы один виртуальный метод.

Операция `dynamic_cast`: пример (указатели)



```
// классы Alpha и Beta образуют полиморфную  
// иерархию, в которой класс Beta открыто  
// наследует классу Alpha
```

```
Alpha *a1 = new Beta;
```

```
if (Beta *bt = dynamic_cast<Beta*>(a1)) {  
    /* успешно */  
}  
else {  
    /* неуспешно */  
}
```

Операция `dynamic_cast`: пример (ссылки)



```
// классы Alpha и Beta образуют полиморфную
// иерархию, в которой класс Beta открыто
// наследует классу Alpha

#include <typeinfo> // для std::bad_cast

void foo(Alpha &a1)
{
    /* ... */
    try {
        Beta &bt = dynamic_cast<Beta&>(a1)
    }
    catch (std::bad_cast) {
        /* ... */
    }
}
```

Операция typeid (1 / 2)



Встроенная унарная операция `typeid`:

- позволяет **установить фактический тип** выражения-операнда;
- может использоваться с выражениями и именами любых типов (**включая выражения встроенных типов и константы**).

Если операнд `typeid` принадлежит типу класса с одной и более виртуальными функциями (**не указателю на него!**), результат `typeid` может не совпадать с типом самого выражения.

Операция `typeid` имеет тип (возвращает значение типа) `type_info` и требует подключения заголовочного файла `<typeinfo>`.

Операция typeid (2 / 2)



Реализация класса `type_info` зависит от компилятора, но в общем и целом позволяет получить результат в виде неизменяемой C-строки (`const char*`), присваивать объекты `type_info` друг другу (`operator =`), а также сравнивать их на равенство и неравенство (`operator ==`, `operator !=`).

```
#include <typeinfo> // для type_info

Alpha *a1 = new Alpha;

if (typeid(a1) == typeid(Alpha*)) /* ... */
if (typeid(*a1) == typeid(Alpha)) /* ... */
```

- Глубина цепочки наследования не увеличивает затраты времени и не ограничивает доступ к унаследованным членам базовых классов.
- Вызов виртуальной функции в большинстве случаев не менее эффективен, чем косвенный вызов функции по указателю на нее.
- При использовании встроенных конструкторов глубина иерархии наследования почти не влияет на производительность.
- Отмена действия механизма виртуализации, как правило, необходима по соображениям повышения эффективности.

Постановка задачи

- Реализовать UML-модель как полиморфную иерархию классов с шаблоном (специализированным шаблоном) класса.
- Дополнить результат иерархией классов исключительных ситуаций, смоделировать каждую ситуацию и обеспечить ее корректную обработку.
- **Цель** — перевести архитектурное описание проекта на языке UML в исходный программный код на языке C++, отвечающий заданным критериям полноты и качества результата.



Спасибо за внимание

Алексей Петров