# Лекция 9. Наследование и композиция

Одним из основных принципов программирования является возможность многократного использования созданного программного кода.   
 В объектно-ориентированных ЯП эта возможность решается на уровне классов. Чтобы заново использовать существующий код, программист создает новый класс, но не «с нуля», а на базе готовых классов, построенных и отлаженных кем-то другим. Проблема заключается в том, чтобы задействовать классы, не нарушая работоспособности существующего кода.

На этой лекции мы рассмотрим два способа решения задачи.   
 Первый способ прост и прямолинеен: объекты существующих классов создаются внутри новых классов. Это называется композицией, поскольку новый (композитный) класс «собирается» из объектов существующих классов.   
 Второй способ не столь тривиален. Новый класс создается как подтип существующего класса. Программист буквально берет существующий класс как «заготовку» и добавляет в него новый код, не изменяя существующего класса. Этот прием называется наследованием, и большая часть работы по его реализации выполняется компилятором.  
 Композиция и наследование обладают сходным поведением, и это логично: оба способа предназначены для создания новых типов на базе существующих. В первой части лекции мы рассмотрим синтаксис, приемы использования и сравним результаты применения композиции и наследования. Во второй части лекции мы используем композицию и наследование в разработке библиотеки классов для макета СУБД.

## 1. Синтаксис

### 1.1. Синтаксис композиции

Вообще говоря, мы уже неоднократно применяли композицию для создания новых классов. Классы в приводимых примерах в основном строились из встроенных   
типов и строк (библиотечный класс). Оказывается, для пользовательских типов композиция реализуется почти так же просто.

Допустим, следующий класс почему-либо представляет ценность для нас:

//L9:Useful.h

//Класс, который должен использоваться многократно

#ifndef USEFUL\_H

#define USEFUL\_H

class X {

int i ;

public:

X(){i = 0;}

void set(int ii ) {i = ii;}

int read() const { return i;}

int permute() {return i = i \* 47;}

};

#endif // USEFUL\_H

Переменные класса объявлены закрытыми, поэтому мы можем абсолютно безопасно включить объект типа X в новый класс как открытый (public) объект, что значительно упрощает интерфейс класса:

//Composition.срр

#include "Useful.h"

class Y {

int i;

public:

X x; // Внутренний объект

Y() {i=0;}

void f(int ii) {i = ii;}

int g() const {return i;}

};

int main(){

Y y;

y.f(47);

y.x.set(37); // Обращение к внутреннему объекту

}

Обращение к функциям внутреннего объекта (в дальнейшем мы будем называть его подобъектом) просто требует дополнительного уточнения имени объекта. На практике внутренние объекты чаще объявляются закрытыми и, таким образом, становятся частью базовой реализации (это означает, что при желании реализацию можно изменить). Функции открытого интерфейса нового класса работают с внутренним объектом, но не обязательно повторяют его интерфейс:

//L9:Composition2.срр

// Закрытые внутренние объекты

#include "..\\UsefulClass\\Useful.h"

class Y {

int i;

X x; // Внутренний объект

public:

Y() {i = 0;}

void f(int ii) {i = ii; x.set(ii);}

int g() const {return i \* x.read();}

void permute() { x.permute();}

};

int main() {

Y y;

y.f(47);

y.permute();

}

Функция внутреннего объекта permute() переносится в интерфейс композитного класса, а остальные функции класса X используются в функциях Y.

### 1.2. Синтаксис наследования

Синтаксис композиции очевиден, а для наследования применяется совершенно иная синтаксическая форма. Используя наследование, вы фактически говорите: «Новый класс похож на такой-то существующий класс». Для этого в программе, как обычно, задается имя класса, но перед открывающей фигурной скобкой через двоеточие указывается имя базового класса (или базовых классов, разделенных запятыми, в случае множественного наследования). После этого в новый класс автоматически включаются все переменные и функции базового класса. Пример:

//L9:Inheritance.срр

// Простое наследование

#include "..\\UsefulClass\\Useful.h"

#include <iostream>

using namespace std;

class Y : public X {

int i; // Переменная i отлична от i класса X

public:

Y() {i = 0;}

int change() {

i = permute(); // Вызов по другому имени

return i;

}

void set(int ii) {

i = ii;

X::set(ii); // Вызов по тому же имени

}

};

Класс Y наследует от X; это означает, что Y содержит все переменные класса X, а также его функции. Фактически Y содержит подобъект X точно так же, как если бы вы создали объект X внутри Y, вместо того чтобы наследовать от X.

Все закрытые элементы X остаются закрытыми в классе Y; иначе говоря, тот факт, что Y наследует от X, не означает, что Y может нарушить работу механизма защиты. Закрытые элементы X никуда не исчезли, они по-прежнему находятся в памяти, — просто к ним нельзя обращаться напрямую.

Из функции main() видно, что элементы данных Y объединяются с данными X, потому что размер, возвращаемый функцией sizeof(Y), вдвое больше размера, возвращаемого функцией sizeof(X).

Обратите внимание: перед именем базового класса находится ключевое слово public. При наследовании по умолчанию используется уровень доступа private. Если бы перед именем базового класса не было ключевого слова public, то все открытые члены базового класса стали бы закрытыми в производном классе. Такое поведение почти всегда нежелательно: обычно открытые члены базового класса должны оставаться открытыми в производном классе. Для этого при наследовании указывается ключевое слово public.

В функции change() вызывается функция permute() базового класса. Производный класс может напрямую вызывать все открытые функции базового класса.

Функция set() производного класса переопределяет функцию set() базового класса. Иначе говоря, при вызове функций read() и permute() для объекта типа Y будут вызваны версии базового класса (в функции main() показано, как это происходит). Но если вызвать для объекта Y функцию set(), будет вызвана переопределенная версия. Следовательно, если вас не устраивает то, как работает унаследованная функция, вы можете изменить ее, а также добавить новые функции, такие как change().

Однако при переопределении функций все же должна сохраняться возможность вызова их версий для базового класса. Если внутри *set()* просто вызвать *set()*, то будет вызвана локальная версия функции, то есть возникнет рекурсия. Чтобы вызвать версию функции базового класса, необходимо явно задать имя базового класса и оператор уточнения области видимости: *X::set().*

#### 1.2.1. Список инициализирующих значений конструктора

Вы уже знаете, какая важная роль в C++ отводится гарантированной инициализации; это в полной мере относится к композиции и наследованию. При создании объекта компилятор гарантирует вызов конструктора для всех его подобъектов. До настоящего момента все подобъекты имели конструкторы по умолчанию, которые автоматически вызывались компилятором. Но что, если подобъекты не имеют конструкторов по умолчанию или вы захотите изменить аргумент конструктора? Возникает проблема, поскольку конструктор нового класса не имеет доступа к закрытым данным подобъекта, а значит, не может инициализировать их напрямую. Однако проблема решается просто: следует вызвать конструктор подобъекта. В C++ для этого предусмотрен специальный синтаксис, называемый списком инициализирующих значений конструктора. Синтаксис списка инициализирующих значений конструктора имитирует синтаксис наследования. При наследовании базовые классы перечисляются после двоеточия, но до открывающей фигурной скобки тела класса. В списке инициализирующих значений вызовы конструкторов подобъектов размещаются после списка аргументов конструктора и двоеточия, но перед открывающей фигурной скобкой тела функции. Для класса МуТуре, наследующего от *Bar*, это может выглядеть так (если у *Bar* имеется конструктор, вызываемый с одним аргументом типа *int*):

МуТуре::МуТуре(int i) : Bar(i) { // ...

#### 1.2.2. Инициализация объектов внутри класса

Оказывается, синтаксис инициализации объектов также подходит и для композиции. Правда, в этом случае указываются имена объектов вместо имен классов. Если список инициализирующих значений содержит более одного вызова конструктора, то вызовы разделяются запятыми:

МуТуре2::MyType2(int i) : Bar(i), m(i+l) { // ...

Так начинается конструктор класса *МуТуре2*, наследующего от *Bar* и содержащего переменную *m*. Обратите внимание: хотя в списке инициализирующих значений конструктора присутствует тип базового класса, для внутреннего объекта указывается его идентификатор.

#### 1.2.3. Встроенные типы в списке инициализирующих значений

Список инициализирующих значений конструктора позволяет явно вызывать конструкторы для внутренних объектов. Более того, вызвать эти конструкторы другим способом нельзя. Идея заключается в том, что все конструкторы внутренних объектов должны быть вызваны до передачи управления в тело конструктора нового класса. В этом случае любые вызовы функций внутренних объектов всегда будут относиться к инициализированным объектам. Невозможно войти за открывающую фигурную скобку конструктора без вызова какого-либо конструктора для каждого внутреннего объекта и объектов базовых классов, даже если компилятору приходится скрыто вызывать конструктор по умолчанию. Это дополнительно подкрепляет гарантии C++ относительно того, что ни один объект (или его часть) не может использоваться в программе без предварительного вызова его конструктора.

Гарантия инициализации внутренних объектов к моменту достижения открывающей фигурной скобки конструктора также оказывается удобной для программиста. Достигнув этой точки, можно предположить, что все внутренние объекты были успешно инициализированы, и сосредоточиться на конкретных задачах, выполняемых конструктором. Но тут кроется подвох: что делать с объектами встроенных типов, не имеющих конструкторов?

Чтобы синтаксис стал более последовательным, было решено интерпретировать встроенные типы так, словно у них имеется конструктор с единственным аргументом, тип которого соответствует типу инициализируемой переменной. Таким образом, возможна инициализация вида

class X {

int i ;  
 float f;  
 char с;   
 char\* s;   
public: X() : i(7), f (1.4), c('x'), s("howdy") {}

};

int main() {  
 X x;  
 int i(100); // Применение к обычному определению  
 int\* ip = new int(47);   
}

Все эти «вызовы псевдо конструкторов» просто присваивают значения соответствующим переменным. Этот прием удобен и считается хорошим стилем программирования, поэтому вы часто будете встречать его в программах. Синтаксис «псевдо конструкторов» может использоваться даже при создании переменных встроенных типов за пределами классов.

### 1.3. Защищенность

После знакомства с наследованием становится понятным ключевое слово protected.   
В идеале закрытые члены класса всегда должны оставаться закрытыми, но в реальных проектах иногда бывает удобно скрыть функцию или переменную класса от внешнего мира, но сохранить доступ к ней из производных классов. Ключевое слово protected отдает дань прагматизму; оно означает: «Закрытый с точки зрения пользователей класса, но доступный для всех классов, производных от данного». Переменные класса лучше всего оставлять закрытыми — тем самым вы сохраняете возможность менять базовую реализацию по своему усмотрению. Доступ из производных классов контролируется при помощи защищенных функций класса.

### 1.4. Функции, которые не наследуются автоматически

Не все функции автоматически наследуются производным классом от базового. Конструкторы и деструкторы участвуют в создании и уничтожении объекта, поэтому они умеют работать только с элементами объекта, относящимися к его собственному классу. Для остальных элементов приходится вызывать конструкторы и деструкторы классов, расположенных ниже в иерархии наследования. Из-за этого конструкторы и деструкторы не наследуются, а явно определяются для каждого производного класса.

Функция operator= тоже не наследуется, поскольку выполняемые ею действия сходны с конструированием. Другими словами, даже если вы знаете, как присвоить значения переменных объекта, находящегося справа от оператора =, переменным объекта, находящимся слева, это вовсе не означает, что смысл присваивания не изменится в результате наследования. При наследовании эти функции генерируются компилятором, если они не были определены явно (причем для того, чтобы компилятор мог сгенерировать конструктор по умолчанию и копирующий конструктор, в классе не может быть определено ни одного конструктора). Сгенерированные конструкторы используют инициализацию, а сгенерированная функция operator= использует присваивание на уровне переменных.

# 2. Объединение композиции с наследованием

Конечно, композиция может использоваться совместно с наследованием. В следующем примере продемонстрировано создание более сложного класса с применением обоих механизмов:

Combined.срр // Наследование и композиция

class А {

int i ;

public:

A(int ii) : i(ii) {}

~A() {}

void f()const {}

};

class В {

int i ;

public:

B(int ii) : i(ii) {}

~B() {}

void f() const {}

}

class С : public В {

A a;

public:

C(int ii) : B(ii), a(ii) {}

~С() {} // Вызывает ~А() и ~В()

void f() const { // Переопределение

a.f();

В::f();

}

};

int main() {

С с(47);

}

Класс С наследует от В и содержит внутренний объект типа А. Из листинга видно, что список инициализирующих значений конструктора содержит вызовы конструкторов как базового класса, так и внутреннего объекта.  
 Функция C::f() переопределяет унаследованную функцию B::f(), а также вызывает версию базового класса. Кроме того, она вызывает a.f(). Следует помнить, что переопределение функций может выполняться только путем наследования; при использовании внутренних объектов можно работать с открытым интерфейсом объекта, но переопределять его нельзя. Кроме того, если функция C::f() не определена, то вызов f() для объекта класса С не приведет к вызову a.f(); вместо этого будет вызвана функция В::f().

# 3. Автоматический вызов конструкторов и деструкторов

Хотя необходимость в явном вызове конструкторов в списке инициализирующих значений возникает достаточно часто, вам никогда не понадобится явно вызывать деструкторы, поскольку в каждом классе может быть только один деструктор, и он вызывается без аргументов. Тем не менее компилятор по-прежнему гарантирует вызов всех деструкторов, то есть всех деструкторов во всей иерархии наследования, начиная с последних классов в цепочке наследования и до корневого базового класса.

Стоит особо подчеркнуть, что конструкторы и деструкторы принципиально отличаются от других функций: они вызываются все от начала и до конца иерархии, тогда как при вызове обычной функции вызывается только эта функция, но не ее версии из базовых классов. Если вы хотите вызвать базовую версию обычной *переопределенной* функции, это придется сделать явно.

## 3.1. Порядок вызова конструкторов и деструкторов

Интересно разобраться в том, в каком порядке вызываются конструкторы и деструкторы при большом количестве подобъектов. Следующий пример позволяет точно понять, что при этом происходит:

//L9:Order.срр

// Порядок вызова конструкторов и деструкторов

#include <fstream>

using namespace std;

ofstream out("order.out");

template <class T>class T{

public:

int m;

T(int n):m(n) {out <<typeid(this).name()<< " constructor\n";}

~T(){out <<typeid(this).name()<< " destructor\n";}

};

class Base1;

class Member1;

class Member2;

class Member3;

class Member4;

class Derived1 : public T<Base1> {

T<Member1> m1;

T<Member2> m2;

public:

Derived1(int n1=1,int n2=2) : m2(n1), m1(n2), T<Base1>(3) {

out << "Derived1 constructor\n";

}

~Derived1() {

out <<" Derived1 destructor\n";

}

};

class Derived2 :public Derived1 {

T<Member3> m3;

T<Member4> m4;

public: Derived2() : m3(1), Derived1(2), m4(3) {

out << "Derived2 constructor\n";

}

~Derived2() {

out << "Derived2 destructor\n";

}

};

int main() {

Derived2 d2;

}

//================================================================

Оператор *typeid* .  
В языке C++ оператор *typeid* возвращает ссылку на объект *type\_info*, описывающий тип объекта, адрес которого указывается в круглых скобках.   
Общая форма записи оператора *typeid* такова:   
typeid(&*объект*)

Оператор *typeid* поддерживает в языке C++ возможность идентификации динамической информации о типе (RTTI).

Класс *type\_info* определяет следующие открытые члены.

bool operator == (const type\_info &ob) const;

bool operator != (const type\_info &ob) const;

const char \*name() const;

Перегруженные операторы == и != служат для сравнения типов. 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.

//=================================================================

Сначала мы создаем объект ofstream для записи всех результатов в файл. Затем для уменьшения объема работы мы определяем шаблон для построения классов, который затем будет использоваться при наследовании и композиции.

Каждый конструктор и деструктор выводит сообщение о своем вызове в файл. Обратите внимание: конструкторы не являются конструкторами по умолчанию; у них имеется аргумент типа int. Этот аргумент используется ещё и для того, чтобы заставить нас явно вызывать *перегруженные* конструкторы в списке инициализирующих значений.

Результат выполнения программы:   
Base1 constructor   
Memberl constructor   
Member2 constructor   
Derivedl constructor   
Member3 constructor   
Member4 constructor   
Derived2 constructor   
Derived2 destructor   
Member4 destructor   
Member3 destructor   
Derivedl destructor   
Member2 destructor   
Memberl destructor   
Member4 destructor   
Basel destructor

Видно, что конструирование начинается с корня иерархии классов, и на каждом уровне сначала вызывается конструктор базового класса, а затем конструкторы внутренних объектов. Порядок вызова деструкторов в точности противоположен порядку вызова конструкторов; это важно из-за потенциальных зависимостей (в конструкторе или деструкторе производного класса должно действовать допущение о том, что подобъект базового класса может использоваться и он уже сконструирован или еще не уничтожен). Также интересно заметить, что порядок вызова конструкторов для внутренних объектов совершенно не зависит от порядка вызовов в списке инициализирующих значений. Он полностью определяется порядком объявления внутренних объектов в классе. Если бы порядок вызова конструкторов можно было изменить при помощи списка инициализирующих значений, то у двух разных конструкторов могли бы быть две разные последовательности вызовов, а деструктор (он в классе единственный) не знал бы, как правильно выстроить обратный порядок вызовов при уничтожении объектов. В итоге возникла бы проблема зависимости.

Конечной целью наследования является поддержка полиморфизма, а изменение списка параметров или типа возвращаемого значения при перегрузке функции приводит к фактическому изменению интерфейса базового класса. Если вы именно этого и добивались, значит, наследование применяется в основном для реализации многократного использования кода, а не для сохранения единого интерфейса базового класса (важного аспекта полиморфизма). Как правило, такой вариант наследования означает, что класс общего назначения специализируется для конкретного случая, а для таких задач обычно рекомендуется применять композицию.

# 4. Пошаговая разработка при наследовании и композиции

К достоинствам наследования и композиции следует отнести естественность пошаговой разработки, то есть включения нового кода в проект с гарантированным сохранением работоспособности существующего кода. Все ошибки локализуются в новом коде. Создавая новые классы на базе готовых работоспособных классов посредством наследования или композиции и добавляя в них новые функции, а также переопределяя существующие функции при наследовании, вы оставляете старый код (который может использоваться еще кем-то) в прежнем состоянии без изменений и ошибок. Если в программе возникает ошибка, вы знаете, что она находится в новом коде, поэтому поиск ошибок выполняется гораздо быстрее и требует меньших усилий, чем при модификации существующего кода.   
При наличии классов логические функции программы очень четко разделяются. Чтобы воспользоваться готовыми функциями классов, вам даже не понадобятся их исходные тексты — достаточно заголовков с описанием класса и объектных или библиотечных файлов с откомпилированными функциями (это относится как к наследованию, так и к композиции).   
Важно понимать, что разработка программы является *пошаговым процессом*. Каким бы долгим ни был предварительный анализ, вы все равно не будете знать всех ответов, пока не начнете работу над проектом. Работа пойдет гораздо успешней и быстрее принесет результаты, если вы начнете «выращивать» проект как живое (работоспособное) существо, а не строить его сразу целиком.

Базовая реализация класса должна быть скрыта от пользователей, чтобы позднее разработчик мог внести в нее изменения. Например, такие изменения вносятся для повышения эффективности, потому что разработчик нашел другой подход к решению проблемы или у него появился новый класс, который должен использоваться в реализации. Любые решения, нарушающие закрытость базовой реализации, лишь делают язык менее гибким.  
Открытая для пользователей часть класса, его *интерфейс, должен оставаться неизменным*.

# 4. Выбор между композицией и наследованием

Как композиция, так и наследование приводят к созданию подобъектов в новом классе. В обоих случаях эти подобъекты конструируются в списке инициализирующих значений конструктора. Возникает вопрос: чем различаются эти два способа и когда следует применять каждый из них? Композиция обычно применяется тогда, когда новый класс должен поддерживать некоторые возможности существующего класса, но не его интерфейс. Иначе говоря, объект внедряется для реализации функциональности нового класса, но пользователь класса работает с ним не через интерфейс исходного класса, а через интерфейс, определенный программистом. Обычно это делается по классическому рецепту включения закрытых объектов существующих классов в новый класс. Тем не менее в отдельных случаях бывает разумно предоставить пользователю класса прямой доступ к внутренним объектам, для чего внутренний объект объявляется открытым. Внутренние объекты сами контролируют доступ к себе, поэтому такое объявление безопасно. С другой стороны, когда пользователь знает, из каких частей состоит объект, ему бывает проще понять его интерфейс. Хорошим примером служит класс Саг («автомобиль»):

class Car :public Vehicle {// автомобиль – это транспортное средство

public:

Engine engine; // У автомобиля двигатель

Wheel wheel[4]; // У автомобиля четыре колеса

Door left, right; // У автомобиля две двери

};

Объявление внутренних объектов открытыми помогает прикладному программисту разобраться в том, как использовать этот класс, и упрощает программирование для создателя класса. Если немного подумать, становится ясно, что нет смысла создавать Саг посредством композиции с включением объекта Vehicle («транспортное средство»), поскольку автомобиль не содержит транспортное средство, а является им.

Но что делать, если в новый класс должна быть включена вся функциональность прежнего класса? Такая ситуация называется выделением подтипа, потому что мы создаем новый тип на базе существующего типа. При этом новый тип имеет точно такой же интерфейс, как существующий тип (а также содержит новые функции, которые вы захотите в него добавить), чтобы он всегда мог использоваться вместо существующего типа. В таких ситуациях уместно применить наследование.

Вывод:

- связи типа «а является частным случаем b» обычно реализуются путем наследования;

- связи «а содержит b» обычно реализуются путем композиции.