Курс «C++. Программирование на языке высокого уровня»

Павловская Т.А.

Лекция 6. Наследование. Шаблоны классов

Простое и множественное наследование классов. Виртуальные методы. Абстрактные классы. Создание и использование шаблонов классов.

Наследование

Наследование является мощнейшим инструментом ООП и применяется для следующих взаимосвязанных целей:

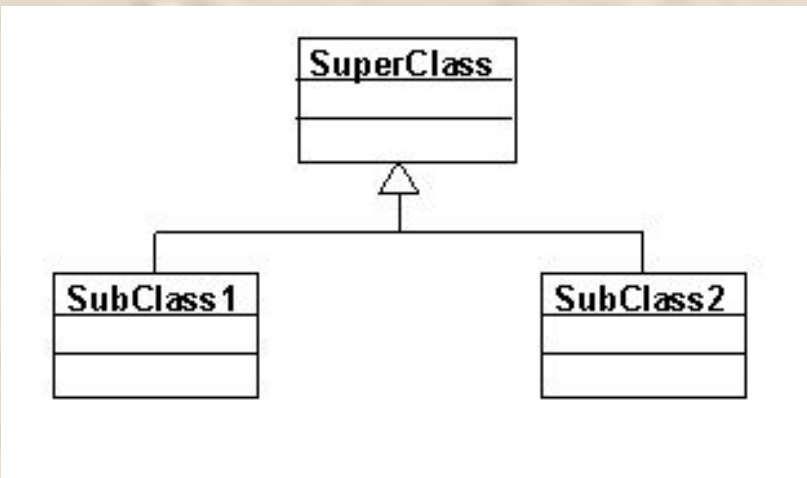
- исключения из программы повторяющихся фрагментов кода;
- упрощения модификации программы;
- упрощения создания новых программ на основе существующих.
- Кроме того, наследование является единственной возможностью использовать объекты, исходный код которых недоступен, но в которые требуется внести изменения.

Синтаксис наследования

Ключи доступа

class имя : [private | protected | public] базовый_класс

{ тело класса };




```
class A { ... };
class B { ... };
class C { ... };
class D: A, protected B, public C
{ ... };
```

- В наследнике можно описывать новые поля и методы и переопределять существующие методы. Переопределять методы можно несколькими способами.
- Если какой-либо метод в потомке должен работать совершенно по-другому, чем в предке, метод описывается в потомке заново. При этом он может иметь другой набор аргументов.
- Если требуется внести добавления в метод предка, то в соответствующем методе потомка наряду с описанием дополнительных действий выполняется вызов метода предка с помощью операции доступа к области видимости.
- Если в программе планируется работать одновременно с различными типами объектов иерархии или планируется добавление в иерархию новых объектов, метод объявляется как виртуальный с помощью ключевого слова virtual. Все виртуальные методы иерархии с одним и тем же именем должны иметь одинаковый список аргументов.

Правила наследования

Ключ доступа	Спецификатор в базовом классе	Доступ в производном классе
private	private	нет
	protected	private
	public	private
protected	private	нет
	protected	protected
	public	protected
public	private	нет
	protected	protected
	public	public

Иными словами:

- private элементы базового класса в производном классе недоступны вне зависимости от ключа. Обращение к ним может осуществляться только через методы базового класса.
- Элементы protected при наследовании с ключом private становятся в производном классе private, в остальных случаях права доступа к ним не изменяются.
- Доступ к элементам public при наследовании становится соответствующим ключу доступа.

Если базовый класс наследуется с ключом private, можно выборочно сделать некоторые его элементы доступными в производном классе:

```
class Base{
   ...
   public: void f();
};
class Derived : private Base{
   ...
   public: Base::void f();
};
```

Простое наследование

```
class daemon : public monstr{
        int brain;
    public:
        // ------------ Конструкторы:
        daemon(int br = 10){brain = br;};
        daemon(color sk) : monstr (sk) {brain = 10;}
        daemon(char * nam) : monstr (nam) {brain = 10;}
        daemon(daemon &M) : monstr (M) {brain = M.brain;}
```

Если конструктор базового класса требует указания параметров, он должен быть явным образом вызван в конструкторе производного класса в списке инициализации

Порядок вызова конструкторов

Конструкторы не наследуются, поэтому производный класс должен иметь собственные конструкторы. Порядок вызова конструкторов:

- Если в конструкторе потомка явный вызов конструктора предка отсутствует, *автоматически вызывается конст-руктор предка по умолчанию*.
- Для иерархии, состоящей из нескольких уровней, конструкторы предков вызываются начиная с самого верхнего уровня. После этого выполняются конструкторы тех элементов класса, которые являются объектами, в порядке их объявления в классе, а затем исполняется конструктор класса.
- В случае нескольких предков их конструкторы вызываются в порядке объявления.

Операция присваивания

```
const daemon& operator = (daemon &M){
        if (&M == this) return *this;
        brain = M.brain;
        monstr::operator = (M);
        return *this;
```

Поля, унаследованные из класса monstr, недоступны функциям производного класса, поскольку они определены в базовом классе как private.

Производный класс может не только дополнять, но и корректировать поведение базового класса. Переопределять в производном классе рекомендуется только виртуальные методы

Наследование деструкторов

- Деструкторы не наследуются. Если деструктор в производном классе не описан, он формируется автоматически и вызывает деструкторы всех базовых классов.
- В деструкторе производного класса не требуется явно вызывать деструкторы базовых классов, это будет сделано автоматически.

Для иерархии, состоящей из нескольких уровней, деструкторы вызываются в порядке, строго обратном вызову конструкторов: сначала вызывается деструктор класса, затем — деструкторы элементов класса, а потом деструктор базового класса.

Раннее связывание

Описывается указатель на базовый класс:

```
monstr *p;
```

Указатель ссылается на объект производного класса:

```
p = new daemon;
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта:

```
p->draw(1, 1, 1, 1); // Метод monstr
```

Можно использовать явное преобразование типа указателя:

```
(daemon * p)->draw(1, 1, 1, 1);
```

Виртуальные методы

```
virtual void draw(int x, int y, int scale, int position);
```

```
monstr *r, *p;
// Создается объект класса monstr
r = new monstr;
// Создается объект класса daemon
p = new daemon;
// Вызывается метод monstr::draw
r->draw(1, 1, 1, 1);
// Вызывается метод daemon::draw
p->draw(1, 1, 1, 1);
//    Обход механизма виртуальных методов
p->monstr::draw(1, 1, 1, 1);
```

vtbl

vptr

Описание и использование виртуальных методов

- Если в предке метод определен как виртуальный, метод, определенный в потомке *с тем же именем и набором параметров*, автоматически становится виртуальным, а *с отличающимся набором параметров* — обычным.
- Виртуальные методы *наследуются*, то есть переопределять их в потомке требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя.
- Если виртуальный метод переопределен в потомке, объекты этого класса могут получить доступ к методу предка с помощью операции доступа к области видимости.
- Виртуальный метод не может объявляться с модификатором static, но может быть объявлен как дружественный.
- Если в классе вводится объявление виртуального метода, он должен быть определен хотя бы как чисто виртуальный.

Чисто виртуальные методы

- содержит признак = 0 вместо тела:

```
virtual void f(int) = 0;
```

- должен переопределяться в производном классе.

Класс, содержащий хотя бы один чисто виртуальный метод, называется *абстрактным*.

- абстрактный класс нельзя использовать при явном приведении типов, для описания типа параметра и типа возвращаемого функцией значения;
- допускается объявлять указатели и ссылки на абстрактный класс, если при инициализации не требуется создавать временный объект;
- если класс, производный от абстрактного, не определяет все чисто виртуальные функции, он также является абстрактным.

Множественное наследование

```
class monstr{
        public: int get_health(); ...
};
class hero{
        public: int get_health();...
};
class ostrich: public monstr, public hero { ... };

int main(){
        ostrich A;
        cout << A.monstr::get_health();
        cout << A.hero::get_health();
}
```

```
class monstr{
        ...
};
class daemon: virtual public monstr{
        ...
};
class lady: virtual public monstr{
        ...
};
class baby: public daemon, public lady{
        ...
};
```

Рекомендации

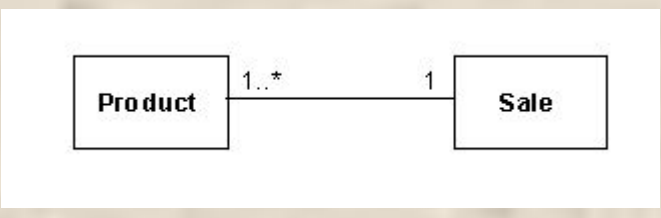
Множественное наследование применяется для того, чтобы обеспечить производный класс свойствами двух или более базовых.

Чаще всего один из этих классов является основным, а другие обеспечивают некоторые дополнительные свойства, поэтому они называются **классами подмешивания**.

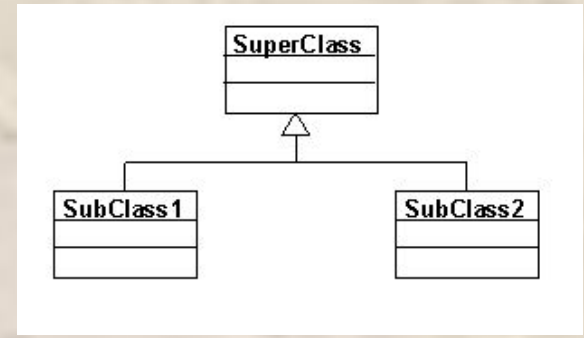
По возможности классы подмешивания должны быть **виртуальными** и создаваться с помощью конструкторов без параметров, что позволяет избежать многих проблем, возникающих, когда у базовых классов есть общий предок.

Виды отношений между классами

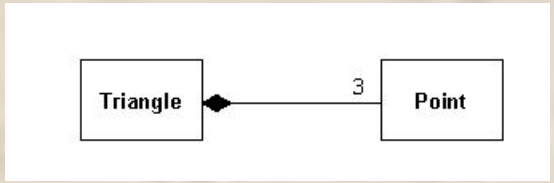
- ассоциация (два класса концептуально взаимодействуют друг с другом);

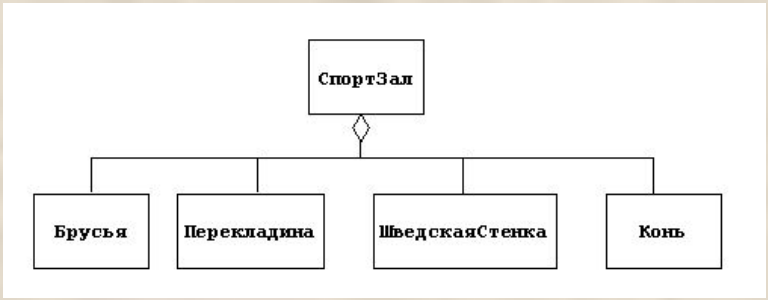


- наследование (отношение обобщения, «is a»);

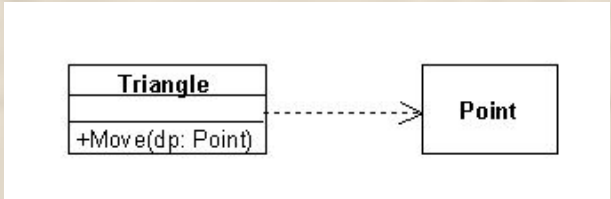


- агрегация (отношение целое/часть, «has a»);
  - строгая (композиция)
  - нестрогая (по ссылке)





- зависимость (отношение использования)



Шаблоны классов

Параметризованный класс создает семейство родственных классов, которые можно применять к любому типу данных, передаваемому в качестве параметра. Наиболее широкое применение шаблоны находят при создании *контейнерных классов*.

Создание шаблонов классов

```
template <описание_параметров_шаблона>
определение_класса;
```

Параметры шаблона перечисляются через запятую. В качестве параметров могут использоваться типы, шаблоны и переменные.

Параметры шаблона - типы

Типы могут быть как стандартными, так и определенными пользователем. Для их описания используется ключевое слово class. Внутри шаблона параметр типа может применяться в любом месте, где допустимо использовать спецификацию типа, например:

```
template <class Data> class List{
    class Node{
        public:
            Data d;
            Node *next;
            Node *prev;
            Node(Data dat = 0){
                d = dat; next = 0; prev = 0;
            }
    };
```

Значения параметров по умолчанию

Для любых параметров шаблона могут быть заданы *значения по умолчанию*, например:

```
template<class T> class mar { /*...*/ };
template<class K, class V, template<class T> class C = mar>
  class Map{
      C<K> key;
      C<V> value;
  ...
  };
```

Параметр можно использовать при описании следующих за ним, например:

```
template<class T, T* p, class U = T> class X { /* ... */ };
```

Методы шаблона класса

Методы шаблона класса автоматически становятся шаблонами функций. Если метод описывается вне шаблона, его заголовок должен иметь следующие элементы:

```
template <описание_параметров_шаблона>
   возвр_тип имя_класса <параметры_шаблона>::
      имя_функции (список_параметров функции)
```

Синтаксис описания методов шаблона на примере:

```
template <class Data> void List<Data>::print()
   { /* тело функции */ }
```

Здесь <class Data> – описание параметра шаблона, void – тип возвращаемого функцией значения, List – имя класса, <Data> – параметр шаблона, print – имя функции без параметров.

Правила описания шаблонов

- Локальные классы не могут содержать шаблоны в качестве своих элементов.
- Шаблоны методов не могут быть виртуальными.
- Шаблоны классов могут содержать статические элементы, дружественные функции и классы.
- Шаблоны могут быть производными как от шаблонов, так и от обычных классов, а также являться базовыми и для шаблонов, и для обычных классов.
- Внутри шаблона нельзя определять friend-шаблоны.

Параметры шаблона - переменные

Переменные могут быть целого или перечисляемого типа, а также указателями или ссылками на объект или функцию. В теле шаблона они могут применяться в любом месте, где допустимо использовать константное выражение:

```
template <class Type, int kol>     class Block{
        public:
            Block(){p = new Type [kol];}
            ~Block(){delete [] p;}
            operator Type *();
        protected:    Type * p;
};
template <class Type, int kol>
    Block <Type, kol>:: operator Type *(){ return p;}
```

Пример параметра–указателя

```
void f1() { cout << "I am f1()." << endl; }
void f2() { cout << "I am f2()." << endl; }

template<void (*pf)()> struct A
{ void Show() { pf();}
};
int main() {
    A<&f1> aa;
    aa.Show(); // вывод: I am f1().
    A<&f2> ab;
    ab.Show(); // вывод: I am f2().
    return 0;
}
```

Использование шаблонов классов

При описании объекта после имени шаблона в угловых скобках перечисляются его аргументы:

имя_шаблона <аргументы>

имя_объекта [(параметры_конструктора)];

Аргументы должны соответствовать параметрам шаблона. Имя шаблона вместе с аргументами можно воспринимать как уточненное имя класса. Примеры создания объектов по шаблонам, описанным в предыдущем разделе:

```
List <int> List_int;
List <double> List_double;
List <monstr> List_monstr;
Block <char, 128> buf;
Block <monstr, 100> stado;
```

Использование шаблонов классов

При использовании параметров шаблона по умолчанию список аргументов может оказаться пустым, при этом угловые скобки опускать нельзя:

```
template<class T = char> class String;
String<>* p;
```

На месте формальных параметров, являющихся переменными целого типа, должны стоять константные выражения.

После создания объектов с помощью шаблона с ними можно работать так же, как с объектами обычных классов, например:

```
for (int i = 1; i<10; i++)List_double.add(i * 0.08);
List_double.print();
//--------------------------------
for (int i = 1; i<10; i++)List_monstr.add(i);
List_monstr.print();
```

Для упрощения использования шаблонов классов можно применить переименование типов с помощью typedef:

```
typedef List <double> Ldbl;
Ldbl List_double;
```

Организация исходного кода

Принято размещать *все определение* шаблонного класса в заголовочном файле и подключать его к нужным файлам с помощью директивы #include. Для предотвращения повторного включения этого файла используйте «стражи включения»

```
// Point.h
#ifndef POINT_H
#define POINT_H
template <class T> class Point {
public:
    Point(T _x = 0, T _y = 0) : x(_x), y(_y) {}
    void Show() const;
private:
    T x, y;
};
template <class T> void Point<T>::Show() const {
    cout << "  (" << x << ", " << y << ")" << endl;
}
#endif /* POINT_H */
```

```
// Main.cpp
#include <iostream>
#include "Point.h"
using namespace std;
int main() {
    Point<double> p1;              // 1
    Point<double> p2(7.32, -2.6);   // 2
    p1.Show(); p2.Show();
    Point<int> p3(13, 15);          // 3
    Point<short> p4(17, 21);        // 4
    p3.Show(); p4.Show();
    return 0;
}
```

Специализация шаблонов классов

-методов
-классов
    -спец-я всего класса
    -частичная специализация

Для *специализации метода* требуется определить вариант его кода, указав в заголовке конкретный тип данных. Например, если заголовок обобщенного метода print шаблона List имеет вид

```
template <class Data> void List <Data>::print();
```

специализированный метод для вывода списка символов будет выглядеть следующим образом:

```
void List <char>::print(){
 // Тело специализированного варианта метода print
```

Специализация всего класса

При *специализации целого класса* после описания обобщенного варианта класса помещается полное описание специализированного класса, при этом требуется заново определить все его методы.

```
// общий шаблон
template <class T> class Sample {
    bool Less(T) const; /*...*/ };
// специализация для char*
template <> class Sample<char*> {
    bool Less(char*) const; /*...*/ };
```

Если шаблонный класс имеет несколько параметров, то возможна частичная специализация:

```
template <class T1, class T2> class Pair { /*...*/ };
// специализация, где для T2 установлен тип int
template <class T1> class Pair <T1, int> { /*...*/ };
```

Использование классов функциональных объектов для настройки шаблонных классов

```
template <class T> struct LessThan {
    bool operator() (const T& x, const T& y) {
        return x < y;
    }
};
template <class T> struct GreaterThan {
    bool operator() (const T& x, const T& y) {
        return x > y;
    }
};
```

```
template <class T, class Compare>
class PairSelect {
public:
    PairSelect(const T& x, const T& y) : a(x), b(y) {}
    void OutSelect() const {
        cout << (Compare()(a, b) ? a : b) << endl;
    }
private:
    T a, b;
};
```

```
int main() {
    PairSelect<int, LessThan<int> > ps1(13, 9);
    ps1.OutSelect();          // вывод: 9
    PairSelect<double, GreaterThan<double> > ps2(13.8, 9.2);
    ps2.OutSelect();          // вывод: 13.8
    return 0;
}
```

Достоинства и недостатки шаблонов

- Шаблоны представляют собой мощное и эффективное средство обращения с различными типами данных, которое можно назвать параметрическим полиморфизмом, а также обеспечивают безопасное использование типов, в отличие от макросов препроцессора.
- Программа, использующая шаблоны, содержит полный код для каждого порожденного типа, что может увеличить размер исполняемого файла.
- С некоторыми типами данных шаблоны могут работать не так эффективно, как с другими. В этом случае имеет смысл использовать специализацию шаблона.