Лекция 8. Динамический полиморфизм. Часть 1

Аркадная игра «MiniMachines»

- **Описание**: по игровому полю проложена трасса. По ней едут машинки. Могут сталкиваться между собой и препятствиями (бочки, деревья). Могут собирать призы. Задача – приехать первым.

- Игровые объекты:
  - машинки,
  - препятствия,
  - призы (монетки).

Реализация объектов. Композиция

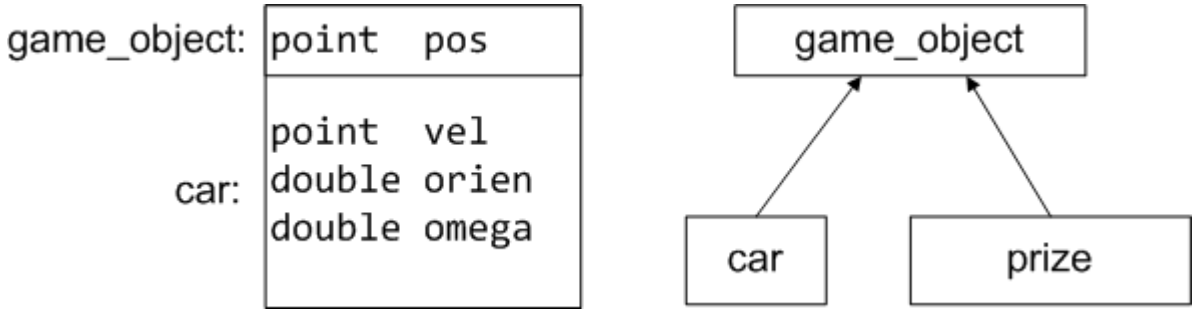
```
1.  struct game_object
2.  {
3.      point pos;
4.  };
5.
6.  struct car
7.  {
8.      game_object obj;
9.      point       vel;
10.     double      orien;
11.     double      omega;
12. };
13.
14. struct prize
15. {
16.     game_object obj;
17.     int         value;
18. };
```

- Машина – есть игровой объект. Но компилятор-то этого не знает. Например, сделать список игровых объектов, в который входят одновременно машины, призы и препятствия затруднительно.

Наследование

```
1.  struct car
2.      : game_object
3.  {
4.      point       vel;
5.      double      orien;
6.      double      omega;
7.  };
8.
9.  struct prize
10.     : game_object
11. {
12.     int value;
13. };
14.
15. void foo()
16. {
17.     game_object* obj = new car; // ok, implicit upcast
18.
19.     vector<game_object*> objects;
20.     objects.push_back(obj);
21.
22.     car* c = obj; // error, implicit downcast is forbidden
23.
24.     car* c = static_cast<car*>(obj); // ok, but be careful
25. }
```

Расположение полей




```
1.  game_object obj;
2.  obj.pos = point(10, 20); //ok
3.
4.  car c;
5.  c.pos = point(20, 10); //ok
```

- Небольшая путаница в терминологии: car – подкласс (подтип) game_object, но с точки зрения множества – надмножество.

Особенности наследование

- Базовый класс должен быть объявлен до наследование (нужен как минимум его размер).
- Из наследника нет доступа к private полям базового класса, но есть к public и protected.
- Наследоваться можно с модификаторами доступа (struct по умолчанию наследует public, class - private). В результирующем объекте используется минимальный модификатор доступа исходного поля и наследования.
- В наследнике можно перекрыть (и в результате скрыть) функцию базового класса.

Наследование. Пример 1

```
1.  struct Base
2.  {
3.      void foo(){}
4.
5.  private:
6.      void bar(){}
7.  };
8.
9.  struct Derived
10.     : public Base
11. {
12.     void foo()
13.     {
14.         Base::foo(); //ok
15.         bar(); // error, private
16.
17.         //...
18.     }
19. };
20.
21. void foo()
22. {
23.     Derived d;
24.     d.foo(); // Derived::foo
25.     d.Base::foo();
26. }
```

Наследование. Пример 2

```
 1. struct Base
 2. {
 3.     void foo   (){}
 4.     void foobar(){}
 5.
 6. private:
 7.     void bar(){}
 8. };
 9.
10. struct Derived
11.     : private Base
12. {
13.     using Base::foo;
14. };
15.
16. void foo()
17. {
18.     Derived d;
19.     d.foo   (); // ok
20.     d.foobar(); // error
21. }
```

Конструкторы

- Конструкторы не наследуются и не бывают виртуальными.
- Вызов конструктора базового класса обязателен, если только у базового класса нет дефолтного конструктора.
- Нельзя непосредственно определять поля базового класса в списке инициализации.
- Сперва вызываются конструкторы базовых классов (в порядке объявления наследования), затем полей. Удаление – в обратном порядке.

Конструкторы. Пример

```
1.  struct game_object
2.  {
3.      game_object(point pos) : pos(pos){}
4.      point pos;
5.  };
6.
7.  struct car
8.      : game_object
9.  {
10.     car(point pos, point vel, /*...*/)
11.         : game_object(pos)
12.         , vel        (vel)
13.         , /*...*/
14.     {}
15.
16.     car(point p)
17.         : pos(p) // error
18.         , /*...*/
19.     {}
20.
21.     point vel;
22.     /*...*/
23. };
```

Полиморфное поведение

- Как узнать на какой реальный объект ссылается указатель Base*? Например, чтобы нарисовать игровые объекты, имея лишь список указателей game_object*.

- Есть 4 варианта действия:
  - Запрет определения. Все объекты однотипные.
  - Хранить поле типа в базовом классе.
  - Делать динамическое преобразование типов (dynamic_cast).
  - Использовать виртуальные функции.

Поле типа

```
1.  enum object_type { ot_car, ot_prize, ot_obstacle };
2.  struct game_object { object_type type; /*...*/};
3.
4.  struct car  : game_object { point vel; /*...*/ };
5.  struct prize: game_object { int value; /*...*/ };
6.
7.  void render_car  (car   const&);
8.  void render_prize(prize const&);
9.
10. void render(game_object const& obj)
11. {
12.     if (obj.type == ot_car)
13.         render_car(static_cast<car const&>(obj));
14.     else if (obj.type == ot_prize)
15.         render_prize(static_cast<prize const&>(obj));
16.     else // ...
17. }
```

- При добавлении нового типа, найти все вхождения крайне сложно (не говоря про перекомпиляцию).

Виртуальные функции

```
1.  struct game_object
2.  {
3.      virtual void render(engine&, scope const&){}
4.  };
5.
6.  struct car  : game_object
7.  { void render(engine&, scope const&){/*...*/} };
8.
9.  struct prize: game_object
10. { void render(engine&, scope const&){/*...*/} };
11.
12. void render_objects(std::vector<game_object*> const& objs/*, ...*/)
13. {
14.     for (auto it = objs.begin(), end = objs.end(); it != end; ++it)
15.         (*it)->render(/*...*/);
16. }
```

Определение виртуальных функций

- Определение в базовом классе начинаются со слова virtual.
- В наследниках должен полностью совпадать прототип (есть послабления для возвращаемого типа).
- Виртуальная функция должна быть определена в классе, для которого объявлена.
- Если не нужна другая реализация у наследника, можно ее и не определять.
- Можно вызвать функцию базового типа.

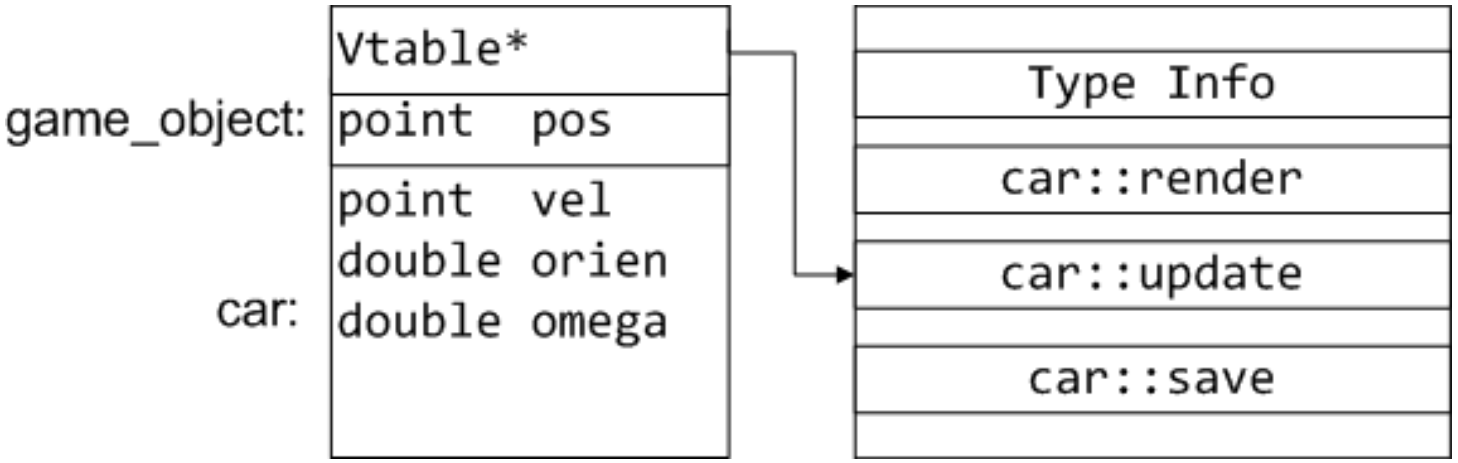
```
1.  struct Base
2.  { virtual void foo(){} };
3.
4.  struct Derived : Base
5.  { void foo(){ Base::foo(); /*...*/ }};
```

Та ли функция переопределена?

```
1.  struct Base
2.  {
3.      virtual void do_smth() const {}
4.  };
5.
6.  struct Derived : Base
7.  {
8.      void do_smth() {}                   // oops, new function
9.      void do_smth() const override {}// compilation check
10. };
```

- Ключевое слово `override` (C++11) заставит компилятор проверить, действительно ли была такая виртуальная функция в базовом классе

Таблица виртуальных функций



- Требуется память под дополнительный указатель на таблицу (сравнимо с полем типа).
- Обращение – всего один косвенный вызов (как и с библиотечной функцией из dll). Нет дополнительных оптимизаций компилятора.

Срезка

```
1.  struct rigid_body
2.  {
3.      virtual void apply_impulse(/*...*/) {}
4.  };
5.
6.  struct car     : game_object, rigid_body {/*...*/};
7.  struct obstacle: game_object, rigid_body {/*...*/};
8.
9.  void hit(rigid_body body, /*...*/) //oops, mistake
10. {
11.     body->apply_impulse(/*...*/); // does nothing
12. }
```

- Выход:
  - закрыть копирование у `rigid_body` ;
  - сделать его абстрактным (если у него нет своей реализации)

Чисто-виртуальные функции

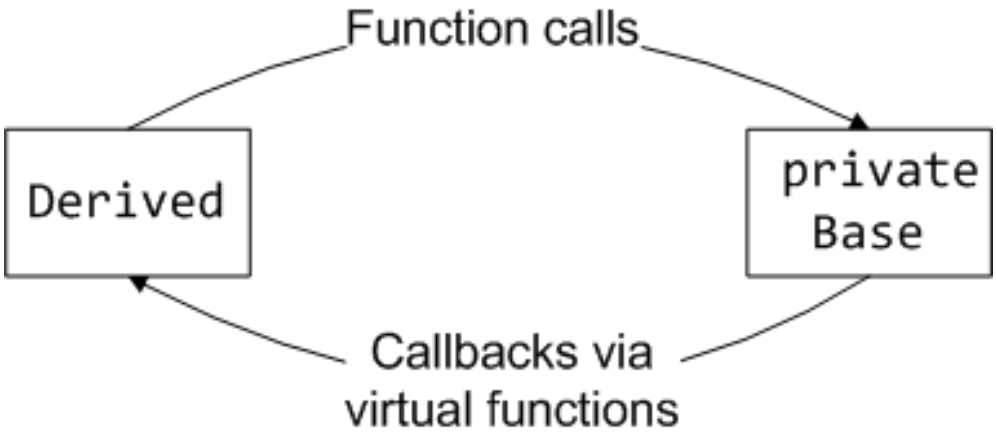
```
1.  struct game_object
2.  {
3.      virtual ~game_object(){}
4.
5.      virtual void update(double time)            = 0;
6.      virtual void render(engine&, scope const&) = 0;
7.      virtual void save  (std::ostream& os)       = 0;
8.  };
```

- Пока какой-нибудь из наследников не определит функцию, она остается чисто виртуальной (*pure*).
- Класс с чисто виртуальной функцией – абстрактный. Создать его экземпляр нельзя.
- Идеально подходят для описания интерфейсов

Открытое наследование

- Открытое наследование моделирует отношение «является». Закрытое - «реализовано с помощью».
- Принцип подстановки Лисков:
  - Все контракты базового класса должны быть выполнены, для чего перекрытие виртуальных функций не должно требовать большего или обещать меньшего, чем их базовая версия.
  - Функции, которые используют базовый тип, должны иметь возможность использовать подтипы базового типа, не зная об этом.

Закрытое наследование



- Почти всегда лучше использовать вместо него делегирование.
- Callbacks можно реализовать через анонимные функции или boost::function/bind
- Исключения: гарантия вызова функции до конструктора/после деструктора; boost::noncopyable

Деструктор

- При открытом наследовании обязательно(!) необходимо объявлять открытый **виртуальный** деструктор, в противном случае, удаляя объект через указатель на его базовый класс, Вы рискуете получить утечку памяти.

- Если Вы предполагаете закрытое наследование, у базового класса должен быть **защищенный** невиртуальный деструктор.

Вызов виртуальных функций

- Не вызывайте виртуальные функции из конструкторов или деструкторов.

- Пока не завершился конструктор полностью, таблица виртуальных функций может быть некорректна (в случае, если Ваш класс не последний в иерархии).

- С деструктором аналогично.

Вопросы?