Лекция 7. Виртуальные функции

ВС++ виртуальные функции (virtual functions) позволяют использовать полиморфизм (polymorhpism) классов. Так как виртуальные функции могут использоваться только внутри классов, то иногда их называют виртуальными методами (virtual methods). Прежде чем воспользоваться виртуальными методами, мы рассмотрим работу обычных методов класса.

Статическое или раннее связывание (static/early binding)

Давайте разберёмся, как происходит вызов обычных функций и методов классов. Вызов обычных функций и методов происходит через механизм, называемый статическим (статичным) связыванием (static binding) или ранним связыванием (early binding).

Раннее связывание использовалось во всех функциях и методах наших программ за исключением тех случаев, где мы использовали указатели на функции.

Когда мы запускаем сборку (building) программы, компилятор просматривает исходный код и превращает все операторы в команды процессора. Допустим, в коде встречается вызов какой-нибудьфункции:
someFunction(arg);

Если это обычная функция (не указатель на функцию), то при вызове используется механизм раннего связывания.

Во время компиляции для кода (определения) функции выделяется память, и назначаются адреса для каждого оператора. Первый адрес в определении (теле функции) является адресом функции. При вызове someFunction, процессор будет переходить на адрес функции и начнёт выполнять тело функции. Самое важное здесь то, что адрес функции назначается во время компиляции, и именно этот адрес используется при вызове функции. Это и есть раннее или статичное связывание. Т.е. имя функции крепко привязано к адресу функции.
Теперь взглянем на небольшой пример: class Base
{
public:
void Method ()
{
cout<< "Базовыйкласс\n";
}
};

class Derived : public Base {};

// внутри main Base b; Derived d; b.Method(); d.Method();

//--------Вывод: Базовый класс Базовый класс

На экран будет выведено две строки *Базовый класс*. На этапе компиляции память выделяется для двух копий Method - для базового класса и для производного. Оба адреса привязываются к именам методов: Base::Method, Derived::Method. Т.е. когда в коде мы вызываем Method, то вызывается метод, соответствующий типу объекта. Чтобы увидеть, что для каждого объекта вызывается свой метод, давайте переопределим метод Derived::Method:
void Method ()
{
cout << "Производный класс\n";
}

// внутри main Base b; Derived d; b.Method(); d.Method();

//--------Вывод: Базовый класс Производный класс

Здесь хорошо видно, что вызываются два разных метода. Теперь следующий пример. Определения классов оставим без изменений. Поработаемсуказателями:

Base* b = new Derived; Derived* d = new Derived; b->Method();d->Method();

//--------Вывод: Базовый класс Производный класс

Самое важное здесь то, что компилятор спокойно "проглатывает" тот факт, что указатель на Base указывает на производный класс. Дело в том, что базовый и производный классы являются совместимыми по типу.

Во время выполнения программы процессор видит, что b - это указатель на Base. Процессор не обращает внимание, что на самом деле этот указатель указывает на объект Derived. При вызове метода объекта bпроцессор переходит к адресу Base::Method.

Полиморфизм (polymorphism) и полиморфные типы (polymorphic types)

Рассмотрим гипотетическую ситуацию: в игре есть несколько типов монстров. Все монстры могут атаковать (attack) и перемещаться (move). При этом каждый вид монстров делает это по своему.

Неплохо было бы иметь возможность хранить объекты всех этих классов вместе и использовать одинаковый синтаксис для вызова методов этих классов. Это и есть полиморфизм (polymorphism) - много (от греческого поли) форм (от греческого морф). Т.е. объекты этих классов должны храниться в одном массиве.

Понятно, что для этого не годится массив объектов, так как в таком массиве для элементов выделяется фиксированное количество памяти. Соответственно, для поддержки полиморфизма нужно использовать массив указателей. Какой тип указателей выбрать? Как мы выяснили раньше, в указателе на базовый тип можно хранить объект любого производного типа - базовый и производный классы являются совместимыми по типу. Почему используется указатели именно на базовый тип? Потому что это более общий класс и от него наследуют все производные классы.

Классы, используемые для получения эффекта полиморфизма, называют

полиморфными типами (polymorphic types).

Позднее/динамическое связывание (late/dynamic binding)

Поздним связыванием в C++ обладают указатели на функции (function pointers). Мы их уже разбирали, поэтому сложностей возникнуть не должно. Сразу пример:

int someFunction (int arg); int (*functionPointer)(int arg);
functionPointer = someFunction.

someFunctionобладаетраннимсвязыванием. Т.е. на этапе компиляции для этой функции выделяется участок памяти, а первый адрес этого участка становится адресом функции. Адрес функции жёстко привязан к имени функции - их нельзя отделить.

functionPointer обладает динамическим (dynamic) или поздним связыванием (late binding). На какую функцию указывает этот указатель, становится известно только во время выполнения программы.При этом functionPointer может указывать на любую функцию, т.е. значение указателя functionPointer может меняться во время выполнения программы. Это и есть позднее связывание.

Виртуальные функции/методы (virtual functions/methods)

Чтобы объявить функцию как виртуальную, необходимо добавить ключевое слово virutal перед именем возвращаемого типа:

class Base

```
{
public:
virtual void Method ()
{
cout << "Базовый класс\n";
}
};
```

Вносить изменения в производные классы не нужно. Хотя можно и там добавить ключевое слово virtual (это не обязательно). Теперьпосмотримнанашкод:

```
Base* b = new Derived; Derived* d = new Derived; b->Method();d->Method();


//--------Вывод: Производный класс Производный класс
```

Наконец-томы можем создать массив указателей на базовый класс и размещать там объекты любого производного класса:

```
BaseMonster* monsters[3]; monsters[0] = new MonsterA;


monsters[1] = new MonsterB; monsters[2] = new MonsterC;


for (int i=0;i<3;++i) monsters[i]->attack();
```

Несколько замечаний по виртуальным функциям:

1. Виртуальные функции используются только в классах. Поэтому часто используется название - виртуальные методы.
2. В массивах указателей на базовый класс можно хранить объекты только полиморфных типов (базовый и все производные).
3. В массив нужно объединять только те объекты, которые обладают методами с одинаковыми названиями, но разной реализацией.

Таблица виртуальных функций (virtual function table)

Для виртуальных методов память выделяется точно так же, как и для обычных: на этапе компиляции под эти методы выделяются участки памяти, первые адреса которых являются адресами методов. Но так как методы виртуальные, то фактические адреса метода не привязывается к именам: Base::vf и Derived::vf. Адрес метода, который назначается на этапе компиляции при выделении памяти, будем называть настоящим (или фактическим) адресом.

Когда в базовом классе объявляется хотя бы одна виртуальная функция, то для всех полиморфных классов создаётся таблица виртуальных функций (virtual function table).

Таблица виртуальных функций - это одномерный массив указателей на функции. Количество элементов в массиве равно количеству виртуальных функций в классе.

Для каждого полиморфного класса (базового и всех производных) создаётся своя таблица виртуальных методов. Количество элементов во всех этих таблицах одинаковое.

Именно в таблице виртуальных функций записываются настоящие адреса методов, т.е. элемент таблицы является указателем на функцию. Для всех полиморфных классов таблицы виртуальных функций будут содержать разные значения. Для каждого класса здесь будут записаны адреса методов данного класса.

Абстрактные классы (abstract classes) и чистые виртуальные функции (pure virtual functions)

Очень часто в программах не требуется создавать объекты базовых классов. Т.е. базовые классы нужны только для того, чтобы построить иерархию классов и определить общие свойства для производных классов. Такие классы можно сделать абстрактными (abstract class). При попытке создания объекта абстрактного класса, компилятор выдаст ошибку. Чтобы сделать класс абстрактным, нужно объявить одну из виртуальных функций чистой. Чистая виртуальная функция (pure virtual function) как бы намекает, что она будет реализована в производных классах.

Чтобы сделать виртуальную функцию чистой (pure), нужно добавить после заголовка функции символы =0 (знак равенства и ноль):

```
class Base
{
public:

virtual void method () =0; virtual ~Base() =0;
};
```

В данном случае уже не нужно писать определение такой функции. Помимо этого теперь нельзя создавать объекты класса Base, так как он стал абстрактным.

Символы =0 необязательно добавлять ко всем виртуальным функциям, достаточно добавить к одной.