# Лекция 10. Полиморфизм и виртуальные функции

Третьим важнейшим аспектом объектно-ориентированных языков программирования (после инкапсуляции и наследования) является полиморфизм, реализованный в C++ при помощи виртуальных функций.   
Полиморфизм выводит принцип отделения интерфейса от реализации на качественно новый уровень: он, если можно так выразиться, разделяет «что» и «как». Виртуальные функции базового класса определяют, что должны делать объекты производных классов, а их переопределённые в производных классах версии определяют, как это действие должно выполняться объектами данного класса.

Полиморфизм помогает создавать расширяемые программы, возможности которых могут наращиваться по мере необходимости:   
- механизм инкапсуляции позволяет создавать новые типы данных за счет объединения характеристик с наследуемым от базового класса интерфейсом;   
- механизм управления доступом позволяет отделить интерфейс от реализации, подробности которой скрываются в закрытой части класса.   
 Благодаря наследованию объект можно интерпретировать как объект своего или базового типа. Эта возможность чрезвычайно важна, поскольку она позволяет интерпретировать разные типы, производные от общего базового типа, как относящиеся к одному типу. Таким образом, вы можете написать фрагмент программы, который будет работать с любым из множества производных типов.   
 Виртуальные функции позволяют одному типу выразить свое отличие от другого сходного типа — при условии, что они оба наследуют от одного базового типа. Отличие проявляется в поведении функций, вызываемых через базовый класс.   
Иными словами, базовый класс определяет, *что* должны делать все объекты производных классов, то есть их интерфейс, а в производных классах определяется реализация функций интерфейса, то есть *как* это будет выполняться в конкретном классе.  
Например, в макете СУБД один и тот же интерфейс базового класса DBTable реализуется в двух разных по своим характеристикам классах: DBTableTxt и DBTableBin, а в классе DBTableSet оба эти класса представлены указателем на базовый класс DBTable\*.

# 1. Повышающее приведение типа

Рассмотрим примеры наследования, в которых объект может интерпретироваться как относящийся к своему или к базовому типу, и покажем, как с таким объектом можно работать через адрес базового типа.

Получение адреса объекта (в виде указателя или ссылки) и его дальнейшая интерпретация как адреса объекта базового типа называется повышающим приведением типа (этот термин появился из-за того, что базовые классы традиционно изображались в верхней части диаграмм иерархии наследования).

//: L10 Instrument.cpp  
// Наследование и повышающее приведение типа   
#include <iostream>

using namespace std;

enum note { Do, Re, Ci };// И т. д.

class Instrument {

public:

void play(note) const {

cout << "Instrument::play" << endl;

}

};

// Духовые музыкальные инструменты (Wind) являются музыкальными

// инструментами (Instrument), поскольку обладают тем же

//интерфейсом:

class Wind : public Instrument {  
 public: // Переопределение интерфейсной функции:

void play(note) const {

cout << "Wind::play" << endl;

}

};

class Stringed : public Instrument {//струнные

public: // Переопределение интерфейсной функции:

void play(note) const {

cout << "Stringed::play" << endl;

}

};

void tune(Instrument\* i) {//мелодия

// ...

i->play(Ci);

}

int main() {

Instrument\* A[2];

Wind flute;

Stringed violin;

cout<<"Размер объекта Instrument = "<<sizeof(Instrument)<<endl;

A[0]=&flute;//(флейта)Повышающее приведение типа

A[1]=&violin;//(скрипка)Повышающее приведение типа

tune(A[0]);

tune(A[1]);

system("pause");

return 0;

}

Функция tune() получает (по указателю) объект Instrument, но она также беспрекословно примет любой объект класса, производного от Instrument. Так, внутри функции main() в функцию tune() передаются объекты Wind и Stringed, не требуя приведения типа. Это вполне логично, весь интерфейс Instrument заведомо присутствует в Wind и Stringed, потому что они открыто наследуются от Instrument.

Повышающее приведение типа Wind к Instrument «сужает» интерфейс производного класса, но никогда не нарушает границ полного интерфейса Instrument. Эти рассуждения остаются истинными и при работе с указателями. Единственное отличие состоит в том, что пользователь должен явно указать адрес объекта, передаваемого функции.

*Проблема.*   
При запуске программы Instrument.cpp неожиданно выявляется проблема: программа выводит строку Instrument::play. Конечно, это не тот результат, на который мы рассчитывали, поскольку известно, что объект в действительности относится к классу Wind, а не Instrument. Предполагалось, что программа выведет Wind::play.   
Как сделать так, чтобы для любого объекта класса, производного от Instrument, всегда вызвалась бы его версия функции play()?   
Чтобы лучше понять происходящее, необходимо разобраться в концепции связывания.

# 2. Связывание вызовов функций

Сопоставление вызова функции (определение адреса, связанного с именем функции) с телом функции называется связыванием. Если связывание выполняется до запуска программы (то есть компилятором и компоновщиком), оно называется ранним связыванием.   
Проблемы в приведенной выше программе возникли именно из-за раннего связывания, поскольку компилятор не может определить правильный адрес вызываемой функции, располагая только адресом Instrument. На помощь приходит “позднее связывание”. Этот термин означает, что связывание производится во время выполнения программы в зависимости от фактического типа объекта. Позднее связывание также называют динамическим связыванием.

Если язык программирования поддерживает позднее связывание, в нем должен присутствовать некий механизм, который бы определял тип объекта во время выполнения и вызывал нужную функцию класса. В компилируемых языках компилятор не знает фактического типа объекта, но вставляет в программу фрагмент кода для определения и вызова правильной функции. Реализация механизмов позднего связывания зависит от языка, но можно предположить, что в любом случае в объекте должна сохраняться информация о его типе.   
Рассмотрим, как этот механизм работает в C++.

# 3. Виртуальные функции

Чтобы в C++ для функции выполнялось позднее связывание, она должна быть объявлена виртуальной, то есть ее объявление в базовом классе должно содержать ключевое слово virtual. Позднее связывание работает только с виртуальными функциями и только при использовании адреса базового класса, содержащего виртуальные функции (хотя функции также могут определяться в одном из более ранних базовых классов).

Итак, для создания виртуальной функции достаточно включить в ее объявление ключевое слово virtual. Ключевое слово virtual обязательно только для объявления функции, но не для ее определения. Если функция объявляется виртуальной в базовом классе, она также становится виртуальной во всех производных классах. Повторное определение виртуальной функции в производном классе обычно называется переопределением. Обратите внимание: ключевое слово virtual обязательно только при объявлении функции в базовом классе. Все функции производных классов, по сигнатуре соответствующие объявлению в базовом классе, будут вызываться с помощью механизма виртуальных функций. Ключевое слово virtual может использоваться в объявлениях производных классов, вреда от него не будет, однако оно является избыточным и может вызвать недоразумения. Чтобы добиться желаемого поведения от программы Instrument.cpp, просто включим ключевое слово virtual в базовый класс перед объявлением функции play():

//L10:Instruments2.срр

// Позднее связывание с ключевым словом virtual

#include <iostream>

using namespace std;

enum note { Do, Re, Ci };// И т. д.

class Instrument {

//int i;

public:

virtual void play(note) const {

cout << "Instrument::play" << endl;

}

};

class Wind : public Instrument {

public: // Переопределение интерфейсной функции:

void play(note) const {

cout << "Wind::play" << endl;

}

};

class Stringed : public Instrument {

public: // Переопределение интерфейсной функции:

void play(note) const {

cout << "Stringed::play" << endl;

}

};

void tune(Instrument\* i) {

// ...

i->play(Ci);

}

int main() {

Instrument\* A[2];

Wind flute;

Stringed violin;

cout<<"Размер объекта Wind = "<<sizeof(Wind)<<endl;

cout<<"Размер объекта Instrument = "<<sizeof(Instrument)<<endl;

A[0]=&flute;

A[1]=&violin;

tune(A[0]); // Повышающее приведение типа

tune(A[1]); // Повышающее приведение типа

system("pause");

}

Если не считать появления ключевого слова virtual, пример идентичен

примеру Instrument.cpp, но поведение программы принципиально изменилось:

в новой версии выводятся строки Wind::play и Stringed:: violin.

Если сравнить размеры объектов Wind и Instrument в проектах Instrument и Instrument2, то мы увидим, что после объявления в классе Instrument функции *play()* виртуальной, размеры объектов этих классов увеличились на 4 байта за счет добавления в них скрытого параметра – адреса таблицы адресов виртуальных функций.

# 4. Расширяемость программ

После определения в базовом классе виртуальной функции play() в программу можно включить сколько угодно новых типов инструментов, не изменяя функции tune(). В хорошо спроектированной объектно-ориентированной программе многие функции следуют примеру tune() и взаимодействуют только с интерфейсом базового класса. Такие программы хорошо расширяются, потому что программист может добавлять в них новые возможности посредством наследования новых типов данных от общего базового класса. При этом функции, работающие через интерфейс базового класса, изменять вообще не придется.

# 5. Позднее связывание в C++

Позднее, или динамическое связывание означает, что связывание производится во время выполнения программы в зависимости от фактического типа объекта, вызывающего функцию.   
Позднее связывание используется для создания обобщенных программ, не зависящих от типа объектов.   
Реализация механизмов позднего связывания зависит от языка, но *во всех реализациях в объекте нужно сохранить информацию о его типе.*

В С++ позднее (динамическое) связывание реализуется с помощью указателей.

Чтобы в C++ для функции выполнялось позднее связывание, она должна быть объявлена виртуальной, то есть ее объявление в базовом классе должно содержать ключевое слово virtual. Повторное определение виртуальной функции в производном классе называется *переопределением*.  
Ключевое слово virtual обязательно только для объявления функции, но не для ее определения.  
Встроенное в язык С++ позднее связывание работает *только с виртуальными функциями и только при использовании адреса базового класса, содержащего виртуальные функции*.  
Ключевое слово virtual сообщает компилятору, что он не должен выполнять раннее связывание. Все функции производных классов, по сигнатуре соответствующие объявлению в базовом классе, будут вызываться с помощью механизма виртуальных функций.  
Для позднего связывания компилятор создает таблицу виртуальных функций для каждого *класса*, содержащего виртуальные функции, и помещает в неё адреса виртуальных функций данного класса. В каждом классе, содержащем виртуальные функции, скрыто сохраняется указатель VPTR, который ссылается на таблицу виртуальных функций. При вызове виртуальной функции через указатель на базовый класс (то есть при полиморфном вызове) компилятор находит адрес функции в таблице виртуальных функций того производного класса, адресом объекта которого инициализирован указатель на базовый класс.   
Ни один из созданных нами классов не содержал явно заданной информации о типе. Тем не менее предыдущие примеры и простая логика подсказывают, что в объектах должна храниться какая-то информация о типе, иначе тип объекта не удалось бы определить во время выполнения программы. В этом можно убедиться, если сравнить размер объекта производного класса до и после объявления в базовом классе одной из его функций виртуальной: размер увеличится на 4 байта (на размер указателя VPTR ).

В С++можно писать обобщенные функции с использованием указателя void\*. В этом случае программист должен сам создавать код, который где-то сохраняет информацию о типе объекта и затем использует её для «ручного» позднего связывания с обрабатывающими функциями, зависящими от типа объекта. Реализация ручного выполняется программистом.

# 6. Использование полиморфизма в макете СУБД

В системе классов для макета СУБД (смотри диаграмму классов в конце текста лекции) мы будем использовать виртуальные функции в базовом классе DBTable для объявления интерфейса производных классов.  
 В производных классах DBTableTxt и DBTableBin нам нужно будет только переопределить эти функции. В классе DBTableSet эти функции будут вызываться одинаково, через указатель DBTable\*, предварительно инициализированный адресом одного из объектов производного класса, то есть класс DBTableSet является полиморфным. Он одинаково обрабатывает табличные данные, хранящиеся как в текстовых, так и в бинарных файлах. Код, сгенерированный компилятором, автоматически определяет, функции какого класса нужно вызывать.

Для реализации возможности хранения в полях таблиц типа DBTableTxt и DBTableBin объектов различных типов, как встроенных, так и пользовательских, нам придется использовать указатель void\*. Это связано с тем, что в С++ встроенные типы, а также типы из STL, например, тип string, не имеют общего базового класса и поэтому для работы с ними нельзя создать виртуальные функции.

Если мы вынуждены использовать для позднего связывания указатель на void\*, то нам нужно будет самим где-то сохранять информацию о типе данных, хранящихся в полях таблицы, и вставлять в программу код для «ручного связывания» void\* с фактическим типом . В макете СУБД эта информация сохраняется в файле в заголовках столбцов таблицы (структура ColumnDesc):  
struct ColumnDesc {

char colName[LENGTH];//имя столбца

DBType colType;//тип данных в столбце таблицы

int length; //максимальное число символов, допустимое

//для представления данных в столбце

};  
а для определения типа данных, адреса которых присваиваются void\* obj, использовать примерно такой код:   
switch(colType) {   
 case Int32: . . . \*(int\*) obj ; break;  
 case Double: . . . \*(double\*) obj; break;  
 . . .  
}

Используя тестовые версии БД LibraryTxt и CompanyTxt создайте с помощью методов DBTableTxt:: ReadDBtableBin() и DBTableTxt:: WriteDBtableBin бинарные версии таблиц БД и поместите текстовые и бинарные версии в папки LibraryTxt, LibraryBin, CompanyTxt и CompanyBin соответственно.

# 7. Использование текстовых файлов и бинарных файлов прямого доступа для таблиц БД.

Файл – это *структурированная* последовательность байтов, то есть последовательность, организованная по определённым правилам. Знание правил организации данных необходимо для их правильного считывания.   
В методах чтения файлов, открытых в текстовом режиме, байты интерпретируются как символы и некоторые из них используются для выделения структур данных, например, символы-разделители в текстовых файлах в формате CSV, символ конца строки или символ конца файла.  
 Достоинство текстовых файлов – удобство создания, чтения и внесения изменений. Для этого имеются специальные программы - текстовые редакторы. Кроме того, использование контейнеров типа *map* (ассоциативных массивов) позволяет обращаться к данным полей таблицы по имени столбца, что значительно облегчает чтение программ.  
Недостаток текстовых таблиц БД – необходимость переформатирования данных при чтении из файла и при записи в файл.

В методах чтения файлов, открытых в бинарном режиме, таких как read() и write(), байты интерпретируются как копия байтов ОП, а для выделения структур данных используется размер структуры. Если нам нужно читать бинарные файлы с различной структурой, то размер структуры в байтах записывается в файл перед её телом. При чтении сначала в переменную *int size* считывается размер структуры (целое число – 4 байта), а затем считываются *size* байт данных структуры. Для просмотра бинарных файлов используются специальные программы - редакторы файлов.   
Достоинство бинарных таблиц БД – отсутствие необходимости переформатирования данных при чтении из файла и при записи в файл. Это достоинство будет проявляться с наибольшей эффективностью, если в ОП данные структур хранятся в последовательных ячейках памяти, друг за другом. Для реализации этого требования структуры данных следует хранить в массивах.  
Недостаток использования массивов – необходимость использования целочисленного индекса (порядкового номера структуры, а не её имени) для обращения к конкретной структуре. Это сильно ухудшает читабельность программы. Однако, этот существенный недостаток можно обойти, если для хранения данных использовать *бинарные файлы прямого доступа*. В этом случае для конкретной таблицы размер полей будет одинаковым, что позволяет достаточно легко реализовать интерфейс ассоциативного массива не на базе бинарного дерева (память динамически выделяется для каждой структуры) а на базе массива. В этом случае память выделяется сразу на всю строку таблицы, а затем из этой памяти, зная размер данных в столбцах таблицы, выбираются байты отдельных структур.  
При чтении таблицы необходимо сначала прочитать заголовок, так как содержащиеся в нём данные используются при чтении строк таблицы.

