Виртуальные функции

При наследовании часто бывает необходимо, чтобы поведение некоторых методов базового класса и классов-наследников различалось. Можно переопределить соответствующие методы в производном классе. Однако тут возникает одна проблема, которую лучше рассмотреть на простом примере:

```
#include <iostream>
using namespace std;
class Base     // базовый класс
{ public:
int  f(const int &d) //метод базового класса
{ return 2*d; }
int CallFunction(const int &d)
{return f(d)+l; }// вызов метода базового класса
};
Class Derived: public Base // производный класс
{   public: // CallFunction наследуется
int f(const int &d) // метод f переопределяется
{ return d*d; }
};
int main()
{     Base a; // объект базового класса
cout<<a.CallFunction(5)<<endl; //получаем 11
Derived b; // объект производного класса
cout << b.CallFunction(5)<< endl; // какой
                метод f вызывается?
return 0;
```

В базовом классе определены два метода – f() и) CallFunction(), причем во втором методе вызывается первый. В классе-наследнике метод f() переопределен, а метод CallFunction() унаследован. Очевидно, метод f() переопределяется для того, чтобы объекты базового класса и класса-наследника вели себя по-разному.

Объявляя объект b типа Derived, программист, естественно, ожидает получить результат 5*5 + 1 = 26 – для этого и переопределялся метод f().

Однако на экран, как и для объекта a типа Base, выводится число 11, которое, очевидно, вычисляется как 2*5+1 = 11.

Несмотря на переопределение метода f() в классе-наследнике, в унаследованной функции CallFunction() вызывается «родная» функция f(), определенная в базовом классе!

При трансляции класса Base компилятор ничего не знает о классах-наследниках, поэтому он не может предполагать, что метод f() будет переопределен в классе Derived.

Его естественное поведение – «прочно» связать вызов f() с телом метода класса Base.

Виртуальные функции

Аналогичная проблема возникает и в несколько другом контексте: при подстановке ссылки или указателя на объект производного класса вместо ссылки или указателя на объект базового.

```
class Clock // базовый класс - часы
{ public:
    void print() const
      { cout <<"Clock!"<<endl; }    };
class Alarm: public Clock
// производный класс  - будильник
{ public:
     void print() const // переопределенный метод
       { cout << "Alarm!" << endl; }
 };
     void settime(Clock d)
       {d.print(); }
//.. .
int main()
 {  Clock W;      // объект базового класса
    settime(W);  // выводится "Clock"
    Alarm U;      // объект производного класса
     settime(U); //  ссылка на производный вместо
                    базового
   Clock *cl=&W;  //адрес объекта базового класса
   cl->print();   // вызов базового метода
   cl = &U; // адрес объекта производного типа
                    вместо базового
cl->print();  //  какой метод вызывается, базовый
                    или производный
}
```

Опять в классе-наследнике переопределен метод для того, чтобы обеспечить различное поведение объектов базового и производного классов. Однако и при передаче параметра по ссылке базового класса в функцию settime(), и при явном вызове метода print() через указатель базового класса наблюдается одна и та же картина: всегда вызывается метод базового класса, хотя намерения программиста состоят в том, чтобы вызвать метод производного.

Для того чтобы разобраться в ситуации, необходимо уяснить, что такое связывание.

Связывание – это сопоставление вызова функции с телом. В приведенных ранее примерах связывание выполняется на этапе трансляции (до запуска) программы. Такое связывание обычно называют ранним, или статическим.

При трансляции функции settime() компилятору ничего не известно о типе реально передаваемого объекта во время выполнения программы. Поэтому вызов метода print() связывается с телом метода базового класса Clock, как и определено в заголовке функции settime().

Точно так же указатель на базовый класс «прочно» связывается с методом базового класса во время трансляции.

Виртуальные функции

Чтобы добиться разного поведения в зависимости от типа, необходимо объявить функцию-метод виртуальной; в C++ это делается с помощью ключевого слова virtual.

Виртуальная функция (virtual function) – это функция-член, объявленная в базовом классе и переопределенная в производном.

Ключевое слово virtual указывается до объявления функции в базовом классе.

Производный класс переопределяет эту функцию, приспосабливая ее для своих нужд. По существу, виртуальная функция реализует принцип "один интерфейс, несколько методов", лежащий в основе полиморфизма.

Виртуальная функция в базовом классе определяет вид интерфейса, т.е. способ вызова этой функции. Каждое переопределение виртуальной функции в производном классе реализует операции, присущие лишь данному классу. Иначе говоря, переопределение виртуальной функции создает конкретный метод (specific method).

При обычном вызове виртуальные функции ничем не отличаются от остальных функций-членов. Особые свойства виртуальных функций проявляются при их вызове с помощью указателей. Указатели на объекты базового класса можно использовать для ссылки на объекты производных классов. Если указатель на объект базового класса устанавливается на объект производного класса, содержащий виртуальную функцию, выбор требуемой функции основывается на типе объекта, на который ссылается указатель, причем этот выбор осуществляется в ходе выполнения программы. Таким образом, если указатель ссылается на объекты разных типов, то будут вызваны разные виртуальные функции. Это относится и к ссылкам на объекты базового класса.

Виртуальные функции

Таким образом, объявление метода f() в базовом и производном классах должно быть таким:

```
virtual  int  f(const int &d)       // в базовом классе
   { return 2*d; }
 virtual int f(const int &d)      // в производном классе
   { return d*d: }
```

После этого для объектов базового и производного классов мы получаем разные результаты: 11 и 26.

Аналогично в объявление метода print() тоже должно начинаться со слова virtual:

```
virtual      void  print() const          // в базовом классе
      { cout << "Clock!" << endl: }
virtual      void  print() const          // в производном классе
      { cout << "Alarm!" << endl; }
```

После этого вызов settime() с параметром базового класса обеспечит вывод на экран слова «Clock», а с параметром производного класса – слова «Alarm». И при вызове по указателю наблюдается та же картина.

Вообще-то ключевое слово virtual достаточно написать только один раз – в объявлении функции базового класса. Определение можно писать без слова virtual – все равно функция будет считаться виртуальной. Однако лучше всегда это делать явным образом, чтобы всегда по тексту было видно, что функция является виртуальной.

Для виртуальных функций обеспечивается не статическое, а **динамическое (позднее, отложенное) связывание**, которое реализуется во время выполнения программы.

В C++ реализованы два типа полиморфизма:

- статический полиморфизм, или полиморфизм времени компиляции (compile-time polymorphism), осуществляется за счет перегрузки и шаблонов функций;
- динамический полиморфизм, или полиморфизм времени выполнения (runtime

Виртуальные функции

С перегрузкой функций "разбирается" компилятор, правильно подбирая вариант функции в той или иной ситуации. Полиморфизм шаблонных функций тоже реализуется на этапе компиляции. Естественно, выбор осуществляется статически.

Выбор же виртуальной функции происходит динамически – при выполнении программы. Класс, включающий в себя виртуальные функции, называется полиморфным.

Правила описания и использования виртуальных функций-методов:

1) **Виртуальная функция может быть только методом класса,**
2) **Любую перегружаемую операцию-метод класса можно сделать виртуальной, например, операцию присваивания или операцию преобразования типа,**
3) **Виртуальная функция, как и сама виртуальность, наследуется,**
4) **Виртуальная функция может быть константной,**
5) **Если в базовом классе определена виртуальная функция, то метод производного класса с такими же именем и прототипом (включая тип возвращаемого значения и константность метода) автоматически является виртуальным (слово virtual указывать необязательно) и замещает функцию-метод базового класса,**
6) **Конструкторы не могут быть виртуальными,**
7) **Статические методы не могут быть виртуальными,**
8) **Деструкторы могут (чаще – должны) быть виртуальными – это гарантирует корректный возврат памяти через указатель базового класса.**

Виртуальные функции

Объясним виртуальные функции ещё раз по-другому:

Работа с объектами чаще всего производится через указатели. Указателю на базовый класс можно присвоить значение адреса объекта любого производного класса, например:

```
monstr *p:              // Описание указателя на базовый класс
p = new daemon;         // Указатель ссылается на объект производного класса
```

Вызов методов объекта происходит в соответствии с типом указателя, а не фактическим типом объекта, на который он ссылается,

поэтому при выполнении оператора, например, `p->draw(l, 1. 1. 1);`

будет вызван метод класса monstr, а не класса daemon, поскольку ссылки на методы разрешаются во время компоновки программы. Этот процесс называется ранним связыванием.

Чтобы вызвать метод класса daemon, можно использовать явное преобразование типа указателя: `(daemon * p)->draw(l, 1, 1. 1);`

Но это не всегда возможно, поскольку в разное время указатель может ссылаться на объекты разных классов иерархии, и во время компиляции программы конкретный класс может быть неизвестен. В качестве примера можно привести функцию, параметром которой является указатель на объект базового класса. На его место во время выполнения программы может быть передан указатель на любой производный класс. Другой пример – связный список указателей на различные объекты иерархии, с которым требуется работать единообразно.

В C++ реализован механизм позднего связывания, когда разрешение ссылок на метод происходит на этапе выполнения программы в зависимости от конкретного типа объекта, вызвавшего метод. Этот механизм реализован с помощью виртуальных методов.

Для определения виртуального метода используется спецификатор virtual, например:

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

Виртуальные функции

Правила описания и использования виртуальных методов:

- **Если в базовом классе метод определен как виртуальный, метод, определенный в производном классе с тем же именем и набором параметров, автоматически становится виртуальным, а с отличающимся набором параметров – обычным,**
- **Виртуальные методы наследуются, то есть переопределять их в производном классе требуется только при необходимости задать отличающиеся действия. Права доступа при переопределении изменить нельзя,**
- **Если виртуальный метод переопределен в производном классе, объекты этого класса могут получить доступ к методу базового класса 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 описание виртуального метода, он должен быть определен хотя бы как чисто виртуальный.**

Чисто виртуальный метод содержит признак - 0 вместо тела, например:

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

Чисто виртуальный метод должен переопределяться в производном классе (возможно, опять как чисто виртуальный).

Виртуальные функции

Если определить метод draw в классе monstr как виртуальный, решение о том, метод какого класса вызвать, будет приниматься в зависимости от типа объекта, на который ссылается указатель:

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

Если объект класса daemon будет вызывать метод draw не непосредственно, а косвенно (то есть из другого метода, определенного в классе monstr), будет вызван метод draw класса daemon.

Итак, виртуальным называется метод, ссылка на который разрешается на этапе выполнения программы, то есть ссылка разрешается по факту вызова.

Виртуальные функции

Механизм позднего связывания

Для каждого класса (не объекта!), содержащего хотя бы один виртуальный метод, компилятор создает таблицу виртуальных методов (vtbl), в которой для каждого виртуального метода записан его адрес в памяти. Адреса методов содержатся в таблице в порядке их описания в классах. Адрес любого виртуального метода имеет в vtbl одно и то же смещение для каждого класса в пределах иерархии.

Каждый объект содержит скрытое дополнительное поле ссылки на vtbl, называемое vptr. Оно заполняется конструктором при создании объекта (для этого компилятор добавляет в начало тела конструктора соответствующие инструкции).

На этапе компиляции ссылки на виртуальные методы заменяются на обращения к vtbl через vptr объекта, а на этапе выполнения в момент обращения к методу его адрес выбирается из таблицы. Таким образом, вызов виртуального метода, в отличие от обычных методов и функций, выполняется через дополнительный этап получения адреса метода из таблицы. Это несколько замедляет выполнение программы.

Рекомендуется делать виртуальными деструкторы для того, чтобы гарантировать правильное освобождение памяти из-под динамического объекта, поскольку в этом случае в любой момент времени будет выбран деструктор, соответствующий фактическому типу объекта. Деструктор передает операции delete размер объекта, имеющий тип size_t. Если удаляемый объект является производным и в нем не определен виртуальный деструктор, передаваемый размер объекта может оказаться неправильным.

Виртуальные функции

Механизм позднего связывания

Четкого правила, по которому метод следует делать виртуальным, не существует. Можно только дать рекомендацию объявлять виртуальными методы, для которых есть вероятность, что они будут переопределены в производных классах.

Методы, которые во всей иерархии останутся неизменными или те, которыми производные классы пользоваться не будут, делать виртуальными нет смысла.

С другой стороны, при проектировании иерархии не всегда можно предсказать, каким образом будут расширяться базовые классы (особенно при проектировании библиотек классов), а объявление метода виртуальным обеспечивает гибкость и возможность расширения.

Для пояснения последнего тезиса представим себе, что вызов метода draw осуществляется из метода перемещения объекта. Если текст метода перемещения не зависит от типа перемещаемого объекта (поскольку принцип перемещения всех объектов одинаков, а для рисования вызывается конкретный метод), переопределять этот метод в производных классах нет необходимости, и он может быть описан как невиртуальный. Если метод draw виртуальный, метод перемещения сможет без перекомпиляции работать с объектами любых производных классов – даже тех, о которых при его написании ничего известно не было.

Виртуальный механизм работает только при использовании указателей или ссылок на объекты. Объект, определенный через указатель или ссылку и содержащий виртуальные методы, называется полиморфным. В данном случае полиморфизм состоит в том, что с помощью одного и того же обращения к методу выполняются различные действия в зависимости от типа, на который ссылается указатель в каждый момент времени.

Виртуальные функции

Сравнение раннего и позднего связывания

Раннее связывание (early binding) означает события, происходящие на этапе компиляции. Раннее связывание означает, что на этапе компиляции известна вся информация, позволяющая выбрать вызываемую функцию. (Иначе говоря, объект и вызов функции связываются друг с другом на этапе компиляции.) Примерами раннего связывания являются обычные вызовы функций (включая стандартные библиотечные функции), вызовы перегруженных функции и операторов. Основное преимущество раннего связывания – эффективность. Поскольку вся информация о вызываемой функции на этапе компиляции уже известна, сам вызов функции происходит очень быстро.

Позднее связывание (late binding) является антиподом раннего. В языке C++ позднее связывание означает, что фактический выбор вызываемой функции осуществляется только в ходе выполнения программы. Основным средством позднего связывания являются виртуальные функции. Выбор вызываемой виртуальной функции зависит от типа указателя или ссылки, с помощью которых она вызывается. Поскольку в большинстве случаев на этапе компиляции эта информация отсутствует, связывание объекта и вызова функции откладывается до выполнения программы. Основное преимущество позднего связывания – гибкость. В отличие от раннего, позднее связывание позволяет создавать программу, реагирующую на события, происходящие в ходе ее выполнения, без использования большого количества кода, предусматривающего всевозможные варианты. Однако позднее связывание может замедлить работу программы.

Виртуальные функции

Абстрактные классы

Класс, содержащий хотя бы один чисто виртуальный метод, называется абстрактным (abstract class). Абстрактные классы предназначены для представления общих понятий, которые предполагается конкретизировать в производных классах. Абстрактный класс может использоваться только в качестве базового для других классов – объекты абстрактного класса создавать нельзя, поскольку прямой или косвенный вызов чисто виртуального метода приводит к ошибке при выполнении (т.к. чисто виртуальные функции не имеют определения).

Несмотря на то что объекты абстрактного класса не существуют, можно создать указатели и ссылки на абстрактный класс. Это позволяет применять абстрактные классы для поддержки динамического полиморфизма и выбирать соответствующую виртуальную функцию в зависимости от типа указателя или ссылки.

При определении абстрактного класса необходимо иметь в виду следующее:

- 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.

Таким образом, можно создать функцию, параметром которой является указатель на абстрактный класс. На место этого параметра при выполнении программы может передаваться указатель на объект любого производного класса. Это позволяет создавать полиморфные функции, работающие с объектом любого тина в пределах одной иерархии.

Виртуальные функции

Практическое занятие: Вызов виртуальной функции с помощью указателя на объект базового класса

```
#include <iostream>
using namespace std;
class base {   public:
  virtual void vfunc() {
    cout<<"Функция vfunc() из класса base.\n"; }  };
class derived1 : public base {    public:
  void vfunc() {
    cout << "Функция vfunc() из класса derived1.\n";
                    }    };
class derived2 : public base {    public:
  void vfunc() {
    cout << "Функция vfunc() из класса derived2.\n";
                    }    };
int main()
{   base *p, b;  derived1 d1;  derived2  d2;
  // Указатель на объект базового класса
  p = &b;
 p->vfunc(); // Вызов функции vfunc() из класса base
  // Указатель на объект класса derivedl
  p = &d1;
  p->vfunc(); // Вызов функции vfunc() из класса
                          derivedl
  // Указатель на объект класса derived2
  p = &d2;
  p->vfunc();  // Вызов функции vfunc() из класса
                          derived2
  return 0;  }
```

Эта программа выводит на экран следующие строки:

- Функция vfunc() из класса base.
- Функция vfunc() из класса derived1.
- Функция vfunc() из класса derived2.

Как показывает эта программа, внутри класса base объявлена виртуальная функция vfunc(). Обратите внимание на ключевое слово virtual в объявлении функции. При переопределении функции vfunc() в классах derived1 и derived2 ключевое слово virtual не требуется. (Однако его использование не является ошибкой, просто оно не обязательно.)

В данной программе классы derived1 и derived2 являются производными от класса base. Внутри каждого из этих классов функция vfunc() переопределяется заново в соответствии с новым предназначением. В программе main() объявлены четыре переменные:

Имя	Тип
p	Указатель на базовый класс
b	Объект базового класса
d1	Объект класса derivedl
d2	Объект класса derived2

Виртуальные функции

Практическое занятие:

Вызов виртуальной функции с помощью указателя на объект базового класса

Указателю p присваивается адрес объекта b, а функция vfunc() вызывается с помощью указателя p. Поскольку указатель p ссылается на объект класса base, выполняется вариант функции vfunc() из базового класса.
Затем указателю p присваивается адрес объекта d1, и функция vfunc() снова вызывается с его помощью. На этот раз указатель p ссылается на объект класса derived1. Следовательно, вызывается функция derived1::vfunc(). В результате указателю p присваивается адрес объекта d2, поэтому выражение p->vfunc() приводит к вызову функции vfunc() из класса derived2. Принципиально важно, что вариант вызываемой функции определяется типом объекта, на который ссылается указатель p. Кроме того, выбор происходит в ходе выполнения программы, что обеспечивает основу динамического полиморфизма.

Виртуальную функцию можно вызывать обычным способом, используя имя объекта и оператор однако полиморфизм достигается только при обращении к ней через указатель.
Например, следующий фрагмент программы является совершенно правильным:

```
d2.vfunc();  // Вызывается функция vfunc() из класса derived2.
```

Несмотря на то что такой вызов виртуальной функции ошибкой не является, никаких преимуществ он не предоставляет.

Виртуальные функции

На первый взгляд, переопределение виртуальной функции в производном классе мало отличается от обычной перегрузки функций. Однако это не так, и термин перегрузка неприменим к переопределению виртуальных функций по нескольким причинам.

Наиболее важное отличие заключается в том, что прототип переопределяемой виртуальной функции должен точно совпадать с прототипом, определенным в базовом классе.

Этим виртуальные функции отличаются от перегруженных, которые отличаются типами и количеством параметров. (Фактически при перегрузке функций типы и количество их параметров должны отличаться! Именно эти отличия позволяют компилятору выбирать правильный вариант перегруженной функции.)

При переопределении виртуальной функции все аспекты их прототипов должны быть одинаковыми. Если не соблюдать это правило, компилятор будет считать эти функции просто перегруженными, а их виртуальная природа будет потеряна.

Второе важное ограничение заключается в том, что виртуальные функции не могут быть статическими членами классов. Кроме того, они не могут быть дружественными функциями. И, наконец, конструкторы не могут быть виртуальными, хотя на деструкторы это ограничение не распространяется.

Из-за перечисленных ограничений для переопределения виртуальной функции в производном классе используется термин замещение (overriding).

Виртуальные функции

Практическое занятие: Наследование атрибута virtual

```
#include <iostream>
using namespace std;
class base {  public:
  virtual void vfunc()
   {cout << "Функция vfunc()  из класса base\n";} };
class derived1 : public base {  public:
  void vfunc()
  {cout<<"Функция vfunc()  из класса derived1\n";}  };
/*Класс derived2 наследует виртуальную функцию
                    vfunc() от класса derived1 */
class derived2 : public derived1 {  public:
  // Функция vfunc()  остается виртуальной
  void vfunc()
   {cout<<"Функция vfunc()  из класса derived2.\n";} };
int main()
{  base *p, b;  derived1 d1;   derived2 d2;
  // Указатель на объект класса base
 p = &b;
 p->vfunc();  // Вызов функции vfunc() из класса base
 // Указатель на объект класса derived1
 p = &d1;
 p->vfunc(); //Вызов функции vfunc() из класса derived1
  // Указатель на объект класса derived1
 p = &d2;
 p->vfunc(); //Вызов функции vfunc() из класса derived2
  return 0;
}
```

При наследовании виртуальной функции ее виртуальная природа также наследуется. Это значит, что если производный класс, унаследовавший виртуальную функцию от базового класса, становится базовым по отношению к другому производному классу, виртуальная функция может по-прежнему замещаться. Иначе говоря, не имеет значения, сколько раз наследовалась виртуальная функция, она все равно остается виртуальной.

Эта программа выводит на экран следующие строки:

- Вызов функции vfunc() из класса base.
- Вызов функции vfunc() из класса derived1.
- Вызов функции vfunc() из класса derived2.

В данном случае класс derived2 является наследником класса derived1, а не класса base, но функция vfunc() остается виртуальной.

Виртуальные функции

Практическое занятие: Иерархичность виртуальных функций

```
#include <iostream>
using namespace std;
 class base {   public:
  virtual void vfunc()
  {cout<<"Функция vfunc()  из класса base\n"; }  };
 class derived1 : public base {    public:
  void vfunc()
  {cout<<"Функция vfunc()  из класса derived1\n";}  };
 class derived2 : public derived1 {    public:
/* Функция vfunc()  не замещается в классе derived2.
Поскольку класс derived2 наследник класса derived1,
вызывается функция vfunc()  из класса derivedl */
};
int main()
{   base *p, b;   derived1 d1;   derived2 d2;
  // Указатель на объект класса base
 p = &b;
 p->vfunc(); //
  // Указатель на объект класса derived1
 p = &d1;
 p->vfunc();//Вызов функции vfunc()  из класса derived1
   // Указатель на объект класса derived2
 p = &d2;
 p->vfunc();//Вызов функции vfunc()  из класса derived1
  return 0;
}
```

Наследование в языке C++ организовано по иерархическому принципу, поэтому виртуальные функции также должны быть иерархическими.

Это значит, что если виртуальная функция не замещается, вызывается ее предыдущая переопределенная версия. Например, в этой программе класс derived2 является наследником класса derived1, который, в свою очередь, является производным от класса base. Однако функция vfunc() в классе derived2 не замещается. Следовательно, ближайшая к классу derived2 версия функции vfunc() определена в классе derived1. Таким образом, вызов функции vfunc() с помощью объекта класса derived2 относится к функции derivedl::vfunc().

Программа выводит на экран:

- Функция vfunc() из класса base.
- Функция vfunc() из класса derived1.
- Функция vfunc() из класса derived1.

Виртуальные функции

Практическое занятие: **Чисто виртуальные функции**

Если виртуальная функция не замещается в производном классе, вызывается ее версия из базового класса. Однако во многих случаях невозможно создать разумную версию виртуальной функции в базовом классе.

Например, базовый класс может не обладать достаточным объемом информации для создания виртуальной функции.

Кроме того, в некоторых ситуациях необходимо гарантировать, что виртуальная функция будет замещена во всех производных классах. Для этих ситуаций в языке C++ предусмотрены чисто виртуальные функции.

Чисто виртуальная функция (pure virtual function) – это виртуальная функция, не имеющая определения в базовом классе. Для объявления чисто виртуальной функции используется следующая синтаксическая конструкция:

```
virtual тип имя_функции(список_параметров) = 0;
```

Чисто виртуальные функции должны переопределяться в каждом производном классе, в противном случае возникнет ошибка компиляции.

Виртуальные функции

Практическое занятие: Чисто виртуальные функции

```
#include <iostream>
using namespace std;
class number {  protected:    int val;
public:  void setval(int i)   { val = i; }
  // Функция show()  является чисто виртуальной
  virtual void show() = 0;    };
class hextype : public number {  public:
  void show()
   { cout << hex << val << "\n"; }    };
class dectype : public number {   public:
  void show()
   {  cout << val << "\n";  }  };
class octtype : public number {  public:
  void show()
   {  cout << oct << val << "\n"; }  };

int main()
{
 dectype d;
 hextype h;
 octtype o;
 d.setval(20);
 d.show();  // Выводит десятичное число 20
 h.setval(20);
h.show(); //Выводит шестнадцатеричное число 14
 o.setval(20);
 o.show();  // Выводит восьмеричное число 24
  return 0;
}
```

Эта программа содержит простой пример чисто виртуальной функции.

Базовый класс number содержит целое число val, функцию setval() и чисто виртуальную функцию show().

Производные классы hextype, dectype и octtype являются наследниками класса number и переопределяют функцию show() так, что она выводит значение val в соответствующей системе счисления (шестнадцатеричной, десятичной или восьмеричной).

Несмотря на простоту этого примера, он достаточно ярко иллюстрирует ситуацию, когда в базовом классе невозможно дать осмысленное определение виртуальной функции. В данном случае класс number просто обеспечивает единообразный интерфейс для использования производных типов. Функцию show() невозможно определить в классе number, поскольку в нем не задана основа системы счисления. Однако чисто виртуальная функция show() гарантирует, что в каждом производном классе она будет соответствующим образом переопределена.

Следует иметь в виду, что все производные классы обязаны переопределять чисто виртуальную функцию. Если этого не сделать, возникнет ошибка компиляции.

Виртуальные функции

Практическое занятие: Применение виртуальных функций

```
#include <iostream>
using namespace std;
class convert {   protected:
  double val1;  // Начальное значение
  double val2;  // Преобразованное значение
public:
  convert(double i) {  val1 = i;  }
  double getconv() { return val2; }
  double getinit() { return val1; }
virtual void compute() = 0;    };
// Преобразование литров в галлоны
class l_to_g : public convert {  public:
  l_to_g(double i) : convert(i) { }
  void compute() {val2 = val1 / 3.7854; }  };
// Преобразование шкалы Фаренгейта в шкалу Цельсия
class f_to_c : public convert {  public:
  f_to_c(double i) : convert(i) { }
  void compute() {val2 = (val1-32) /1.8;}   };
int main()
{ convert *p;  // Указатель на базовый класс
  l_to_g  lgob(4);       f_to_c  fcob(70);
 // Применение виртуальной функции
  p = &lgob;
  cout << p->getinit() << " литра(ов) равно ";
  p->compute();
  cout<<p->getconv()<<" галлонов\n";// л в гл
  p = &fcob;
  cout << p->getinit() << " по Фаренгейту = ";
  p->compute();
 // F^0 в C^0
cout << p->getconv() << " по Цельсию\n";
   return 0;  }
```

В примере иерархия классов, выполняющих преобразование единицы измерения из одной системы мер в другую (например, литров – в галлоны).

В базовом классе convert объявлены две переменные – val1 и val2, в которых хранятся исходное и преобразованное значения соответственно. Кроме того, в нем определены функции getinit() и getconv(), возвращающие эти значения. Эти члены класса convert фиксированы и могут использоваться во всех производных классах. Однако функция compute(), выполняющая фактическое преобразование, является чисто виртуальной и должна определяться во всех классах, производных от класса convert. Конкретный смысл функции compute() зависит от типа выполняемого преобразования.

В этой программе создаются классы l_to_g и f_to_c, производные от класса convert. Эти классы выполняют преобразования объема, измеренного в литрах, в объем, выраженный в галлонах, а также переводят шкалу температур по Фаренгейту в шкалу по Цельсию соответственно.

Каждый производный класс замещает функцию compute() по-своему, выполняя необходимое преобразование.

Однако, несмотря на различие фактических преобразований (т.е. методов) в классах l_to_g и f_to_c, их интерфейс одинаков.

Виртуальные функции

Практическое занятие: Применение виртуальных функций

```
// Преобразование футов в метры
class f_to_m : public convert
 {
  public:
    f_to_m(double i) : convert(i) { }
    void compute()
     {
       val2 = val1 / 3.28;
     }
 };
```

Виртуальные функции позволяют очень легко реагировать на новую ситуацию.

Предположим, что в предыдущей программе необходимо предусмотреть преобразование футов в метры, включив его в класс convert.

Базовый класс convert служит примером абстрактного класса. Виртуальная функция compute() не определяется в классе convert, поскольку в нем нет информации о выполняемом преобразовании. Функция compute() наполняется конкретным смыслом лишь в производных классах.

Абстрактные классы и виртуальные функции позволяют создавать библиотеки классов (class library), носящие обобщенный характер. Любой программист может создать класс, производный от библиотечного, добавив свои собственные функции. При этом будет сохранен единообразный интерфейс, определенный базовым классом
Таким образом, библиотечные классы можно адаптировать к новым ситуациям.

Виртуальные функции

Практическое занятие: **Применение виртуальных функций**

Пример:

```
class B { . . . public:   ... virtual void fun(); };
class D1 : public B {...public: virtual void fun();}; // замещение функции из B
class D2 : public B {…public: ... void fun();};     /* слова virtual нет, но замещение тоже есть */
void work (B* b) { b->fun(); ...}  /* функция может работать с объектами как базового, так и любого из производных классов */
B b;   D1 d1;   D2 d2;
B* pb=&b;              // указатель базового класса указывает на объект базового класса B
work(pb);              //будет вызвана функция B::fun
pb=&d1;          // настроили указатель базового класса на объект производного класса D1
work(pb);              //будет вызвана функция D1::fun
pb=&d2;          // настроили указатель базового класса на объект производного класса D2
work(pb);          //будет вызвана функция D2::fun
```

Виртуальные функции

Практическое занятие: Применение виртуальных функций

Рассмотрим основные случаи, вызывающие обычно сложности при изучении виртуальных функций.

Виртуальные функции. Примеры. Нормальная виртуальность.

	class B { public:	class D: public B {	D d; B*pb=&d; // указатель на базовый класс указывает на объект произв. класса
1	virtual void f1();	<virtual> void f1();	pb->f1(); // D::f1 - полное совпадение профиля функции
2	virtual void f2()=0;	<virtual> void f2();	pb->f2(); // D::f2 - замещение чисто виртуальной функции
3	virtual void f3(B*);	virtual void f3(D*);	pb->f3(&d); // D::f3 - допустимое различие типов параметров: указатель на базовый – указатель на производный
4	virtual void f4(B&);	virtual void f4(D&);	pb->f4(d); // D::f4 - допустимое различие типов параметров: ссылка на базовый – ссылка на производный
5	virtual B* f5();	virtual D* f5();	pb->f5(); // D::f5 - допустимое различие типов возвращаемого значения: указатель на базовый - указатель на производный
6	virtual B& f6();	virtual D& f6();	pb->f6(); // D::f6 - допустимое различие типов возвращаемого значения: ссылка на базовый - ссылка на производный
7	virtual ~B();	virtual ~D();	Деструкторы виртуальны, хотя их имена, разумеется, всегда различны. D* pd=new D; pb=pd; ....delete pb; // удаляет объект полностью, а не только «срез» базового класса

Виртуальные функции

Практическое занятие: **Применение виртуальных функций**

Виртуальные функции. Примеры. Виртуальность и её отсутствие

	class B { public:	class D: public B {	D d; B*pb=&d; // указатель на базовый класс указывает на объект произв. класса
1	virtual void f1();	virtual int f1();	Error! Различаются типы возвращаемых значений, совпадает только сигнатура функции, а для виртуальности необходимо полное совпадение профиля. Совпадение сигнатур не позволяет различать эти функции при перегрузке. Ошибка времени компиляции.

2	virtual void f2();	virtual void f2() const;	pb->f2(); // B::f2 функции различны по сигнатуре => нет виртуальности
			const B* pbc =&d;
			pbc->f2(); // Error! В классе B нет функции, которую можно вызвать для const-объекта.
			const D dc; pbc=&dc;
			pbc->f2(); // Error! (то же)

3	virtual void f3() const;	virtual void f3();	pb->f3(); // B::f3 const - функции различны по сигнатуре => нет виртуальности
			const B* pbc =&d;
			pbc->f3(); // B::f3 const
			const D dc; pbc=&dc;
			pbc->f3(); // B::f3const
			(B* pb=&dc; //Error! Типы указателей различны)

4	void f4();	virtual void f4();	pb->f4(); // B::f4 - нет виртуальности

Виртуальные функции

Практическое занятие: **Применение виртуальных функций**

Виртуальные функции. Примеры. Виртуальность и её отсутствие *продолжение*

	class B { public:	class D: public B {	D d; B*pb=&d; // указатель на базовый класс указывает на объект произв. класса
5	virtual void f5();	Нет такой функции	pb->f5 (); // B:: f5 - нет виртуальности
6	Нет такой функции	virtual void f6();	pb->f6(); // Error! нет такой функции в базовом классе, нет виртуальности

7	virtual void f7();	virtual void f7(int);	pb->f7(); // B::f7 - нет виртуальности
			pb->f7(1); // Error! в B нет f7(int)

8	virtual
	void f8 (double);
		virtual void f8(int);	pb->f8(l); // B::f8(double)
			pb->f8(1.5); // B::f8(double)
			нет виртуальности,
			в базовом классе ищется наилучшим образом подходящая функция

9	private: virtual void f9():	public: virtual void f9():	pb->f9(); // Error! f9() - private в B
10	public: virtual void fl0();	private: (protected:) virtual void fl0();	pb->fl0(); // D::f10 - есть виртуальность, получили доступ к закрытой (защищенной) функции. Опасно!

Виртуальные функции

Практическое занятие: **Применение виртуальных функций**

Виртуальные функции. Виртуальность и значения параметров по умолчанию

```
class A { int ax;
    public:   A (int n = 1)    { ax = n; }
      virtual int  f1(int a = 5, int b = 5) { cout<< " A::f1 "; return 0;}
       virtual int  f2(int x = 10) { cout<<" A::f2 x= "<<x<<endl; return 0;
}
          };
class C : public A { int cx;
                public:   C ( int n = 3 )  { cx = n; };
                  int f1(int a, int b = 0)
                {cout<< " C::f1  a= "<<a<<" b= "<<b<<endl; return 0;};
              int f2(int x)  {cout<< " C::f2  x= "<<x<<endl; return 0; }
                           };
int main() {
A * p; C c;
p=&c;          // Здесь функции виртуальные и вызываются как виртуальные - по типу объекта
p->f1();       // C::f1  a=5   b=5    значения параметров по умолчанию, заданные в функции базового
          // класса, передаются  в  функцию производного класса
p->f1(2);       // C::f1  a= 2   b=5
p->f1(2,2);    // C::f1   a= 2   b=2
p->f2();       // C::f2  x= 10
p->f2(3);      // C::f2  x= 3
          }
```

Виртуальные функции

Практическое занятие: **Использование абстрактных классов**

```
#include <stdio.h>
#include <iostream.h>
#include <string.h>
#include <conio.h>
class File
{ protected:
  char **str;
  char Name[30];
  int n;
  virtual int ReadFile()=0;
  public:
  File(char*,int);
  ~File();
  void display();
};
File::File(char * FileName,int k)
{ strcpy(Name,FileName);
  n=k;
  str=new char*[n];
  for(int i=0;i<n;i++)
   str[i]=new char[80];
}          // см. продолжение
```

```
                                  // продолжение

void File::display()
{ clrscr();
  int k=ReadFile();
  for(int i=0;i<k;i++)
   cout<<str[i];
  getch();
}
struct info
{char name[20];
 char numb[10];
 float value;
};
class InfoFile:public File
{ int ReadFile();
  public:
  InfoFile(char* st,int k):File(st,k)  {}
  void WriteFile(int );
};
                                  // см. продолжение
```

Виртуальные функции

Практическое занятие: **Использование абстрактных классов**

```
int InfoFile::ReadFile()                    // продолжение
{ FILE *fp;
  int i=0;
  info x;
  if((fp=fopen(Name,"r"))!=NULL)
   { while(!feof(fp))
      { fread(&x,sizeof(info),1,fp);
     sprintf(str[i],”Запись N  %d  %s  %s  %f\n",i+1,x.name,x.numb,x.value);
        i++;
      }
     fclose(fp);
     return i-1;
    }
  return 0;}
void InfoFile::WriteFile(int k)
{ info x;
  FILE *fp;
  if((fp=fopen(Name,"w+"))!=NULL)
  { for(int i=0;i<k;i++)
   {cout<<”Введите "<<i+1<<" -ю запись";
    cin>>x.name>>x.numb>>x.value;
    fwrite(&x,sizeof(info),1,fp);
   }
   fclose(fp);
  }}                                        // см. продолжение
```

Виртуальные функции

Практическое занятие: **Использование абстрактных классов**

```
        // продолжение
class HelpFile:public File
{ int ReadFile();
  public:
  HelpFile():File("help.dat",25){}
};
int HelpFile::ReadFile()
{  int i=0;
   FILE *fp;
   if((fp=fopen(Name,"r"))!=NULL)
   { while(!feof(fp))
     {fgets(str[i],80,fp);
      i++;
     }
    fclose(fp);
    return i-1;
   }
   return 0;}
        // см. продолжение
```

```
        // продолжение
main()
{
  HelpFile hp;
  hp.display();
  InfoFile If("info",40);
  If.WriteFile(3);
  If.display();
}
```

Виртуальные функции

Практическое занятие: Использование абстрактных классов

Программа, приведенная в листинге, работает с файлами двух типов – информационным файлом (класс InfoFile), предназначенным для хранения простейшей базы данных – нескольких структур типа info, а также файлом помощи (класс HelpFile), который хранит текстовую информацию, предположительно – справку о самой программе. Общая сущность двух этих типов файлов, выражающаяся в имени файла Name, буфере str для временного хранения информации из файла, размере используемого файлом буфера n, а также метода display вывода считанной из файла информации на экран, выделена в родительский класс (класс File). Схема иерархии классов программы приведена на рисунке.

Метод display класса File считывает информацию из файла методом ReadFile в буфер и построчно выводит ее на экран. Однако, метод ReadFile в самом классе File не может быть полноценно определен, поскольку этот класс в терминах предметной области является абстрактной основой для двух других классов (InfoFile и HelpFile) и для него не известен, например, тип хранящейся в файле информации.

Метод ReadFile определен в классах InfoFile и HelpFile, причем в первом данный метод считывает из файла информацию в виде экземпляров структуры info, преобразует ее в текстовую форму и записывает в буфер, а во втором - информация непосредственно считывается в виде текстовых строк, которые записываются в буфер str. Таким образом, метод ReadFile не надо определять в классе File по логике представления предметной области, однако обязательно необходимо определить по правилам синтаксиса языка C++ (так как метод display использует этот метод). При этом необходимо обеспечить полиморфное поведение метода ReadFile для того, чтобы при вызове метода display объектом класса InfoFile в теле метода display была вызвана реализация метода ReadFile для соответствующего класса. Поэтому метод ReadFile объявлен в классе File как чистая виртуальная функция:

```
virtual int ReadFile()=0;
```

Таким образом, класс File является абстрактным классом, для которого запрещено создание объектов, то есть ошибкой будет такое объявление:

```
File MyFile("file.txt",50);
```

Класс File может быть только основой для дальнейшего наследования другими классами. Подводя итог, можно сказать, что абстрактные классы используются для спецификации интерфейсов операций (методы, реализующие эти операции впоследствии определяются в производных классах абстрактного класса). Абстрактные классы удобны на фазе анализа требований к системе, так как они позволяют выявить аналогию в различных, на первый взгляд, операциях, определенных в анализируемой системе.

Шаблоны функций и классов

Домашние задания

1. **Индивидуальные задания на виртуальные функций и абстрактные классы каждому студенту.**