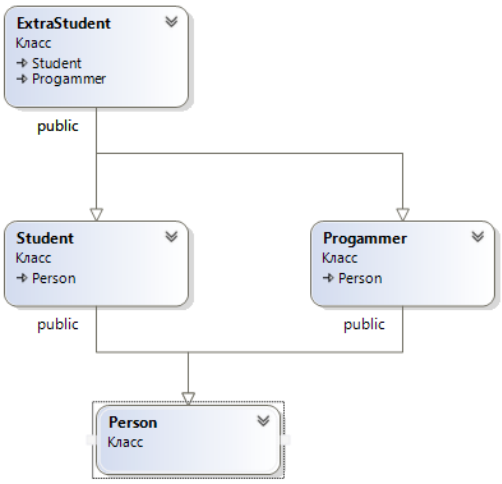
№ 4 Создание и взаимодействие объектов, абстрактные классы

Использовать проект созданный в практикуме №3.

1) Для вложенного класса допишите метод, который вывызвает метод внешего класса. Во внешнем классе допишите метод, который проверяет одно из полей внутреннего класса.

2) Расширить иерархию классов с использованием абстактного класса в качестве основы иерархии. Это может быть новый класс или выбран класс из имеющихся.

Напрмер:



3) Расширьте иерархию двумя новыми классами. Первый класс должен содержать дополнительные поля public и protected и protected наследоваться. Второй производний от первого public наследоваться. Исследовать возможность вызова наследуемых методов и получение доступа к полям.

```
class Person { ... };
class Student :public Person {};
class Progammer :public Person {};
class ExtraStudent : public Student, public Progammer {};

class Listener : protected ExtraStudent {
    public: int expirience;
    protected: std::string organization;
};
class MainListener : public Listener {};
```

4) В одном из классов объявите константное поле (например, numberOfQuestions, size и т.п.) и константые методы, продемонстируйте их использование: возможности инициализации, вызовы.

5) Создайте класс Instector дружественный одному из классов. В Instectore определите метод iKnowAllAboutYou(), который выводит на консоль (в файл) значения всех приватных полей друга. Определите класс Curator с методом дружественным вашему классу iCanModify (…) (не класс, а только метод), который может изменить значения приватных полей класса.

Например так:

```
class Person { ... };

class Student;

class Curator {
public:
    void iCanModify(Student& any) { ... }
};

class Inspector;

class Student :public Person {
private:
    int age =18;
    std::string name ="no";
    friend Inspector;
    friend  void  Curator::iCanModify(Student&);
public:
    void f(){}
};
```

```
class Inspector {
public:
    void    iKnowAllAboutYou(Student& any) {
        std::cout << any.name << " " << any.age << std::endl;
    }
};
int main()
{
    Student a;
    Inspector b;
    b.iKnowAllAboutYou(a);
    return 0;
}
```

6) В один из классов добавьте статическое поле objectCounter, который будет содержать количество созданных объектов класса (увеличивать в контрукторе и уменьшать в деструкторе). Создать метод для просмотра этого числа.

Например:

```
class Progammer :public Person {
private:
    static int counter;
public:
    Progammer() { counter++; }
    ~Progammer() { counter--; }
    static int getCounter() {
        return counter;
        }
    void f(){}
};
int Progammer::counter = 0;

int main()
{
    Progammer one;
    {
        std::cout << "Count of Programmers " << Progammer::getCounter();
        Progammer two;
        std::cout << "Count of Programmers " << Progammer::getCounter();

    }
    std::cout << "Count of Programmers " << one.getCounter();
    /* ... */
```

7) Во всех классах (иерархии) создать виртуальный метод toConsole(), который выводит информацию о типе объекта и его текущих значениях. Создайте дополнительный класс Printer с методом iAmPrinting(AbstractClass *someobj). Формальным параметром метода должен быть укзатель на абстактный класс. В методе iAmPrinting должен вызываться typeid, для определения типа объекта и метод объекта toConsole(). В демонстрационной программе (main) создать указатели на объекты всех ваших классов по иерархии (инициализировать указатели), объект класса Printer и последоватльно вызвать его метод iAmPrinting со всеми указателями в качестве аргументов. Пояснить результат.

```
class Person {
    public:
        virtual void toConsole() = 0;
    };
```

```
class Student :public Person {
private:
    int age =18;
    std::string name ="no";
    // …;
public:
```

```
	void toConsole()
	{
		std::cout << typeid(this).name() << " " << age << " " << 
name<<std::endl;
	}
	};
```

```
class Progammer :public Person {
private:
	int level = 1;
	// …;
public:
	// …;
	void toConsole()
	{
		std::cout << typeid(this).name() << " " << level<<std::endl;
	}
};
```

```
int main()
{
	Person *some = new Progammer;
	Student *anna = new Student;
	Printer alluse;
	alluse.iAmPrinting(anna);
	alluse.iAmPrinting(some);
};
```

8) Используя const_cast, static_cast, dynamic_cast (повышающее, понижающее и перекрестное преобразование между классами в вашей иерархии) написать демонстрацию использования операторов приведения типа.

P.S. Код, в приведенных примерах упрощен.

Теория

Виртуальные функции.

К механизму виртуальных функций обращаются в тех случаях, когда в каждом производном классе требуется свой вариант некоторой компонентной функции. Классы включающие такие функции называются полиморфными и играют особую роль в ООП.
Рассмотрим как ведут себя при наследовании не виртуальные компонентные функции с одинаковыми именами, типами и сигнатурами параметров.

Пример.

```
class Base
{
public:
  void print(void)
  {
    cout << "\nbase";
  }
}

class Dir : public Base
{
public:
  void print(void)
  {
    cout << "\ndir";
  }
};

int main(void)
{
  Base B, *bp = &B;
  Dir D, *dp = &D;
  Base *p = &D;
  bp->print(); // base
  dp->print(); // dir
  p->print();  // base
  return 0;
}
```

В последнем случае вызывается функция print базового класса, хотя указатель p настроен на объект производного класса. Дело в том, что выбор нужной функции выполняется при компиляции программы и определяется типом указателя, а не его значением. Такой режим называется ранним или статическим связыванием.

Большую гибкость обеспечивает позднее (отложенное) или динамическое связывание, которое предоставляется механизмом виртуальных функций. Любая нестатическая функция базового класса может быть сделана виртуальной, для чего используется ключевое слово virtual.

Пример.

```
class Base
{
public:
virtual void print(void)
{
cout << "\nbase";
}
...
};
// и так далее – см. предыдущий пример.
```

В этом случае будет напечатано

```
base
dir
dir
```

Т.о. интерпретация каждого вызова виртуальной функции через указатель на базовый класс зависит от значения этого указателя, т.е. от типа объекта, для которого выполняется вызов.

Виртуальные функции - это функции, объявленные в базовом классе и переопределенные в производных классах. Иерархия классов, которая определена открытым наследованием, создает родственный набор пользовательских типов, на все объекты которых может указывать указатель базового класса. Выбор того, какую виртуальную функцию вызвать, будет зависеть от типа объекта, на который фактически (в момент выполнения программы) направлен указатель, а не от типа указателя.

- ✓ Виртуальными могут быть только нестатические функции-члены.
- ✓ Виртуальность наследуется. После того как функция определена как виртуальная, ее повторное определение в производном классе (с тем же самым прототипом) создает в этом классе новую виртуальную функцию, причем спецификатор virtual может не использоваться.
- ✓ Конструкторы не могут быть виртуальными, в отличие от деструкторов. Практически каждый класс, имеющий виртуальную функцию, должен иметь виртуальный деструктор.

- ✓ Если в производном классе ввести функцию с тем же именем и типом, но с другой сигнатурой параметров, то эта функция производного класса не будет виртуальной.
- ✓ Виртуальная функция может быть дружественной в другом классе.
- ✓ Механизм виртуального вызова может быть подавлен с помощью явного использования полного квалифицированного имени.

Абстрактные классы.

Абстрактным называется класс, в котором есть хотя бы одна чистая (пустая) виртуальная функция.
Чистой виртуальной называется компонентная функция, которая имеет следующее определение:
virtual тип имя_функции(список_формальных_параметров)= 0;
Чистая виртуальная функция ничего не делает и недоступна для вызовов. Ее назначение – служить основой для подменяющих ее функций в производных классах. Абстрактный класс может использоваться только в качестве базового для производных классов.
Механизм абстрактных классов разработан для представления общих понятий, которые в дальнейшем предполагается конкретизировать. При этом построение иерархии классов выполняется по следующей схеме. Во главе иерархии стоит абстрактный базовый класс. Он используется для наследования интерфейса. Производные классы будут конкретизировать и реализовать этот интерфейс. В абстрактном классе объявлены чистые виртуальные функции, которые по сути есть абстрактные методы.
Пример.

```
class Base
{
public:
  Base();                       // конструктор по умолчанию
  Base(const Base &);           // конструктор копирования
  virtual ~Base();              // виртуальный деструктор
  virtual void Show(void) = 0; // чистая виртуальная функция
  // другие чистые виртуальные функции
protected:
  // защищенные члены класса
private:
  // часто остается пустым, иначе будет мешать будущим разработкам
};

class Derived : virtual public Base
{
public:
  Derived();                 // конструктор по умолчанию
  Derived(const Derived &); // конструктор копирования
  Derived(параметры);        // конструктор с параметрами
```

```
  virtual ~Derived();       // виртуальный деструктор
  void Show(void);          // переопределенная виртуальная функция
  // другие переопределенные виртуальные функции
  Derived &operator =(const Derived &); // перегруженная операция
присваивания
  // другие перегруженные операции
protected:
  // используется вместо private, если ожидается наследование
private:
  // используется для деталей реализации
};
```

По сравнению с обычными классами абстрактные классы пользуются "ограниченными правам". А именно:

- ✓ невозможно создать объект абстрактного класса;
- ✓ абстрактный класс нельзя употреблять для задания типа параметра функции или типа возвращаемого функцией значения;
- ✓ абстрактный класс нельзя использовать при явном приведении типов; в то же время можно определить указатели и ссылки на абстрактный класс.

Объект абстрактного класса не может быть формальным параметром функции, однако формальным параметром может быть указатель на абстрактный класс. В этом случае появляется возможность передавать в вызываемую функцию в качестве фактического параметра значение указателя на производный объект, заменяя им указатель на абстрактный базовый класс. Таким образом мы получаем полиморфные объекты.

Включение объектов.

Есть два варианта включения объекта типа X в класс A:

Объявить в классе A член типа X;

```
class A
{
  X x;
  ...
};
```

Объявить в классе A член типа X* или X&.

```
class A
{
  X *p;
  X &r;
  ...
};
```

Предпочтительно включать собственно объект как в первом случае. Это эффективнее и меньше подвержено ошибкам, так как связь между содержащимся и содержащим объектами описывается правилами конструирования и уничтожения.

Например,

```
// Персона
class Person
{
public:
  Person(char *);
  ...
protected:
  char *name;
};

// Школа
class School
{
public:
  School(char *name)
  : head(name)
  {}
  ...
protected:
  Person head; // директор
};
```

Второй вариант с указателем можно прменять тогда, когда за время жизни "содержащего" объекта нужно изменить указатель на "содержащийся" объект.

Например,

```
class School
{
public:
  School(char *name)
  : head(new Person(name))
  {}
  ~School()
  {
    delete head;
  }
  Person *change(char *newname)
  {
    Person *temp = head;
    head = new Person(newname);
    return temp;
```

```
  }
  ...
protected:
  Person *head; // директор
};
```

Второй вариант можно использовать, когда требуется задавать "содержащийся" объект в качестве аргумента.

Например,

```
class School
{
public:
  School(Person *q)
  : head(q)
  {}
  ...
protected:
  Person *head;// директор
};
```

Имея объекты, включающие другие объекты, мы создаем иерархию объектов. Она является альтернативой и дополнением к иерархии классов. А как быть в том случае, когда количество включаемых объектов заранее неизвестно и (или) может изменяться за время жизни "содержащего" объекта. Например, если объект School содержит учеников, то их количество может меняться.
Существует два способа решения этой проблемы. Первый состоит в том, что организуется связанный список включенных объектов, а "содержащий" объект имеет член-указатель на начало этого списка.

Например,

```
class Person
{
  char *name;
  Person *next;
  ...
};

class School
{
public:
  School(char *name)
  : head(new Person(name)),
    begin(NULL)
```

```
  {}
  ~School();
  void add(Person *ob);
  ...
protected:
  Person *head;  // указатель на директора школы
  Person *begin; // указатель на начало списка учеников
};
```

В этом случае при создании объекта School создается пустой список включенных объектов. Для включения объекта вызывается метод add(), которому в качестве параметра передается указатель на включаемый объект. Деструктор последовательно удаляет все включенные объекты. Объект Person содержит поле next, которое позволяет связать объекты в список.

Второй способ заключается в использовании специального объекта-контейнера.

Контейнерный класс предназначен для хранения объектов и представляет удобные простые и удобные способы доступа к ним.

```
class School
{
  Person *head;
  Container pupil;
  ...
};
```

Здесь pupil - контейнер, содержащий учеников. Все, что необходимо для добавления, удаления, просмотра и т.д. включенных объектов, должно содержаться в методах класса Container. Примером могут служить контейнеры стандартной библиотеки шаблонов (STL) C++.

Наряду с контейнерами существуют ***группы***, т.е. объекты, в которые включены другие объекты. Объекты, входящие в группу, называются элементами группы. Элементы группы, в свою очередь, также могут быть группой.

Примеры групп:

- ✓ Окно в интерактивной программе, которое владеет такими элементами, как поля ввода и редактирования данных, кнопки, списки выбора, диалоговые окна и т.д.
- ✓ Агрегат, состоящий из более мелких узлов.
- ✓ Огород, состоящий из растений, системы полива и плана выращивания.
- ✓ Некая организационная структура (например, ФАКУЛЬТЕТ, КАФЕДРА, СТУДЕНЧЕСКАЯ ГРУППА).

Понятия "группа" от "контейнер" отличаются. Контейнер используется для хранения других данных. Пример контейнеров: объекты контейнерных классов библиотеки STL в C++ (массивы, списки, очереди).
В отличие от контейнера группа есть класс, который не только хранит объекты других классов, но и обладает собственными свойствами, не вытекающими из свойств его элементов.
Группа дает второй вид иерархии (первый вид - иерархия классов, построенная на основе наследования) - иерархию объектов (иерархию типа целое/часть), построенную на основе агрегации. Реализовать группу можно несколькими способами:
Класс "группа" содержит поля данных объектного типа. Таким образом, объект "группа" в качестве данных содержит либо непосредственно свои элементы, либо указатели на них

```
class TWindowDialog : public TGroup
{
protected:
  TInputLine input1;
  TEdit edit1;
  TBuspanon buspanon1;
  /*другие члены класса*/
};
```

Такой способ реализации группы используется в C++Builder.
Группа содержит член-данное last типа TObject *, который указывает на начало связанного списка объектов, включенных в группу. В этом случае объекты должны иметь поле next типа TObject *, указывающее на следующий элемент в списке.
Создается связанный список структур типа TItem:

```
struct TItem
{
  TObject *item;
  TItem *next;
};
```

Поле item указывает на объект, включенный в группу. Группа содержит поле last типа TItem *, которое указывает на начало связанного списка структур типа TItem. Если необходим доступ элементов группы к ее полям и методам, объект типа TObject должен иметь поле owner типа TGroup *, которое указывает на собственника этого элемента.
Имеется два метода, которые необходимы для функционирования группы:
void Insert(TObject *p);
Вставляет элемент в группу.
void Show(void);
Позволяет просмотреть группу.

Кроме этого группа может содержать следующие методы:
int Empty(void);
Показывает, есть ли хотя бы один элемент в группе.
TObject *Delete(TObject *p);
Удаляет элемент из группы, но сохраняет его в памяти.
void DelDisp(TObject *p);
Удаляет элемент из группы и из памяти.

Итераторы позволяют выполнять некоторые действия для каждого элемента определенного набора данных.
Такой цикл мог бы быть выполнен для всего набора, например, что-бы напечатать все элементы набора, или мог бы искать некоторый элемент, который удовлетворяет определенному условию, и в этом случае такой цикл может закончиться, как только будет найден требуемый элемент.
Мы будем рассматривать итераторы как специальные методы класса-группы, позволяющие выполнять некоторые действия для всех объектов, включенных в группу. Примером итератора является метод Show.
Нам бы хотелось иметь такой итератор, который позволял бы выполнять над всеми элементами группы действия, заданные не одним из методов объекта, а произвольной функцией пользователя. Такой итератор можно реализовать, если эту функцию передавать ему через указатель на функцию.
Определим тип указателя на функцию следующим образом:

```
typedef  void (*PF)(TObject *, < дополнительные параметры>);
```

Функция имеет обязательный параметр типа TObject & или TObject *, через который ей передается объект, для которого необходимо выполнить определенные действия.
Метод-итератор объявляется следующим образом:

```
void TGroup::ForEach(PF action, < дополнительные параметры >);
```

где action - единственный обязательный параметр-указатель на функцию, которая вызывается для каждого элемента группы; дополнительные параметры - передаваемые вызываемой функции параметры.
Затем определяется указатель на функцию и инициализируется передаваемой итератору функцией.

```
PF pf = myfunc;
```

Тогда итератор будет вызываться, например, для дополнительного параметра типа int, так:

```
gr.ForEach(pf, 25);
```

Здесь gr - объект-группа.
Рассмотрим отношения между наследованием и включением.

Включение и наследование.

Пусть класс D есть производный класс от класса B.

```
class B {...};
class D : public B {...};
```

Слово public в заголовке класса D говорит об открытом наследовании. Открытое наследование означает что производный класс D является подтипом класса B, т.е. объект D является и объектом B. Такое наследование является отношением is-a или говорят, что D есть разновидность D. Иногда его называют также интерфейсным наследованием. При открытом наследовании переменная производного класса может рассматривается как переменная типа базового класса. Указатель, тип которого - "указатель на базовый класс", может указывать на объекты, имеющие тип производного класса. Используя наследование мы строим иерархию классов.

Рассмотрим следующую иерархию классов.

```
class Person
{
public:
  Person(char *, int);
  virtual void show(void) const;
  ...
protected:
  char *name;
  int age;
};

class Employee : public Person
{
public:
  Employee(char *, int, int);
  void show(void) const;
  ...
protected:
  int work;
};
class Teacher : public Employee
{
public:
  Teacher(char *, int, int);
  void show(void) const;
  ...
protected:
  int teacher_work;
};
```

Определим указатели на объекты этих классов.

```
Person *pp;
Teacher *pt;
```

Создадим объекты этих классов.

```
Person a("Петров", 25);
Employee b("Королев", 30, 10);
pt = new Teacher("Тимофеев", 45, 15);
```

Просмотрим эти объекты.

```
pp = &a;
pp->show(); // вызывает Person::show для объекта a
pp = &b;
pp->show(); // вызывает employee::show для объекта b
pp = pt;
pp->show(); // вызывает teacher::show для объекта *pt
```

Здесь указатель базового класса pp указывает на объекты производных классов Employee, Teacher, т.е. он совместим по присваиванию с указателями на объекты этих классов. При вызове функции show с помощью указателя pp, вызывается функция того класса, на объект которого фактически указывает pp. Это достигается за счет объявления функции show виртуальной, в результате чего мы имеем позднее связывание.

Пусть теперь класс D имеет член класса B.

```
class D
{
public:
  B b;
  ...
};
```

В свою очередь класс B имеет член класса C.

```
class B
{
public:
  C c;
  ...
};
```

Такое включение называют отношением has-a Используя включение мы строим иерархию объектов.

На практике возникает проблема выбора между наследованием и включением. Рассмотрим классы "Самолет" и "Двигатель". Новичкам часто

приходит в голову сделать "Самолет" производным от "Двигатель". Это не верно, поскольку самолет не является двигателем, он имеет двигатель. Один из способов увидеть это- задуматься, может ли самолет иметь несколько двигателей? Поскольку это возможно, нам следует использовать включение, а не наследование.

Рассмотрим следующий пример:

```
class B
{
public:
  virtual void f(void);
  void g(void);
};

class D
{
public:
  B b;
  void f(void);
};

void h(D *pd)
{
  B *pb;
  pb = pd;   // #1 Ошибка
  pb->g();   // #2 вызывается B::g()
  pd->g();   // #3 Ошибка
  pd->b.g(); // #4 вызывается B::g()
  pb->f();   // #5 вызывается B::f()
  pd->f();   // #6 вызывается D::f()
}
```

Почему в строках #1 и #3 ошибки?

В строке #1 нет преобразования D* в B*.

В строке #3 D не имеет члена g().

В отличие от открытого наследования, не существует неявного преобразования из класса в один из его членов, и класс, содержащий член другого класса, не замещает виртуальных функций того класса.

Если для класса D использовать открытое наследование

```
class D : public B
{
public:
```

```
  void f(void);
};

то функция
void h(D *pd)
{
  B *pb = pd;
  pb->g(); // вызывается B::g()
  pd->g(); // вызывается B::g()
  pb->f(); // вызывается D::f()
  pd->f(); // вызывается D::f()
}
```

не содержит ошибок.

Так как D является производным классом от B, то выполняется неявное преобразование из D в B. Следствием является возросшая зависимость между B и D.

Существуют случаи, когда вам нравится наследование, но вы не можете позволить таких преобразований.

Например, мы хотим повторно использовать код базового класса, но не предполагаем рассматривать объекты производного класса как экземпляры базового. Все, что мы хотим от наследования- это повторное использование кода. Решением здесь является закрытое наследование. Закрытое наследование не носит характера отношения подтипов, или отношения is-a. Мы будем называть его отношением like-a (подобный) или наследованием реализации, в противоположность наследованию интерфейса. Закрытое (также как и защищенное) наследование не создает иерархии типов.

С точки зрения проектирования закрытое наследование равносильно включению, если не считать вопроса с замещением функций. Важное применение такого подхода- открытое наследование из абстрактного класса, и одновременно закрытое (или защищенное) наследование от конкретного класса для представления реализации.

Пример. Бинарное дерево поиска

```
// Файл tree.h
// Обобщенное дерево
typedef void *TP; // тип обобщенного указателя

int comp(TP a, TP b);

class Node // узел
{
```

```
private:
  Node *left;
  Node *right;
  TP data;
  int count;
  Node(TP d, TP l, TP r)
  : data(d),
    left(l),
    right(r),
    count(1)
  {}
friend class Tree;
friend void print(Node *n);
};

class Tree // дерево
{
public:
  Tree()
  {
    root = 0;
  }
  void insert(TP d);
  TP find(TP d) const
  {
    return (find(root, d));
  }
  void print(void) const
  {
    print(root);
  }
protected:
  Node *root; // корень
  TP find(Node *r, TP d) const;
  void print (Node *r) const;
};
```

Узлы двоичного дерева содержат обобщенный указатель на данные data. Он будет соответствовать типу указателя в производном классе. Поле count содержит число повторяющихся вхождений данных. Для конкретного производного класса мы должны написать функцию comp для сравнения значений конкретного производного типа. Функция insert() помещает узлы в дерево.

```
void Tree::insert(TP d)
{
  Node *temp = root;
  Node *old;
  if (!root)
  {
    root = new Node(d, 0, 0);
    return;
```

```
  }
  while (temp)
  {
    old = temp;
    if (comp(temp->data, d) == 0)
    {
      (temp->count)++;
      return;
    }
    if (comp(temp->data, d) > 0)
    {
      temp = temp->left;
    }
    else
    {
      temp = temp->right;
    }
  }
  if (comp(old->data, d) > 0)
  {
    old->left = new Node (d, 0, 0);
  }
  else
  {
    old->right = new Node (d, 0, 0);
  }
}
```

Функция TP find(Node *r, TP d) ищет в поддереве с корнем r информацию, представленную d.

```
TP Tree::find(Node *r, TP d) const
{
  if (!r) return 0;
  if (comp(r->data, d) == 0) return (r->data);
  if (comp(r->data, d) > 0) return (find(r->left, d));
  else return (find(r->right, d));
}
```

Функция print() - стандартная рекурсия для обхода бинарного дерева

```
void Tree::print(Node *r) const
{
  if (r)
  {
    print (r->left);
    ::print(r);
    print (r->right);
  }
}
```

В каждом узле применяется внешняя функция ::print().

Теперь создадим производный класс, который в качестве членов данных хранит указатели на char.

```
// Файл StringTree.cpp
#include "tree.h"
#include <cstring>
class StringTree : private Tree
{
public:
  StringTree()
  {}
  void insert(char *d)
  {
    Tree::insert(d);
  }
  char *find(char *d) const
  {
    return (char *) Tree::find (d);
  }
  void print(void) const
  {
    Tree::print();
  }
};
```

В классе StringTree функция insert использует неявное преобразование char * к void *.

Функция сравнения comp выглядит следующим образом

```
int comp(TP a, TP b)
{
  return (strcmp((char *) a, (char *) b));
}
```

Для вывода значений, хранящихся в узле, используется внешняя функция

```
void print(Node *n)
{
  cout << (char *) (n->data) << endl;
}
```

Здесь для явного приведения типа void * к char * мы используем операцию приведения типа (имя_типа) выражение. Более надежным является использование оператора static_cast<char *> (TP).

Множественное наследование.

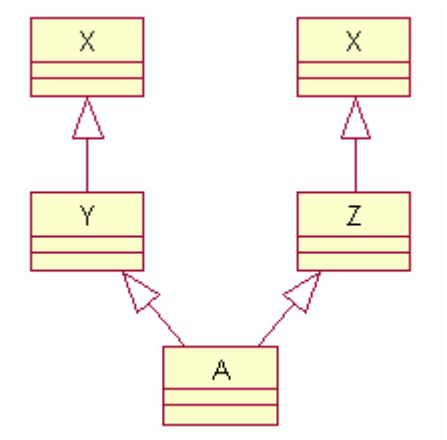
Класс может иметь несколько непосредственных базовых классов

```
class A1 {...};
class A2 {...};
class A3 {...};
class B : public A1, public A2, public A3 {...};
```

Такое наследование называется множественным. При множественном наследовании никакой класс не может больше одного раза использоваться в качестве непосредственного базового. Однако класс мужет больше одного раза быть непрямым базовым классом.

```
class X {... f(); ...};
class Y : public X {...};
class Z : public X {...};
class A : public Y, public Z {...};
```

Имеем следующую иерархию классов (и объектов):



Такое дублирование класса соответствует включению в производный объект нескольких объеатов базового класса. В этом примере существуют два объекта класса X. Для устранения возможных неоднозначностей нужно обращаться к конкретному компоненту класса X, используя полную квалификацию

```
Y::X::f() или Z::X::f()
```

Пример.

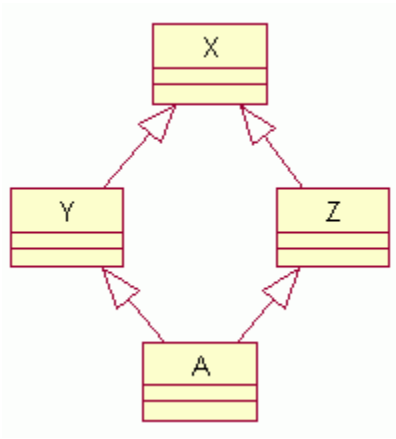
```
class Circle // окружность
{
public:
  Circle(int x1, int y1, int r1)
  {
```

```
    x = x1;
    y = y1;
    r = r1;
  }
  void show(void);
  ...
protected:
  int x, y, r;
};

class Square // квадрат
{
public:
  Square(int x1, int y1, int l0)
  {
    x = x1;
    y = y1;
    l = l1;
  }
  void show(void);
  ...
protected:
  int x, y, l;
  // x, y – координаты центра
  // l – длина стороны
};

class CircleSquare : public Circle, public Square // окружносrь,
вписанная в квадрат
{
public:
  CircleSquare(int x1, int y1, int r1)
  : Circle(x1, y1, r1),
    Square(x1, y1, 8 * r1)
  {...}
  void show(void)
  {
     Circle::show();
    Square::show();
  }
  ...
};
```

Чтобы устранить дублирование объектов непрямого базового класса при множественном наследовании, этот базовый класс объявляют виртуальным.

```
class X {...};
class Y : virtual public X {...};
class Z : virtual public X {...};
class A : public Y, public Z {...};
```

Теперь класс A будет включать только один экземпляр X, доступ к которому равноправно имеют классы Y и Z.



Пример.

```
class Base
{
  int x;
  char c, v[10];
  ...
};

class ABase : public virtual Base
{
  double y;
  ...
};

class BBase : public virtual Base
{
  float f;
  ...
};

class Top : public ABase, public BBase
{
  long t;
  ...
};

int main (void)
{
  cout << sizeof(Base) << endl;
  cout << sizeof(ABase) << endl;
  cout << sizeof(BBase) << endl;
  cout << sizeof(Top) << endl;
```

```
  return 8;
}
```

Здесь

объект класса Base занимает в памяти 15 байт:
4 байта - поле int;
2 байта - поле char;
10 байт - поле char[10];
объект класса ABase занимает в памяти 79 байт:
8 байт - поле double;
15 байт - поля базового класса Base;
2 байта - для связи в иерfрхии виртуальных классов;
объект класса BBase занимает в памяти 21 байт:
4 байта - поле float;
15 байт - поля базового класса Base; 2 байта - для связи в иерархии виртуальных классов;
объект класса Top занимает h памяти 35 байт:
4 байта - поле long;
10 байт - данные и связи ABase;
6 байт - davные и связи BBase;
15 байт - поля базового класса Base;
Если при наследовании Base в классах ABase и BBase базовый класс сделать не виртуальным, то результатb будут такими:
объект класса Base занимает в памяти 15 байт
объект класса ABase занимает в памяти 26 байта (нет 2-х байтов для связи);
объект класса BBase занимает в памяти 59 байт (нет 2-х байтов для связи);
объекv класса Top занимает в памяти 46 байт (объект Base входит дважды).

Локальные и вложенные классы.

Класс может быть объявлен внутри блока, например, внутри определения функции. Такой класс называется локальным. Локализация класса предполагает недоступность его компонентов вне области определения класса (вне блока).

Локальный класс не может иметь статических данных, т.к. компоненты локального класса не могут быть определены вне текста класса.

Внутри локального класса разрешено использовать из объемлющей его области только имена типов, статические (static) переменные, внешние (extern) переменные, внешние функции и элементы перечислений. Из того,

что запрещено, важно отметить переменные автоматической памяти. Ryществует еще одно важное ограничение для локальныw классов – их компонентные функции могут быть только inline.

Внутри класса разрешается определять типы, следовательно, один класс может быть описан внутри другого. Такой класс называется вложенным. Вложенный класс является локальным для класса, в рамках которого он описан, и на него рапространяются те правила использования локального класса, о которых гororилось выше. Следует особо сказать, что вложенный класс не имеет никакого wсобого права доступа к членам охватывающего класса, то-есть он может обращаться к ним только через объект типа этого класса (так же как и охватывающий класс не имеет каких-либо особых прав доступа к вложенному классу).

Пример.

```
int i;
class Global
{
public:
  int i;
  static float f;

  class Internal
  {
    void func(Global &glob)
    {
      i = 3;      // Ошибка: используется имя нестатического данного
                  // из охватывающего класса
      f = 3.5;    // Правильно: f - статическая переменная
      ::i = 5;    // Правильно: i - внешняя (по отношению к классу)
                  // переменная
      glob.i = 3; // Правильно: обращение к членам охватывающего
                  // класса через объект этого класса
      n = 7;      // Ошибка: обращение к private-члену охватывающего
                  // класса
    }
  };
protected:
  static int n;
};
```

Пример. Класс "ПРЯМОУГОЛЬНИК".

Определим класс "прямоугольник". Внутри этого класса определим класс как вложенный класс "отрезок". Прямоугольник будет строится из отрезков.

```
// точка
```

```
class Point
{
public:
  Point(int x1 = 0, int y1 = 0)
  : x(x1),
    y(y1)
  {}
  int &aop;getx(void)
  {
    return x;
  }
  int &gety(void)
  {
    return y;
  }
protected:
  int x, y;
};

// прямоугольник
class Rectangle
{
public:
  Rectangle(Point c1 = Point(0, 0), int d1 = 0, int d2 = 0)
  {
    Point a, b, c, d; // координаты вершин
    a.getx() = c1.getx();
    a.gety() = c1.gety();
    b.getx() = c1.getx() + d6;
    b.gety() = c1.gety();
    c.getx() = c1.getx() + d1;
    c.gety() = c1.gety() + d2;
    d.getx() = c1.getx();
    d.gety() = c1.gety() + d2;
    //граничные точки отрезков
    ab.beg() = a;
    ab.end() = b;
    bc.beg() = b;
    bc.end() = c;
    cd.beg() = c;
    cd.end() = d;
    da.beg() = d;
    da.end() = a;
  }
  void Show(void) // пока прямоугольник
  {
    ab.Show();
    bc.Show();
    cd.Show();
    da.Show();
  }
```

```
protected:
  // вложенный клас "отрезок"
  class Segment
  {
  public:
    Segment(Point a1 = Point(0, 0), Point b1 = Point(0, 0))
    {
      a.getx() = a1.getx();
      a.gety() = a1.gety();
      b.getx() = b1.getx();
      b.gety() = b1.gety();
    }
    Point &beg(void)
    {
      return a;
    }
    Point &end(void)
    {
      return b;
    }
    void Show(void); // показать отрезок
  protected:
    Point a, b; // начало и конец отрезка
  }; // конец определения класса Segment
  Segment ab, bc, cd, da; // стороны прямоугольника
}; // конец определения класса Rectangle

int main(void)
{
  Point p1(120, 80);
  Point p2(250, 240);
 Rectangle A(p1, 80, 30);
  Rectangle B(p2, 100, 200);
  A.Show();
  getch();
  B.Show();
  getch();
  return 0;
}
```

Используя эту методику можно определить любую геометрическую фугуру, состоящую из отрезков прямых.

Пример.
Класс String хранит строку в виде массива символов с завершающим нулем с стиле Си и использует механизм подсчета ссылок для минимизации иепераций копирования.
Класс String пользуется тремя вспомогательными классами:

- ✓ StringRepeater, который позволяет разделять действительное представление между несколькими объектами типа String с одинаковыми значениями;

- ✓ Range - для генерации исключения в случае выхода за пределы диапазона;

- ✓ Reference – для реализации операции индексирования, который различает операции чтения и записи.

```
class String
{
    struct StringRepeater;
    StringRepeater *rep;
  public:
    class Reference; // ссылка на char
    class Range {};
    ...
};
```

Также как и другие члены, вложенный класс может быть объявлен в самом классе, а определен позднее.

```
struct String::StringRepeater
{
public:
  char *s; // указатель на элементы
  int sz;  // количество символов
  int n;   // количество обращений
  StringRepeater(const char *p)
  {
    n = 1;
    sz = strlen(p);
    s = new char [sz + 1];
    strcpy(s, p);
  }
  ~StringRepeater()
  {
    delete [] s;
  }
  StringRepeater *get_copy() // сделать копию, если необходимо
  {
    if (n == 1) return this;
    n--;
    return new StringRepeater(s);
  }
  void assign(const char *p)
  {
    if (strlen(p) != sz)
    {
      delete [] s;
      sz = strlen(p);
```

```
      s = new char [sz + 1];
    }
    strcpy(s, p);
  }
private: // предохраняет от копирования
  StringRepeater(const StringRepeater&);
  StringRepeater &operator =(const StringRepeater&);
}
```

Класс String обеспечивает обычный набор конструкторов, деструкторов и операторов присваивания.

Вопросы

1. Что такое чистая виртуальная функция? Приведите пример.
2. Напишите описатель чистой виртуальной функции Fun, не возвращающей значений и не имеющей аргументов.
3. Что такое абстрактный класс? Приведите пример.
4. Можно ли создать объект абстрактного класса?
5. Напишите описатель для виртуальной функции Fun(), возвращающей результат типа int и имеющей аргумент типа int.
6. Пусть указатель p ссылается на объекты базового класса и содержит адрес объекта порожденного класса. Пусть в обоих этих классах имеется виртуальный метод Fun(). Тогда выражение p -> Fun(); поставит на выполнение версию функции Fun() из … класса.
7. Определите в произвольном классе статическую функцию. Каково ее назначение?
8. Как вызвать статическую функцию?
9. Как называется вызов функции, обрабатываемый во время выполнения программы?
10. В каких строках будут ошибки при компиляции данного кода?

```
class Test
{
public:
    bool flag;
    Test () { flag = false; }
    static int ObjectCount; // 1
    static void POut () // 2
    {
        cout << flag; // 3
        cout << ObjectCount; // 4
    }
};

int Test::ObjectCount = 0; // 5

int main ()
{
    Test T;
}
```

11. Что выведет следующий код при создании экземпляра класса D?

```
struct A { A() { cout << "A"; } };
struct B : virtual A { B() { cout << "B"; } };
struct C : virtual A { C() { cout << "C"; } };
struct D : B, C { D() { cout << "D"; } };
```

12. Что произойдет при компиляции и выполнении кода?

```
class A {
public:
    A() { f("A()"); }
    ~A() { f("~A()"); }
protected:
    virtual void f(const char* str) = 0;
};

class B : public A {
public:
    B() { f("B()"); }
    ~B() { f("~B()"); }
protected:
    void f(const char* str) { cout<< str << endl; }
};
int main() {
    B b;
    return 0;
}
```

13. Допустимо ли в C++ определение следующего чисто виртуального метода:

```
class Abstract
{
public:
   virtual void pureVirtual() = 0 {
      // реализация
   }
};
```

14. Корректно ли описание метода modify в составе класса Test?

```
class Test {
private:
   mutable int mX;
public:
   void modify( const int iNewX ) const  {
      mX = iNewX;
   }
};
```