ГОСУДАРСТВЕННЫЙ КОМИТЕТ СВЯЗИ, ИНФОРМАТИЗАЦИИ И ТЕЛЕКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ РУз

ТАШКЕНТСКИЙ УНИВЕРСИТЕТ ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ

Факультет “Профессиональное образование”

КУРСОВАЯ РАБОТА

по предмету “Объектно-ориентированные языки программирования”

Выполнила: Студентка группы 252-10 КТИр

Розыходжаева Дилдора

Приняла: _____

Ташкент – 2013 г.

Аннотация

В данной курсовой работе по предмету «Объектно-ориентированные языки программирования» на тему «Отношения между классами» рассматривается актуальность и этапы разработки программ, отражающих иерархию классов. В качестве средства разработки используется объектно-ориентированный язык C++, дается краткое описание особенностей применения классов, использования в них конструкторов и деконструкторов, создания полей и методов. Отдельно описывается процесс наследования и разрабатывается программа, демонстрирующая его свойства.

Общий объем работы: 22 страницы

Рисунков: 1

Автор: Розыходжаева Дилдора Аброровна, гр. 252-10

Оглавление

Введение

Цель объектно-ориентированного программирования состоит в повторном использовании созданных классов, что экономит время и силы программиста. Если уже создан некоторый класс, то возможны ситуации, что новому классу нужны многие или даже все особенности уже существующего класса, и необходимо добавить один или несколько элементов данных или функций. В таких случаях C++ позволяет строить новый объект, используя характеристики уже существующего объекта. Другими словами, новый объект будет наследовать элементы существующего класса (называемого базовым классом).

Наследование является фундаментальной концепцией объектно-ориентированного программирования. Если программы используют наследование, то для порождения нового класса необходим базовый класс, т.е. новый класс наследует элементы базового класса. Для инициализации элементов производного класса программа должна вызвать конструкторы базового и производного классов. Используя оператор точку или ->, программы могут легко обращаться к элементам базового и производного классов.

Для разрешения конфликта имен между элементами базового и производного классов программа может использовать оператор глобального разрешения, указывая перед ним имя базового или производного класса.

Цель данной курсовой работы – рассмотреть особенности объектно-ориентированного программирования и изучить принципы организации наследования в c++.

Задачами курсовой работы являются: получить практические навыки программирования по теме «Отношения между классами» и решить задачу. Реализовать путем наследования методы для автоматизации поиска указанной информации и указанных операций:

ШКОЛА (название, ПРЕПОДАВАТЕЛИ, КЛАССЫ, УЧЕНИКИ)

Все данные относительно ученика.

Классы и объекты в C++

Классы и объекты в C++ являются основными концепциями объектно-ориентированного программирования ООП.

Объектно-ориентированное программирование (ООП) — расширение структурного программирования, в котором основными концепциями являются понятия классов и объектов. Основное отличие языка программирования C++ от C состоит в том, что в C нету классов, а следовательно язык C не поддерживает ООП, в отличие от C++.

Классы в C++ можно сравнить с велосипедом. Велосипед — это объект, который был построен согласно чертежам. Так вот, эти самые чертежи играют роль классов в ООП. Таким образом классы — это некоторые описания, схемы, чертежи по которым создаются объекты. Теперь ясно, что для создания объекта в ООП необходимо сначала составить чертежи, то есть классы. Классы имеют свои функции, которые называются методами класса. Передвижение велосипеда осуществляется за счёт вращения педалей, если рассматривать велосипед с точки зрения ООП, то механизм вращения педалей — это метод класса. Каждый велосипед имеет свой цвет, вес, различные составляющие — всё это свойства. Причём у каждого созданного объекта свойства могут различаться. Имея один класс, можно создать неограниченно количество объектов (велосипедов), каждый из которых будет обладать одинаковым набором методов, при этом можно не задумываться о внутренней реализации механизма вращения педалей, колёс, срабатывания системы торможения, так как всё это уже будет определено в классе. Разобравшись с назначением класса, дадим ему грамотное определение.

Классы в C++ - это абстракция описывающая методы, свойства, ещё не существующих объектов. **Объекты** - конкретное представление абстракции, имеющее свои свойства и методы. Созданные объекты на основе одного класса называются экземплярами этого класса. Эти объекты могут иметь различное

поведение, свойства, но все равно будут являться объектами одного класса. В ООП существует три основных принципа построения классов:

1. **Инкапсуляция** - это свойство, позволяющее объединить в классе и данные, и методы, работающие с ними и скрыть детали реализации от пользователя.

2. **Наследование** - это свойство, позволяющее создать новый класс-потомок на основе уже существующего, при этом все характеристики класса родителя присваиваются классу-потомку.

3. **Полиморфизм** - свойство классов, позволяющее использовать объекты классов с одинаковым интерфейсом без информации о типе и внутренней структуре объекта.

Рассмотрим структуру объявления классов.

```
1   // объявление классов в C++
2   class /*имя класса*/
3   {
4     private:
5     /* список свойств и методов для использования внутри класса */
6     public:
7     /* список методов доступных другим функциям и объектам программы */
8     protected:
9     /*список средств, доступных при наследовании*/
10  };
```

Объявление класса начинается с зарезервированного ключевого слова **class**, после которого пишется имя класса. В фигурных скобочках, **строки 3 — 10** объявляется тело класса, причём после закрывающейся скобочки обязательно нужно ставить точку с запятой, **строка 10**. В теле класса объявляются три метки спецификации доступа, **строки 4, 6, 8,** после каждой метки нужно обязательно ставить двоеточие. В **строке 4** объявлена метка спецификатора доступа private. Все методы и свойства

класса, объявленные после спецификатор доступа private будут доступны только внутри класса. В строке 6 объявлен спецификатор доступа public, все методы и свойства класса, объявленные после спецификатора доступа public будут доступны другим функциям и объектам в программе. Пока на этом остановимся, спецификатор доступа protected разбирать сейчас не будем, просто запомните, что он есть. П

Конструкторы и деконструкторы

Каждый класс имеет специальные функции, которые называются конструктор и деструктор. Конструктор класса вызывается всякий раз, когда объект создается в памяти ЭВМ и служит обычно для инициализации данных класса. Конструктор имеет то же имя, что и имя класса. Деструктор вызывается при удалении класса из памяти и используется, как правило, для освобождения ранее выделенной памяти под какие-либо данные этого класса. Имя деструктора совпадает с именем класса, но перед ним ставится символ '~'. Рассмотрим пример реализации конструктора и деструктора для класса CPos.

```
class CPos
{
public:
CPos() {printf("Вызов конструктора.\n");}
~CPos() {printf("Вызов деструктора.\n");}

int sp_x, sp_y; //координата начала
int ep_x, ep_y; //координата конца
};
```

Здесь ключевое слово public используется для обеспечения общего доступа к функциям и переменным класса.

Для создания нового экземпляра класса в памяти ЭВМ используется оператор new языка C++, а для удаления – оператор delete. Использование

данных операторов для создания экземпляра класса CPos и его удаления выглядит следующим образом:

```
CPos *pos_ptr = new CPos(); //создание объекта
delete pos_ptr; //удаление объекта
```

В результате выполнения этих двух строк программы на экране появятся сообщения:

Вызов конструктора.

Вызов деструктора.

Экземпляр класса также можно создать подобно обычным переменным без использования указателей, как показано ниже

```
CPos pos;
```

В этом случае переменная pos называется представителем класса, у которого также вызывается конструктор при его создании и деструктор при его удалении из памяти.

Следует отметить, что при создании нового экземпляра класса можно выполнять инициализацию различных переменных путем передачи их значений через конструктор. В этом случае конструктор должен быть объявлен с набором необходимых аргументов, например, так:

```
class CPos
{
public:
CPos(int x1, int y1, int x2,int y2)
{
sp_x = x1; sp_y = y1;
ep_x = x2; ep_y = y2;
}
~CPos() {}
```

```
int sp_x, sp_y;
int ep_x, ep_y;
};
```

и процесс создания экземпляра класса принимает вид:

```
CPos *pos_ptr = new CPos(10,10,20,20);
```

или

```
CPos pos(10,10,20,20);
```

Такой способ описания и вызова конструктора представляет дополнительное удобство инициализации данных при создании нового объекта. При этом конструктор, как и любую функцию, можно перегружать. Это значит, что можно задать несколько типов конструкторов (с разным набором входных параметров) в одном и том же классе. Например, если создается экземпляр класса графического примитива, но для него неизвестны начальные и конечные координаты, то целесообразно вызвать конструктор CPos() без аргументов, а если координаты известны, то выполнить их инициализацию путем вызова конструктора с аргументами. Для описания нескольких типов конструкторов в одном классе достаточно дать их определения в нем:

```
class CPos
{
public:
CPos() {}
CPos(int x1, int y1, int x2,int y2)
{
sp_x = x1; sp_y = y1;
ep_x = x2; ep_y = y2;
}
~CPos() {}
```

```
int sp_x, sp_y;
int ep_x, ep_y;
};
```

В классах помимо переменных, конструкторов и деструкторов можно задавать описания и обычных функций, которые, в этом случае, называются методами.

Основы наследования

Цель объектно-ориентированного программирования состоит в повторном использовании созданных классов, что экономит время и силы. Если уже создан некоторый класс, то возможны ситуации, что новому классу нужны многие или даже все особенности уже существующего класса, и необходимо добавить один или несколько элементов данных или функций. В таких случаях C++ позволяет вам строить новый объект, используя характеристики уже существующего объекта. Другими словами, новый объект будет наследовать элементы существующего класса (называемого базовым классом). Когда строится новый класс из существующего, этот новый класс часто называется производным классом.

Простое наследование

Наследование представляет собой способность производного класса наследовать характеристики существующего базового класса. Например, предположим, что у вас есть базовый класс employee:

```
class employee
{
public:
  employee(char *, char *, float);
  void show_employee(void);
private:
  char name[64];
```

```
  char position[64];
  float salary;
};
```

Далее предположим, что программе требуется класс manager, который добавляет следующие элементы данных в класс employee:

```
float annual_bonus;
char company_car[64];
int stock_options;
```

В данном случае программа может выбрать два варианта: во-первых, программа может создать новый класс manager, который дублирует многие элементы класса employee, или программа может породить класс типа manager из базового класса employee. Порождая класс manager из существующего класса employee, вы снижаете объем требуемого программирования и исключаете дублирование кода внутри вашей программы.

Для определения этого нужно указать ключевое слово class, имя manager, следующее за ним двоеточие и имя employee, как показано ниже:

Производный класс *//----->* class manager : public employee { *<-------//* ***Базовый класс***

```
// Здесь определяются элементы
};
```

Ключевое слово public, которое предваряет имя класса employee, указывает, что общие (public) элементы класса employee также являются общими и в классе manager. Например, следующие операторы порождают класс manager.

```
class manager : public employee
{
public:
  manager(char *, char *, char *, float, float, int);
```

```
  void show_manager(void);
private:
  float annual_bonus;
  char company_car[64];
  int stock_options;
};
```

Когда порождается класс из базового класса, частные элементы базового класса доступны производному классу только через интерфейсные функции базового класса. Таким образом, производный класс не может напрямую обратиться к частным элементам базового класса, используя оператор точку.

Наследование представляет собой способность производного класса наследовать характеристики существующего базового класса. Простыми словами это означает, что, если есть класс, чьи элементы данных или функции-элементы могут быть использованы новым классом, можно построить этот новый класс в терминах существующего (или базового) класса. Новый класс в свою очередь будет наследовать элементы (характеристики) существующего класса. Использование наследования для построения новых классов сэкономит значительное время и силы на программирование. Объектно-ориентированное программирование широко использует наследование, позволяя вашей программе строить сложные объекты из небольших легко управляемых объектов.

Предположим, например, что используется следующий базовый класс book внутри существующей программы:

```
      class book
      {
public:
  book (char *, char *, int);
  void show_book(void);
```

```
private:
  char title[64];
  char author[б 4];
  int pages;
};
```

Далее предположим, что программе требуется создать класс library_card, который будет добавлять следующие элементы данных в класс book:

```
char catalog[64];
int checked_out; // 1, если проверена, иначе О
```

Программа может использовать наследование, чтобы породить класс library _card из класса book, как показано ниже:

```
class library_card : public book
{
public:
  library_card(char *, char *, int, char *, int);
  void show_card(void);
private:
  char catalog[64] ;
  int checked_out;
};
```

Защищенные элементы

При изучении определений базовых классов можно встретить элементы, объявленные как public, private и protected (общие, частные и защищенные). Производный класс может обращаться к общим элементам базового класса, как будто они определены в производном классе. С другой стороны, производный класс не может обращаться к частным элементам базового класса напрямую. Вместо этого для обращения к таким элементам производный класс должен использовать интерфейсные функции. Защищенные элементы базового класса

занимают промежуточное положение между частными и общими. Если элемент является защищенным, объекты производного класса могут обращаться к нему, как будто он является общим. Для оставшейся части программы защищенные элементы являются как бы частными. Единственный способ, с помощью которого программы могут обращаться к защищенным элементам, состоит в использовании интерфейсных функций. Следующее определение класса book использует метку protected, чтобы позволить классам, производным от класса book, обращаться к элементам title, author и pages напрямую, используя оператор точку:

```
    class book
    {
public:
  book(char *, char *, int) ;
  void show_book(void) ;
protected:
  char title [64];
  char author[64];
  int pages;
};
```

Если предполагается, что через некоторое время придется породить новые классы из создаваемого сейчас класса, устанавливается, должны ли будущие производные классы напрямую обращаться к определенным элементам создаваемого класса, и объявите такие элементы защищенными, а не частными.

Разрешение конфликта имен

Если порождается один класс из другого, возможны ситуации, когда имя элемента класса в производном классе является таким же, как имя элемента в базовом классе. Если возник такой конфликт, C++ всегда использует элементы производного класса внутри функций производного класса. Например,

предположим, что классы book и library_card используют элемент price. В случае класса book элемент price соответствует продажной цене книги, например $22.95. В случае класса library'_card price может включать библиотечную скидку, например $18.50. Если в вашем исходном тексте не указано явно (с помощью оператора глобального разрешения), функции класса library_card будут использовать элементы производного класса {library_card). Если же функциям класса library_card необходимо обращаться к элементу price базового класса {book), они должны использовать имя класса book и оператор разрешения, например book::price. Предположим, что функции show_card необходимо вывести обе цены. Тогда она должна использовать следующие операторы:

```
cout << "Библиотечная цена: $" << price << endl;
cout << "Продажная цена: $" << book::price << endl;
```

Итак, исходя из вышесказанного можно сделать вывод:

1. Наследование представляет собой способность производить новый класс из существующего базового класса.
2. Производный класс — это новый класс, а базовый класс — существующий класс.
3. Когда порождается один класс из другого (базового класса), производный класс наследует элементы базового класса.
4. Для порождения класса из базового нужно начать определение производного класса ключевым словом class, за которым следует имя класса, двоеточие и имя базового класса, например class dalmatian: dog.
5. Когда порождается класс из базового класса, производный класс может обращаться к общим элементам базового класса, как будто эти элементы определены внутри самого производного класса. Для доступа к частным данным базового класса производный класс должен использовать интерфейсные функции базового класса.

6. Внутри конструктора производного класса ваша программа должна вызвать конструктор базового класса, указывая двоеточие, имя конструктора базового класса и соответствующие параметры сразу же после заголовка конструктора производного класса.

7. Чтобы обеспечить производным классам прямой доступ к определенным элементам базового класса, в то же время защищая эти элементы от оставшейся части программы, C++ обеспечивает защищенные {protected) элементы класса. Производный класс может обращаться к защищенным элементам базового класса, как будто они являются общими. Однако для оставшейся части программы защищенные элементы эквивалентны частным.

8. Если в производном и базовом классе есть элементы с одинаковым именем, то внутри функций производного класса C++ будет использовать элементы производного класса. Если функциям производного класса необходимо обратиться к элементу базового класса, вы должны использовать оператор глобального разрешения, например base class:: member.

Реализация наследования (иерархия классов)

Постановка задачи

Определить иерархию классов. Реализовать путем наследования методы для автоматизации поиска указанной информации и указанных операций.

ШКОЛА (название, ПРЕПОДАВАТЕЛИ, КЛАССЫ, УЧЕНИКИ)

Все данные относительно ученика

Решение задачи

Для предметной области определяем общие (родительские) и частные (дочерние) элементы.

В городе может быть множество различных школ, однако все объединены общим признаком: имеются классы, их классные руководители и ученики, составляющие основу этого класса. Следовательно, эти элементы составляют базовый (родительский) класс:

```
class klasses_info
{
public:
        klasses_info (string, string, string); //конструктор класса
        void show_klasses_info (void); //метод класса, выводящий данные на экран
public:
        string teacher;
        string klass;
        string pupil_name;
};
```

Инициализируем конструктор родительского класса:

```
klasses_info::klasses_info(string teacher, string klass, string pupil_name)
{
        this-> teacher=teacher;
        this-> klass=klass;
        this->pupil_name = pupil_name;
}
```

Инициализируем метод для печати объектов класса:

```
void klasses_info::show_klasses_info(void)
{
        cout<<"Teacher: "<< teacher << endl;
        cout<<"Klass: "<< klass << endl;
        cout<<"Pupil_name: "<< pupil_name << endl;
}
```

Далее, определяем производный класс. Это элементы, которые в целом различны для заданной предметной области. Соответственно, их составляют школы:

```
class school_name :public klasses_info
{
```

```
public:
        school_name(string , string , string, string); // конструктор
        void show_card(void); // метод для вывода на экран
private:
        string school;
};
```

Инициализируем конструктор производного класса:

```
school_name::school_name(string teacher, string klass, string pupil_name, string school):
klasses_info (teacher, klass, pupil_name)
{
        this->school=school;
}
```

Как видно из кода, конструктор дочернего класса сначала вызывает конструктор родительского класса и передает ему элементы, а затем инициализирует элемент своего класса.

Инициализируем метод печати данных класса:

```
void school_name::show_card(void)
{
        show_klasses_info();
        cout<<"School: "<< school << endl;
        cout<<"\n----------\nCreated by Rozikhodjaeva D. "<< endl;
}
```

Внутри метода сначала вызывается метод базового класса, где печатаются его элементы, а затем уже выводятся элементы дочернего класса и дополнительная информация.

После того как основные классы созданы, необходимо разработать класс, который обрабатывает данные описанных классов. В этот класс будет передаваться число объектов (элементов массива), сам объект (массив элементов) и условие отбора.

```
class filtr
{
public:
        static void filtr_pupil (int n, school_name *card[], string name);
};
```

Метод данного класса имеет вид:

```
        void filtr::filtr_pupil(int n, school_name *card[], string name)
{
        for(int i=0;i<n;i++)
                if(card[i]->pupil_name==name)
                card[i]->show_card();
        cout<<"\n Pupil with such name not find ";

}
```

Суть метода: цикл проходит по всем объектам класса school_name и ищет среди элементов pupil_name то, которое совпадает с введенной пользователем строкой name. Если совпадение обнаружено, вызывается метод печати для данного объекта. Если же к окончанию цикла совпадения не обнаружены, программа выводит соответствующее оповещение.

В главной программе main инициализируются объекты класса и вызывается метод поиска ученика по имени (filtr_pupil)

Код программы:

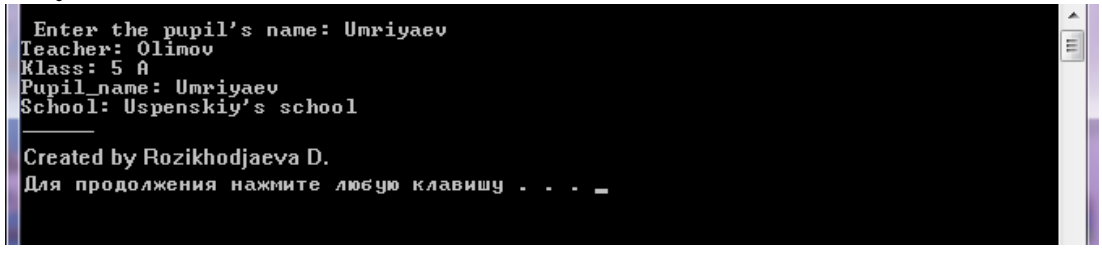
```
#include<iostream>
#include <string>

using namespace std;

class klasses_info
{
public:
	klasses_info (string, string, string);
	void show_klasses_info (void);
public:
	string teacher;
	string klass;
	string pupil_name;
};
klasses_info::klasses_info(string teacher, string klass, string pupil_name)
{
	this-> teacher=teacher;
	this-> klass=klass;
	this->pupil_name = pupil_name;
}
void klasses_info::show_klasses_info(void)
{
	cout<<"Teacher: "<< teacher << endl;
	cout<<"Klass: "<< klass << endl;
	cout<<"Pupil_name: "<< pupil_name << endl;
}
class school_name :public klasses_info
{
public:
	school_name(string , string , string, string);
	void show_card(void);
private:
	string school;
};
```

```
school_name::school_name(string teacher, string klass, string pupil_name,string school):
klasses_info (teacher, klass, pupil_name)
{
	this->school=school;
}
void school_name::show_card(void)
{
	show_klasses_info();
	cout<<"School: "<< school << endl;
	cout<<"\n----------\nCreated by Rozikhodjaeva D. "<< endl;
}
class filtr
{
public:
	static void filtr_pupil (int n, school_name *card[], string name);
};
	void filtr::filtr_pupil(int n, school_name *card[], string name)
{
	for(int i=0;i<n;i++)
		if(card[i]->pupil_name==name)
		card[i]->show_card();
}
int main()
{
	school_name *card[6];
	card[0]=new school_name("Petrova", "8 D", "Vahabova",  "55 middle school");
	card[1]=new school_name("Olimov", "5 A", "Zakirova",  "Uspenskiy's school");
	card[2]=new school_name("Olimov", "5 A", "Umriyaev",  "Uspenskiy's school");
	card[3]=new school_name("Olimov", "5 A", "Sobirov",  "Uspenskiy's school");
	card[4]=new school_name("Bakaeva", "7 B", "Turaeva",  "60 German school");
	card[5]=new school_name("Halikova", "9 G", "Bikova",  "49 middle school");
	string h;
	cout<<"\n Enter the pupil's name: ";
	cin>>h;
	filtr::filtr_pupil(6, card, h);
	return 0;
}
```

Результат:



Заключение

Наследование представляет собой способность производить новый класс из существующего базового класса. Наследование в C++ позволяет строить /порождать) новый класс из существующего класса. Строя таким способом один класс из другого, можно уменьшить объем программирования, что, в свою очередь, экономит время. C++ позволяет порождать класс из двух или нескольких базовых классов. Когда мы порождаем класс из базового класса, производный класс может обращаться к общим элементам базового класса, как будто эти элементы определены внутри самого производного класса. Для доступа к частным данным базового класса производный класс должен использовать интерфейсные функции базового класса.

В данной курсовой работе были рассмотрены особенности объектно-ориентированного программирования и изучены принципы организации наследования в c++.

Полученные знания помогли получить практические навыки программирования по теме «Отношения между классами» и реализовать путем наследования методы для автоматизации поиска информации. Данные навыки будут полезны в дальнейшем при разработке качественных программ, работающих с базами данных или на их принципе.

Список литературы

1. Т. Сван. Освоение Borland C++ 4.5: Пер. с англ. - Киев: Диалектика, 1996.
2. Г. Шилдт. Самоучитель C++: Пер. с англ. - Санкт-Петербург: BHV-Санкт-Петербург, 1998. 620с.
3. У. Сэвитч. C++ в примерах: Пер. с англ. - Москва: ЭКОМ, 1997. 736с.
4. К. Джамса. Учимся программировать на языке C++: Пер. с англ. - Москва: Мир, 1997. 320с.
5. В.А. Скляров. Язык C++ и объектно-ориентированное программирование: Справочное издание. - Минск: Вышэйшая школа, 1997. 480с.
6. Х. Дейтел, П. Дейтел. Как программировать на C++: Пер. с англ. - Москва: ЗАО "Издательство БИНОМ", 1998. 1024с.
7. Подбельский В. В. Глава 10.2 Множественное наследование и виртуальные базовые классы // Язык Си++ / М.: Финансы и статистика, 2003
8. Wikipedia.org
9. www.allbest.ru
10. www.rogrammer.com