Лекции по C/C++: Классы, часть 2 (наследование)

Производный (derived) класс — это *расширение* существующего базового класса или *класс-потомок*. Он может модифицировать права доступа к данным родительского класса, добавить новые свойства или переопределить (перегрузить) существующие у родителя методы. Описание потомка выглядит следующим образом:

```
class ИмяПроизводногоКласса : БазовыйСписок {
 СписокЧленов;
};
```

Здесь `БазовыйСписок` содержит перечень разделенных запятой спецификаторов атрибутов доступа (`public`, `protected` или `private`) и имен базовых классов:

```
class Child : public Papa, public Mama {
 //...
};
```

Производный класс наследует все члены перечисленных базовых классов, но может использовать только члены с атрибутом `public` или `protected`.

Класс обычно наследуется как `public` или как `private`. При этом модификатор `private` трансформирует компоненты базового класса с атрибутами доступа `public` и `protected` в компоненты `private` производного класса, в то время как `private`-компоненты становятся недоступны в производном классе.

Модификатор наследования `public` не изменяет уровня доступа. Производный класс наследует все компоненты своего базового класса, но может непосредственно использовать только те из них, которые определены с атрибутами `public` и `protected`. Подробнее особенности наследования рассмотрены ниже в примере 6.

Пример 1. Для дальнейшей работы используем класс `Student` из предыдущей лекции и вынесем его описание в заголовочный файл `student.h`

В реальных проектах так делают *всегда* и *всегда* интерфейс класса (свойства и прототипы методов) описан в файле `имякласса.h`, а реализация (тела функций-методов) - в файле `имякласса.cpp`

Теперь представим, что у класса "Студент" есть класс-потомок "Студент, имеющий хобби", а само дополнительное свойство "хобби" представляет собой односимвольный идентификатор типа `char`. Назовём класс-потомок `Hobbit` и включим его описание в файл `student.h`, который примет следующий вид:

```
#ifndef STUDENTH
#define STUDENTH

class Student {
 private:
```

```
  char *Name;
  int Group;
 public:
  Student (void);
  Student (char *,int);
  Student (Student &);
  ~Student ();
  void setName (char *);
  char *getName (void);
  void setGroup (int);
  int getGroup (void);
  void showStudent (void);
};

class Hobbit : public Student {
 protected:
  char Hobby;
 public:
  Hobbit(): Student() { Hobby='1'; };
  Hobbit (Hobbit &From): Student(From) { setHobby(From.Hobby); }
  Hobbit(char *Name,int Group=0): Student(Name,Group) { Hobby='2'; };
  Hobbit(char *Name,int Group,char Hobby): Student(Name,Group) {
   this->Hobby=Hobby;
  };
  void setHobby (char Hobby) { this->Hobby = Hobby; }
  char getHobby () { return this->Hobby; }
  void showStudent ();
};
#endif
```

Директива `#ifndef STUDENTH` страхует от повторного включения содержимого файла в проект.

Итак, в `student.h` здесь добавлен прототип класса-потомка `Hobbit` для класса-родителя `Student`.

Обратите внимание, что конструкторы класса-потомка используют соответствующие конструкторы родителя, вызывая их через список инициализации:

```
Hobbit(char *Name,int Group=0): Student(Name,Group) { Hobby='2'; };
```

Дополнительные методы `setHobby`, `getHobby` будут работать с добавленным свойством `Hobby`. Недостающий перегруженный метод `showStudent` будет написан чуть позже.

Добавим в проект файл исходного кода `student.cpp` и внесём туда код, написанный для базового класса:

```
#include <cstdlib>
#include <cstring>
#include <iostream>
#include "student.h"
```

```
using namespace std;

Student::Student (void) { Name = NULL; Group = 0; } //1

Student::Student (char *newName, int newGroup=0) { //2
 setGroup (newGroup);
 int n=strlen(newName);
 if (n>0) {
  Name = new char [n+1];
  if (Name!=NULL) strcpy(Name,newName);
 }
}

Student::Student (Student &From) { //3
 setGroup (From.Group);
 if (From.Name==NULL) { Name=NULL; return; }
 int n=strlen(From.Name);
 if (n>0) {
  Name = new char [n+1];
  if (Name) strcpy (Name,From.Name);
 }
}

Student::~Student () { if (Name) delete[] Name; }

void Student::setName (char *newName) { //4
 int n=strlen(newName);
 if (n>0) {
  if (Name) delete[] Name;
  Name = new char [n+1];
  if (Name) strcpy(Name,newName);
 }
}

void Student::setGroup (int g) { Group=g; }
char * Student::getName () { return Name; }
int Student::getGroup () { return Group; }

void Student::showStudent (void) { //5
 cout << endl << (Name==NULL?"NULL":Name) << ", " << Group;
}
```

Комментарии к коду:

1. Конструктор по умолчанию инициализирует свойства базового класса пустыми значениями.

2. Конструктор с параметрами выделяет память для свойства `Name` и копирует туда строку, переданную в конструктор параметром `newName`.

3. Конструктор копирования проверяет, не пусто ли свойство `Name` у объекта справа от знака "=", если это так, свойство `Name` текущего объекта ставится в `NULL`. Это должно обеспечить корректное копирование и "пустых" объектов, например

```
Student *s1 = new Student();
Student s3 = *s1;
```

Кроме того, конструктор копирования не должен просто вызывать `setName` для установки свойства `Name` – ведь у объекта слева от знака "=" свойство `Name` может быть ещё не определено и содержать "мусор", не равный, в том числе, и значению `NULL`.

4. Метод `setName` всегда вызывается для уже "сконструированного" объекта, поэтому он может освободить память, занятую текущим именем `Name`.

5. Метод вывода информации учитывает, что может потребоваться вывод объекта, у которого не установлено свойство `Name`.

> **Замечание**: как правило, на практике в проект добавляется объект "Класс C++", *сразу же* содержащий файлы `.cpp` и `.h` и эта пара файлов содержит описание *одного* класса.

Непосредственно после этого кода в файле `student.cpp` напишем реализацию недостающего метода класса-потомка, а также продемонстрируем функцию, не являющуюся членом класса, но создающую его объект:

```
void Hobbit::showStudent() {
 this->Student::showStudent(); //сначала вызываем метод родителя!
 cout << ", " << Hobby; //затем "дописываем" информацию потомка
}

Hobbit * Construct (Hobbit &From) {
 //Это пример метода, не являющегося членом класса,
 //но возвращающего указатель на его новый экземпляр
 Hobbit *NewHobbit = new
  Hobbit(From.getName(),From.getGroup(),From.getHobby());
 return NewHobbit;
}
```

Код главного файла проекта, например, с именем `main.cpp`:

```
#include <iostream>
#include "student.h"
using namespace std;

int main() {
 Hobbit Vasya("Vasya",210);
 Vasya.showStudent();
 Hobbit Petya("Petya");
 void (Hobbit:: *pointer)(void) = &Hobbit::showStudent;
  //Присвоили адрес функции класса указателю на функцию
 (Petya.*pointer)();
  //Теперь можем вызвать метод класса "косвенно", а значит,
```

```
  //переопределить вызов динамически, если это нужно!
 extern Hobbit * Construct (Hobbit &);
  //прототип внешнего метода Construct, описанного в другом файле проекта
 Hobbit *Grisha = Construct (Petya);
 Grisha->showStudent();
 Hobbit *Tema = new Hobbit ("Tema",201,'3');
 Tema->setHobby ('4');
 (Tema->*pointer)(); //Или Tema->showStudent();
 Hobbit Anastas=Vasya;
  //Или =*Tema, тут вызван конструктор копирования!
 Anastas.showStudent();

 cin.sync(); cin.get(); return 0;
}
```

Друзья класса. Функции, не являющиеся членами класса, но объявленные его "друзьями" с помощью ключевого слова `friend`, имеют полный доступ к данным класса, даже если эти данные приватны.

Пример 2. Функция-друг класса.

```
#include <iostream>
using namespace std;

class A {
 private: int n;
 public: A(int n=0) { this->n = n; }
 friend void show (A &);
};

void show(A &a) { cout << endl << a.n; }

int main () {
 A a(1);
 show(a); //работает, хотя n - приватный член класса!
 system("pause>nul"); return 0;
}
```

В нашем проекте для объявления другом класса `Hobbit` функции `Construct` также было бы достаточно добавить её прототип с ключевым словом `friend` в описание класса `Hobbit`:

```
class Hobbit : public Student {
  //...
  friend Hobbit * Construct (Hobbit &From);
};
```

Чрезмерное использование "друзей" не рекомендуется, так как не соответствует требованиям к безопасности кода.

Виртуальные функции. Для написания иерархии классов часто применяются виртуальные методы. Такие функции базового класса, объявленные с ключевым

словом `virtual`, специально предназначены для реализации в классах-потомках. Например, метод `draw` ("нарисовать") у базового класса "геометрическая фигура" разумно сделать виртуальным, чтобы каждый из классов-наследников "прямоугольник" и "окружность" мог реализовать свой собственный метод отрисовки. Но все методы будут вызываться одним ключевым словом, что удобно. Кроме того, в ряде случаев следует объявлять виртуальными явные деструкторы (см. ниже).

Если функцию объявить виртуальной и перегрузить её в классе-потомке, то при её вызове через указатель на базовый класс будет вызываться именно та версия функции, которая определена в классе-потомке. Иначе была бы вызвана та версия, которая определена в базовом классе.

Встраиваемые функции. Объявляются в описании класса с ключевым словом `inline`. Встретив такое описание, компилятор может встроить тело функции непосредственно в точку вызова. Таким образом, объём кода увеличивается, но за счёт отказа от вызова функций и передачи параметров через стек, код может работать быстрее. Чаще всего встраивание применяется для небольших по размеру функций. Современные компиляторы могут игнорировать указание директивы `inline` в случаях, когда встраивание противоречит текущим правилам оптимизации кода.

Кроме того, функцию с модификатором `inline` можно помещать в файл `*.h`. Даже если функция не будет встроена, её множественное определение в разных модулях компиляции не должно вызывать проблем при сборке приложения.

Пример 3. Встраиваемые и виртуальные методы. Пример также показывают "упрощённую" работу со строковыми полями – память под свойство `name` всегда выделяется фиксированного размера `MAXSIZE`.

```
#include <iostream>
#include <locale.h>
using namespace std;
#define MAXSIZE 30

class Papa {
protected:
 char *name; //имя
public:
 Papa (char *name=NULL) {
  if (name) {
   this->name = new char [MAXSIZE+1];
   strncpy(this->name,name,MAXSIZE);
   this->name[strlen(name)]='\0';
  }
 }
 virtual ~Papa() {
  cout << "~Papa() ";
   //просто для иллюстрации, что вызывался деструктор
  if (this->name) delete[] this->name;
 }
 inline char *getname () { return this->name; }
 inline virtual void show () { cout << endl << this->name; }
```

```
};

class Sinok : public Papa {
private:
 char *midname;
public:
 Sinok (char *name, Papa *p) : Papa (name) {
  this->midname = new char [strlen(p->getname())+1];
   //отчество ставим в папино имя
  strcpy(this->midname,p->getname());
 }
 ~Sinok() {
  cout << "~Sinok() ";
  if (this->midname) delete[] this->midname;
 }
 inline void show () {
  cout << endl << this->name << " " << this->midname;
 }
};

int main(void){
 setlocale (LC_ALL,"Russian");
 Papa *p = new Papa ("Fedya"); p->show();
 Sinok *s = new Sinok ("Vasya",p); s->show();
 delete s; cout << endl;
  //Проверим, что были вызваны оба деструктора для потомка
 delete p; //Вызов деструктора предка
 system("pause>nul"); return 0;
}
```

Если из класса `Sinok` исключить метод `show`, при вызове `s->show();` будет вызываться метод родителя, печатающий только имя.

При создании иерархии классов *деструктор базового класса должен быть всегда виртуальным*.

Пример 4. Виртуальный деструктор базового класса

```
#include <iostream>
using namespace std;

class A {
 public:
  A() { cout << "A()"; }
  ~A() { cout << "~A()"; }
};

class B : public A {
 public:
  B() { cout << "B()"; }
```

```
  ~B() { cout << "~B()"; }
};

int main () {
 A *a = new B;
 delete a;
 system ("pause");
 return 0;
}
```

Эта программа напечатает: `A()B()~A()`

То есть, деструктор класса `B` не вызывался, соответственно, он мог не освободить память, если таковая выделялась конструктором!

Изменив деструктор класса `A` на

```
virtual ~A() { cout << "~A()"; }
```

получим вывод `A()B()~B()~A()`

Таким образом, виртуальный деструктор нужен для правильного полиморфного удаления объектов. Во избежание проблем всегда следует как минимум:

- При наличии хотя бы одного виртуального метода объявлять виртуальным и деструктор;
- Также его надо явно объявлять виртуальным, если класс предполагается в будущем сделать базовым.

Ниже приводятся дополнительные примеры для закрепления и углубления материала.

Пример 5. Класс-окружность и его наследник - цилиндр.

```
#include <iostream>
#include <locale.h>
using namespace std;
const double pi = 3.1415926;

class Circle {
protected:
 double r; //радиус
public:
 Circle (double rVal =0) : r(rVal) {}
 void setRadius(double rVal) { r = rVal; }
 double getRadius () { return r; }
 double Area() { return pi*r*r; }
 void showData() ;
};

class Cylinder : public Circle {
protected:
 double h; //высота
public:
 Cylinder(double hVal = 0, double rVal = 0):h(hVal),Circle(rVal) {}
 void setHeight(double hVal) { h = hVal; }
```

```
 double getHeight() { return h; }
 double Area() { return 2*Circle::Area()+2*pi*r*h; }
 void showData() ;
};

void Circle::showData() {
 cout << "Радиус окружности = " << getRadius() << endl
      << "Площадь круга = " << Area() << endl << endl;
}

void Cylinder::showData() {
 cout << "Радиус основания = " << getRadius() << endl
      << "Высота цилиндра = " << getHeight() << endl
      << "Площадь поверхности = " << Area () << endl;
}

int main(void){
 setlocale (LC_ALL,"Russian");
 Circle circle(2);
 Cylinder cylinder(10, 1);
 circle.showData ();
 cylinder.showData();
 system("pause>nul"); return 0;
}
```

Пример 6. Иерархия классов и наследование с разными атрибутами доступа.

```
#include <iostream>
#include <stdlib.h>
using namespace std;

class Parent {
 int a;
 protected:
 int b;
 public:
 int c;
 Parent () { a=1; b=2; c=3; }
 inline virtual void show (void) {
  cout << endl << "Parent: " << a << "," << b << "," << c;
 }
};

class Child1 : protected Parent {
 //Поля Parent, бывшие private, остались private,
 //наследовались уровни доступа protected и public
 public:
 inline virtual void show (void) {
  cout << endl << "Child1: " << b << "," << c;
```

```
 }
};

class Child2 : private Parent {
 //Поля Parent, бывшие protected и public, стали private.
 //Этот класс не сможет "напрямую" использовать поле a,
 //а его потомки - поля b и c
 public:
 inline virtual void show (void) {
  cout << endl << "Child2: " << b << "," << c;
 }
};

class Child11 : public Child1 {
 //поле a было в Child1 приватным,
 //прямой доступ к нему больше невозможен
 public:
 inline void show (void) {
  cout << endl << "Child11: " << b << "," << c;
 }
};

class Child21 : public Child2 {
 //Этот класс не сможет "увидеть" ни a, ни b, ни c
 public:
 inline void show (void) {
  cout << endl << "Child21: no accessible fields!";
 }
};

int main () {
 Parent A; A.show ();
 Child1 B; B.show ();
 Child2 C; C.show ();
 Child11 B1; B1.show ();
 Child21 C1; C1.show ();
 system ("pause>nul");
 return 0;
}
```

Скачать проект "Студент" в архиве .zip, Studio 2010 и выше (5 Кб)

Оглавление всей серии лекций

теги: c++ учебное

Здесь можно оставить комментарий, обязательны только выделенные цветом поля.
Не пишите лишнего, и всё будет хорошо

Ваше имя:

Пароль (если желаете запомнить имя):

Любимый URL (если указываете, то вставьте полностью):

Текст сообщения (до 1024 символов):

Введите код сообщения: ~~3~~**9**83

Добавить Сброс

10.03.2016, 16:04; рейтинг: 6668