Лекции по C/C++: Классы, часть 1 (инкапсуляция)

С усложнением программ основной идеей обеспечения их управляемости стало *моделирование*. Реальный мир состоит из взаимодействующих между собой объектов. С точки зрения программирования объекты, характеризуемые разнородными данными, можно описать как раз *структурами*, а взаимодействие объектов обеспечить *функциями*, которые будут работать с данными этих структур. При классическом процедурном подходе плохо лишь то, что данные "сами по себе", а функции их обработки тоже "сами по себе", тогда как моделировать удобнее и естественнее объекты, обладающие некой целостностью.

В принципе, механизм структур позволяет решить задачу объединения данных и функций, которые их обрабатывают, в одном объекте – ведь в структуры можно включать такие объекты, как *указатели на функции*. Указатель на функцию - это переменная, которая содержит адрес функции определённого типа с определённым набором параметров. Соответственно, косвенное обращение по этому указателю представляет собой вызов функции. А если переставить указатель на другую функцию того же типа с соответствующим списком параметров, то и будет вызвана другая функция. Поведение программы становится гибким, а значит, возможности моделирования возрастают.

Приведём пример. Мы хотим написать программу, позволяющую табулировать различные функции одним и тем же кодом. Естественным решением будет описание структуры, содержащей нужную информацию о функции:

```
#include <iostream>
#define M_PI 3.1415926 /* вспомогательные */
#define EPS 1e-9       /* константы */
using namespace std;

typedef double (*function) (double);
 //указатель на функцию вида double имя(double)

struct tab { //структура-табулятор функций
        double x1, x2; //пределы табулирования
        int n;         //количество шагов от x1 до x2
        function f;    //конкретная функция, которую табулируем,
                       //заданная указателем на неё
        static void run(tab t); //функция, которая табулирует f(x)
};
```

Какая конкретно функция табулируется, будет определять указатель `f`, включённый в структуру. Передавая структуру типа `tab` в функцию табулирования `run`, мы сможем решить поставленную задачу:

```
void run(tab t) { //функция табулирования любой функции
        double dx = (t.x2 - t.x1) / t.n;
        cout << endl; cout.width(10); cout << "X";
        cout.width(10); cout << "Y";
        for (double x = t.x1; x <= t.x2 + EPS; x += dx) {
                cout << endl;
                cout.width(10); cout << x;
                cout.width(10); cout << t.f(x);
        }
}
```

Допишем программу, построив таблицы значений нескольких функций:

```
//конкретные функции для примера
double f1(double x) { return sin(x); }
double f2(double x) { return cos(x); }
double f3(double x) { return x*x; }

int main() {
        tab fun1 = { 0, M_PI, 10, f1 }; run(fun1);
         //описали и табулировали одну функцию
        tab fun2 = { 0, M_PI/2, 10, f2 }; run(fun2);
         //а теперь совсем другую
        fun1.f = f3; run(fun1);
     //программно поменяли функцию в структуре, переставив
         //указатель на другой объект
        cin.sync(); cin.get(); return 0;
}
```

В нашем коде хорошо то, что программа может динамически менять свою логику, подставляя нужные функции в структуры. Однако хватает и неудобств:

- в функцию `run` приходится передавать структуру или указатель на неё;
- по-прежнему функция `run` никак не связана со структурой;
- нельзя разделить или ограничить доступ к переменным и функциям структуры.

Можно сделать функцию "полноценным" членом структуры, но тогда всё равно придётся использовать инструментарий классов, немного забегая вперёд (показаны только изменённые участки кода):

```
struct tab {
        double x1, x2;
        int n;
        function f;
     void run(); //изменено
};

//...

void tab::run(void) { //изменено
        double dx = (this->x2 - this->x1) / this->n;
        cout << endl; cout.width(10); cout << "X";
```

```
        cout.width(10); cout << "Y";
        for (double x = this->x1; x <= this->x2 + EPS; x += dx) {
                cout << endl;
                cout.width(10); cout << x;
                cout.width(10); cout << this->f(x);
        }
}

//...

int main() {
        //изменено
        tab fun1 = { 0, M_PI, 10, f1 }; fun1.run();
        tab fun2 = { 0, M_PI / 2, 10, f2 }; fun2.run();
        fun1.f = f3; fun1.run();
        cin.sync(); cin.get(); return 0;
}
```

Примечания:

1. В этом коде `this` - "указатель объекта на самого себя", `this->x1` означает, что надо взять переменную `x1`, описанную именно внутри текущей структуры (в `fun1` при вызове `fun1.run();`), а не в другой структуре или глобально.

2. Оператор `::` имеет в C++ наивысший приоритет и означает указание области видимости объекта; в нашем случае то, что функция `run` относится к структурному типу `tab`.

В таком коде видно, что функция относится именно к структуре, мы вызываем её в виде `fun1.run()`, а не `run(fun1)`. Кроме того, мы сэкономили память стека, не передавая целые структуры внутрь функций.

Хотя у любого объекта класса есть собственная копия всех данных-членов, каждая функция-член существует в единственном экземпляре. Функция-член может обращаться к данным-членам разных объектов с помощью указателя `this`. Код с классами легче модифицируется и остаётся управляемым в больших проектах. Все современные языки и системы программирования объектно ориентированы, то есть, используют механизм классов.

Итак, объектно-ориентированное программирование (ООП) - парадигма программирования, основанная на понятиях *класса* и *объекта*.

Объект – это сущность, обладающая определённым состоянием, которое характеризуется *свойствами*, и поведением, которое характеризуется *методами*.

С точки зрения процедурного программирования, свойствам соответствуют переменные или массивы, описывающие состояние объекта, методам - функции, позволяющие управлять свойствами или обмениваться информацией с внешним миром.

Например, у класса "Студент" могут быть такие свойства как фамилия, группа, дата рождения, стипендия и т.д., методы "показать данные", "сменить номер группы", "начислить стипендию" и т.п.

Объекты всегда принадлежат одному или нескольким классам, которые определяют поведение объекта и являются его моделью. Термины "объект" и "экземпляр класса" взаимозаменяемы.

Класс - абстрактный тип данных, определяющий интерфейс и реализацию для всех своих экземпляров.

Например, объектом (экземпляром) класса "Студент" является конкретный "студент Иванов":

```
Student Ivanov;
//или чаще:
Student *Ivanov = new Student ();
```

Объекты обладают тремя базовыми свойствами (они же – **три базовых принципа ООП**):

1. *Инкапсуляция* - механизм языка, разделяющий и ограничивающий доступ к составляющим объект компонентам (методам и свойствам). Например, приватные свойства и методы класса доступны только для экземпляров этого же, но не других классов.

Так, для класса "Служащий" свойство "Зарплата" может быть приватно. А для начисления зарплаты или получения сведений о ней мы можем предусмотреть в классе публичные методы "ПолучитьВеличинуЗП", "НачислитьЗП" и т.п. Вообще, чаще всего свойства класса приватны, а основные методы публичны. Это связано ещё и с тем, что прямое изменение свойств класса в виде `объект.свойство=значение` не даёт возможности выполнения дополнительных действий и затрудняет модификацию программы.

> Пояснение этой важной мысли:
>
> Код
>
> ```
> object.property=a;
> ```
>
> конечно, выглядит естественней, чем
>
> ```
> object.setproperty(a);
> ```
>
> Но что, если свойство, изменённое нами "напрямую", меняет кто-то или что-то ещё? А если понадобилось при каждом "прямом" присваивании значения свойству сделать нечто дополнительно (сообщить в налоговую, что на счету добавилось денег)? А если нужно при присваивании провести встроенный контроль на допустимость значения? А в программе уже 1000 записей вида `object.property=a`? В случае же `object.setproperty(a)` мы изменим только один-единственный метод `setproperty` в классе объекта.

2. *Наследование* - механизм языка, позволяющий описать новый класс на основе уже существующего (родительского, базового) класса. Класс-потомок может добавить собственные методы и свойства, а также пользоваться родительскими методами и свойствами.

Например, класс "Студент" в нашей программе может быть потомком класса "Человек", и пользоваться рядом его свойств и методов ("Фамилия", "Дата рождения"), а также иметь свои, отсутствующие в базовом классе ("номер группы", "стипендия").

3. *Полиморфизм* – это единообразная обработка разнотипных данных, возможность перегрузки классами своих методов и методов классов-предков.

Например, базовый класс "Геометрическая фигура" может иметь метод "Нарисовать", по-разному реализованный в классах-потомках "Прямоугольник" и "Окружность". Но для программиста все методы будут вызываться под одним именем `draw`, что удобно.

В языках программирования, построенных на парадигме ООП, обычно имеется иерархия встроенных классов, на вершине которой находится класс с именем вроде `Object`. А все остальные классы являются его потомками того или иного иерархического "поколения".

Итак, использование классов позволяет:

- повысить степень повторного использования кода;
- обеспечить удобный механизм для коллективной разработки приложений;
- писать сколь угодно большие проекты, основанные на иерархии классов, в то время как процедурная программа длиннее 10-20 тысяч строк неизбежно становится плохо управляемой;
- более гибко управлять созданием, удалением и доступностью объектов в процессе выполнения программы;
- за счет механизма инкапусляции обеспечивать безопасность приложений.

Общий вид оператора описания класса следующий:

```
class ИмяКласса {
 private: //область видимости в пределах класса,
          //доступ имеют только функции-члены данного класса
  Список членов класса;
 protected: //могут использоваться методами данного
            //и производных от него классов
  Список членов класса;
 public:  //видимы вне класса, может осуществляться
          //доступ извне
  Список членов класса;
};
```

Например,

```
class Student {
 private:
  char *name; //имя - приватный член класса
 public:
  void show (void); //функция "показать" – публичная
  void setname (char *); //функция "установить имя" - публичная
};
```

По умолчанию члены класса приватны, так что `private:` в начале можно было не указывать.

Функции-члены связаны с именем класса оператором `::` и обычно описаны сразу после описания класса, к которому принадлежат. В остальном они выглядят как обычные функции C++:

```
void Student::show (void) {
 //функция относится к классу Student
```

```
}
```

Доступ к членам класса осуществляет оператор `.` (точка), как и для структур, он позволяет связать свойство или метод с конкретным экземпляром класса. При использовании указателей, как и со структурами, конструкция `(*указатель).поле` заменяется на `указатель->поле`.

Для того, чтобы вызываемая функция точно "знала", с каким из объектов класса она работает, Си++ неявно передает в любую нестатическую функцию дополнительный скрытый параметр — указатель на текущий объект, называемый `this`. Параметр `this` видим в теле вызываемой функции, устанавливается при вызове в значение адреса начала объекта и может быть использован для доступа к членам объекта.

```
void Student::setname (char *name) {
 strcpy (this->name, name);
}
```

Указание `this` позволит не перепутать имеющие одинаковые имена `name` параметр функции и имя одного из свойств класса.

Конструктор

Класс на Си++ может иметь любое количество *конструкторов*, предназначенных для инициализации данных класса при создании нового экземпляра. Конструктор всегда имеет то же имя, что и класс, в котором он определён, с созданием экземпляра всегда связано явное или неявное выполнение конструктора. Если отсутствует явно описанный конструктор, создается конструктор по умолчанию. Конструктор вызывается компилятором явно при создании объекта и неявно при выполнении оператора new, применяемого к объекту, а также при копировании объекта данного класса. Конструктор не возвращает значений. Как правило, он описан в public-секции класса. В остальном он оформляется как обычная функция, например с явно описанным конструктором наш класс примет вид:

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

class Student {
 private:
  char *name;
 public:
  Student (char *); //конструктор с 1 параметром
  void show (void);
  void setname (char *);
};

Student::Student (char *name=NULL) { //реализация конструктора
 if (name) {
  this->name = new char [strlen(name)+1];
  if (this->name) setname (name);
 }
 else this->name=NULL;
}
```

```
void Student::setname (char *name) {
 strcpy (this->name, name);
}

void Student::show (void) {
 cout << endl;
 if (this->name)  cout << this->name;
 else cout << "NULL";
}

int main() {
 Student *Ivanov = new Student ("Ivanov");
 Ivanov->show();
 Student Petrov("Petrov");
 Petrov.show();
 cin.sync(); cin.get(); return 0;
}
```

Заметим, что в этом несложном классе уже хватает "подводных камней", например, функция `setname` не проверяет, достаточно ли места там, куда она копирует строку:

```
Petrov.setname("Popandopulo"); //крах программы
```

С другой стороны, мы не можем "заставлять" функцию каждый раз перераспределять память, освобождая прежнее содержимое – ведь для объектов из стека, таких как `Petrov`, делать `delete` попросту нельзя. Одним из возможных решений было бы "не копировать лишнего", учитывая то, что имени может быть и не задано:

```
void Student::setname (char *name) {
 if (this->name) {
  int len = strlen(this->name);
  strncpy (this->name, name, len);
  this->name[len]='\0';
 }
}
//...
Student Petrov("Petrov");
Petrov.setname("Popandopulo");
Petrov.show(); //Popand
Student Sidorov;
Sidorov.setname("Sidorov");
Sidorov.show(); //NULL
```

Однако такая функция может породить серьёзные проблемы при копировании в неинициализированный объект наподобие того, что делается в строке

```
Student Copeikin = *Ivanov;
```

У такого объекта `name` может быть заполнено "мусором" и `len` получит неправильное значение, что неизбежно вызовет ошибку при попытке освобождения памяти. Приемлемой была бы такая реализация:

```
void Student::setname (char *name) {
 if (this->name) delete[] this->name;
 int len = strlen(name);
 if (len) {
  this->name = new char [len+1];
  if (this->name!=NULL) strcpy(this->name,name);
 }
}
```

Но и это будет работать при условии, что свойство `this->name` предварительно инициализировалось в конструкторе до вызова `setname`.

Конструктор копирования для класса `X` имеет один аргумент типа `&X` и выполняется при присваивании экземпляров класса:

```
Student *Ivanov = new Student ("Ivanov");
Student Copeikin = *Ivanov;
```

Компилятор "опознаёт" конструктор копирования потому, что его параметром является ссылка на объект класса. Это тот самый объект, что находится справа от знака "присвоить", а объект слева от "=" доступен через `this`. Напишем реализацию конструктора копирования:

```
public:
  //...
  Student (Student &); //конструктор копирования

//...
Student::Student (Student &from) { //реализация конструктора копирования
 this->name = new char [strlen(from.name)+1];
 if (this->name) setname (from.name);
}
```

Деструктор

Деструктор выполняет действия, противоположные действиям конструктора, то есть, "разрушает" объект данного класса и вызывается явно или неявно. Неявный вызов деструктора связан с прекращением существования объекта из-за завершения области его определения. Явное уничтожение объекта выполняет оператор `delete`. Деструктор имеет то же имя, что класс, но предваренное символом `~`. Деструкторы не могут получать аргументы и быть перегружены. Класс может объявить только один общедоступный деструктор. Если класс не содержит объявления деструктора, компилятор автоматически создаст его.

Для нашего класса явный деструктор может иметь вид:

```
Student::~Student () { //реализация деструктора
 if (name) delete[] name;
}
```

Здесь в области видимости внутри тела деструктора нет других переменных с именем `name`, так что `this->name` можно не писать.

При этом деструктор был описан в секции `public` класса:

```
public:
  //...
  ~Student(); //деструктор
```

Если мы, при наличии явного деструктора, в приведённом выше коде

```
Student Copeikin = *Ivanov;
```

не задали явного конструктора копирования, конструктор копирования по умолчанию выполнил `Copeikin.name = Ivanov->name`, то есть, не скопировал строку, а лишь установил указатель `Copeikin.name` на тот же адрес памяти, куда показывал `Ivanov->name`. После явного или неявного удаления объекта `Ivanov` следствием попытки показать значение `Copeikin.name` будет крах программы. Подобные "потери памяти" очень трудноуловимы, поэтому за работой явных деструкторов нужно следить особенно внимательно.

Существует "правило большой тройки" для любых классов: если в классе есть явный деструктор, то требуется явный конструктор копирования и оператор "=". Возможно, имеет смысл определять в этом случае и явный конструктор без аргументов.

Пример 1. Класс "паскалевских" массивов на C++. Для этих массивов элементы можно нумеровать с произвольного целого числа, а не только с нуля.

```
#include <iostream>
#include <windows.h>
#define MAXDOUBLE 1.797693E+308

using namespace std;

class Array {
 private:
  int low,hi;
  double *value;
 public:
  Array() : low(0), hi(0), value(NULL) {}
  Array(int);
  Array(int,int);
  ~Array();
  double get (int index);
  void set (double value,int index);
  //Способ лучше: переопределить оператор [] в классе
};

Array::Array (int k1, int k2) { low=k1; hi=k2; value = new double [k2-k1+1]; }

Array::Array (int n) : Array (0,n-1) { }

Array::~Array () { delete value; }

double Array::get (int index) {
 return (index<low || index>hi ? MAXDOUBLE : value[index-low]);
}
```

```
void Array::set (double newvalue,int index) {
 if (index>=low || index<=hi) value[index-low]=newvalue;
}

int main () {
 Array my(-5,5);
 for (int i=-5; i<=5; i++) {
  my.set(i+5.,i);
  cout << "\nKey=" << i << ", value=" << my.get(i);
 }
 my.set(my.get(5),-5);
 cout << "\nItem -5 changed=" << my.get(-5);
 cout << "\nKey=-10 (bad), value=" << my.get(-10) << endl;
 system("pause");
 return 0;
}
```

Узнав о переопределении операторов, мы сможем обращаться к объектам класса в виде `my[5]` вместо `my.get(5)`.

Оператор : (двоеточие) в конструкторе класса означает "список инициализации". Это список, в котором через запятую перечислены пары из имени члена класса и значения, которое необходимо ему присвоить, взятого в круглые скобки. Через список инициализация членов класса происходит перед тем, как выполняется вход в тело конструктора, и выполняется быстрее. К тому же, константные данные можно инициализировать только через список.

Пример 2. Определим альтернативный класс "Студент" и выполним над его объектами основные описанные в лекции действия.

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <windows.h>

using namespace std;

class Student {
 private: //Обычно атрибут дается основным свойствам
  char *Name;
  int Group;
 public: //Основные методы
  //Конструкторы:
  Student (void); //по умолчанию: Student *s=new Student();
  Student (char *,int); //фамилия и номер группы:
                        //Student *s2=new Student("Petrov",210);
  Student (Student &);  //копирования
  ~Student (); //деструктор
  void setName (char *); //Функции для получения
  char *getName (void); //и установки свойств
```

```
  void setGroup (int);  //с атрибутом private
  int getGroup (void);
  //Прочие функции
  void showStudent (void);
};

Student::Student (void) {
 Name = NULL; Group = 0;
}
Student::Student (char *newName,int newGroup=0) {
 Name =(char *)malloc((strlen(newName)+1)*sizeof(char));
 if (Name != NULL) { strcpy(Name,newName); Group = 0; }
 Group = newGroup;
}
Student::Student (Student &From) {
 setName (From.Name);
 setGroup (From.Group);
}

Student::~Student () { delete Name; }

void Student::setName (char *Name) {
 int n=strlen(Name);
 if (n>0) {
  this->Name = new char [n+1];
  if (this->Name!=NULL) strcpy(this->Name,Name);
 }
}

void Student::setGroup (int g) { Group=g; }

char * Student::getName () { return Name; }

int Student::getGroup () { return Group; }

void Student::showStudent (void) {
 printf ("\n%s,%d\n",Name,Group);
}

int main () {
 Student *NoName = new Student ();
 Student *Ivanov = new Student ("Ivanov");
 Ivanov->setGroup(110);
 Student *Petrov319 = new Student ("Petrov",319);
 NoName->showStudent();
 Ivanov->showStudent();
 Petrov319->showStudent();
 delete Petrov319;
```

```
 Student Popov("Popov",210);
 Popov.showStudent();
 Popov=*Ivanov;
 Popov.showStudent();
 system("pause");
 return 0;
}
```

Немного примеров-рассуждений о конструкторах и деструкторах есть также в этой заметке.

Оглавление 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

11.02.2016, 12:25; рейтинг: 7642