Основы ООП в C++

Артамонов Ю.Н.

Филиал «Котельники» университета «Дубна»

14 ноября 2018 г.

Содержание

Базовые понятия ООП

Человек мыслит объектами: люди, животные, самолеты, дома. Куда бы мы не посмотрели, мы везде видим объекты. При этом человеческое мышление обладает способностью к абстракции: выделению объектов на экране монитора, а не светящихся пикселей, жилых домов, а не груды кирпичей и т.д. Объекты естественным образом выстраиваются в определенную иерархию по уровню абстракции от конкретных свойств: живое существо - человек - мужчина - Сидоров. Каждый объект в нашем понимании обладает определенным набором свойств (атрибутов): у человека - это имя, цвет волос и т.д., дальнейшие уточнения атрибутов приводят к все более конкретному объекту. Кроме атрибутов, объекты обладают различным поведением: человек ходит, разговаривает и т.д.

Объектно-ориентированное программирование моделирует объекты реального мира при помощи их программных эквивалентов.

Базовые понятия ООП

ООП использует понятие *класса*, когда ряд объектов имеет одинаковые атрибуты. Также оно использует отношение *наследования* или даже *сложного наследования*, когда вновь создаваемые классы объектов получаются путем наследования характеристик существующих классов, но при этом содержат и свои собственные уникальные атрибуты, поведение. Это подобно тому, как дети наследуют свойства своих родителей, но у них всегда есть своя индивидуальность. Также как люди посылают друг другу сообщения, так и *объекты взаимодействуют посредством сообщений*. ООП скрывает - *инкапсулирует* данные (атрибуты) и функции (поведение) внутри класса. Это означает, что узнать о значениях скрытых данных, функций другие объекты могут только, если в классе реализован специальный интерфейс для доступа к ним - по принципу: можно эффективно управлять автомобилем, не зная его устройство.

Базовые понятия ООП

Инкапсуляция позволяет программисту не задумываться над внутренней реализацией кода, что существенно экономит время. Действительно, в интерфейсе класса осуществляются необходимые проверки на допустимость значений атрибутов, функций, и программист не задумывается о корректности этих значений, не погружается в изучение чужого кода.
При этом ООП является дальнейшим развитием понятия структуры.
Для демонстрации этого реализуем структуру Time:

```
struct Time{
  int hour;
  int minute;
  int second;
}
int main(){
  Time t;
  ...
}
```

Особенности и недостатки структур

При доступе к полям структуры нет никакой проверки на допустимость вводимых значений. Например, пользователь может ввести t.hour = 77; - что некорректно. Это происходит потому, что программист непосредственно манипулирует типом данных. Не сущесвует никакого интерфейса, позволяющего гарантировать, что программист корректно использует типы данных, и что данные остаются действительными. Рассмотрим реализацию данной структуры в ООП.

Пример реализации класса Time

```
#include<iostream>
using namespace std;
class Time
{
public:
  Time();
  void setTime(int, int, int);
  void printTime();
private:
  int hour;
  int minute;
  int second;
};
```

Пример реализации класса Time (продолжение)

```
Time::Time() {hour = minute = second = 0;}
void Time::setTime(int h, int m, int s)
{
  hour = (h>=0 && h<24)? h : 0;
  minute = (m>=0 && m<60)? m : 0;
  second = (s>=0 && s<60)? s : 0;
}
void Time::printTime()
{
  cout<< ((hour == 0 || hour == 12) ? 12 : hour%12)
      <<":"<< (minute<10 ? "0" : "") << minute
      <<":"<<(second<10 ? "0" :"")<<second
      <<(hour<12 ? "AM" : "PM");
}
```

Пример реализации класса Time (продолжение)

```
int main()
{
  Time t;
  cout<<"Начальное значение t"<<endl;
  t.printTime(); cout<<endl;
  t.setTime(14, 63, 17);
  cout<<"Изменили значение"<<endl;
  t.printTime(); cout<<endl;
  return 0;
}
```

Пояснения к коду

Определение класса Time начинается с ключевого слова class. Тело определения класса ограничено левой и правой фигурными скобками. Определение класса завершается точкрй с запятой. В определении класса, аналогично структуре Time, содержатся три целочисленных элемента: hour, minute, second. Метки public:, private: называются *спецификаторами доступа к элементам*. Все элементы данных и функций, объявленные после спецификатора public: (вплоть до следующего спецификатора доступа) доступны всюду, где программа имеет доступ к какому-либо объекту класса Time. Все элементы данных и функций, объявленные после спецификатора private (вплоть до следующего спецификатора доступа), доступны только для функций и данных класса. Спецификаторы доступа к элементам класса всегда заканчиваются двоеточием (:) и могут многократно появляться в определении класса.

Пояснения к коду (продолжение)

Определение класса после спецификатора public: содержит прототипы трех функций - *методов класса*: Time, setTime, printTime. Эти функции являются открытыми (публичными) и реализуют интерфейс класса, они могут использоваться для доступа и манипулирования данными класса. Функция с тем же именем, что и сам класс, называется *конструктором класса*. Конструктор - это специальная функция класса, которая инициализирует элементы данных при создании объекта (*экземпляра класса*).

Три целочисленных элемента появляются после спецификатора private:. Это означает, что эти элементы доступны только для функций класса. Именно в этом смысле говорят, что реализация класса скрыта от пользователей класса.

Имя класса становится новым типом данных. Может существовать много объектов класса, подобно тому, как много может быть переменных типа int.

Пояснения к коду (продолжение)

Определение класса содержит объявление его элементов: данных и функций. Объявления функций представляют собой прототипы функций. Функции могут быть определены внутри класса (вместо включения туда прототипа), однако хорошим стилем считается определение функций вне определения класса. Для доступа к функциям класса во внешней программе для их определения используется *операция разрешения области действия* (::), поскольку различные классы могут иметь одинаковые имена элементов. Операция разрешения области действия однозначно привязывает имя элемента к имени класса, однозначно идентифицируя функции данного класса.

Область действия класса и доступ к элементам класса

Имена переменных и функций, объявленных в определении класса, принадлежат к области действия класса. Внутри области действия класса доступ возможен просто по имени. Вне области действия класса на элементы класса можно ссылаться либо через имя объекта, либо через ссылку, либо через указатель на объект. Функции класса могут быть перегружены, но только функциями в области действия класса. Для перегрузки необходимо задать в области определения класса прототип перегруженной функции и написать для каждой версии отдельное определение.

Функции класса имеют область действия функции: если внутри такой функции определена переменная с тем же именем, что и в области действия класса, то последняя будет скрыта внутри функции. Доступ к таким скрытым переменным можно получить посредством операции разрешения области действия, поместив перед этой операцией имя класса. Доступ к элементам класса идентичен доступу к элементам структуры: точка (.), стрелка (->).

Инициализация объектов класса: конструкторы

После создания объектов их элементы могут быть инициализированы с помощью конструкторов. Конструктор представляет собой функцию класса с тем же именем, что и сам класс. Программист пишет конструктор, который затем автоматически вызывается всякий раз, когда сздается объект этого класса. Элементы класса не могут быть инициализированы в определении класса. Элементы данных либо должны инициализироваться в конструкторе, либо их значения могут быть установлены позже, после создания объекта. Конструкторы могу быть перегружены, чтобы предусмотреть различные способы инициализации объектов класса.Конструкторы могут содержать аргументы по умолчанию. Например, можно это сделать в определении класса:

```
class Time{
public:
  Time(int = 0; int = 0; int = 0);
  void setTime(int, int, int);
  ...
```

Инициализация объектов класса: конструкторы

Если для класса не определено ни одного конструктора, компилятор создает конструктор по умолчанию. Такой конструктор не выполняет никакой инициализации, поэтому после создания объекта не гарантируется его корректное состояние. Поэтому целесообразно всегда предусматривать конструктор, который выполняет соответствующую инициализацию объектов своего класса.

Деструкторы

Деструктор - это специальная функция класса. Имя деструктора состоит из символа тильды, за которым следует имя класса. Деструктор класса вызвается автоматически, когда объект класса выходит из области действия. Сам деструктор не разрушает объект, а просто освобождает память, занятую объектом для ее последующего использования. Деструктор не принимае параметров и не возвращает значения. У класса может быть только один деструктор, перегрузка деструкторов не допускается. Любая попытка передать параметры в деструктор, возвратить значение из деструктора или перегрузить деструктор будет заканчиваться сообщением об ошибке. Для простых классов деструкторы редко определяются в явном виде. Определение деструкторов целесообразно для классов, объекты которых содержат динамически выделенную память (динамические массивы, и т.д.)

Вызов конструкторов, деструкторов

Обычно конструкторы, деструкторы вызываются автоматически. Порядок, в котором происходят обращения к этим функциям, зависит от того порядка, в каком объекты входят в область действия и выходят из нее. Как правило, вызов деструкторов происходит в порядке, обратном порядку вызова конструкторов. Для объектов, объявленных в глобальной области действия, конструкторы вызываются в начале выполнения программы. Соответствующие деструкторы по ее завершении. Для локальных объектов контрукторы вызываются при их объявлении, а деструкторы, когда объект выходит из области действия. Следующий пример демонстрирует работу конструкторов и деструкторов.

Пример вызова конструкторов, деструкторов

```
#include<iostream>
#include<time.h>
using namespace std;
class create_destroy
{
public:
  create_destroy(int);
  ~create_destroy();
private:
  int data;
};
create_destroy::create_destroy(int a)
{
  data = a;
  cout<<"Объект "<<data<<" создан"<<endl;
}
```

Пример вызова конструкторов, деструкторов (продолжение)

```
create_destroy::~create_destroy()
{
  cout<<"Объект "<<data<<" разрушен"<<endl;
}
void create_obj(void)
{
  create_destroy fourth(4);
  cout<<time(NULL)<<endl;
  static create_destroy fifth(5);
  cout<<time(NULL)<<endl;
}
```

Пример вызова конструкторов, деструкторов

```
create_destroy first(1);
int main()
{
  create_destroy two(2);
  cout<<time(NULL)<<endl;
  static create_destroy third(3);
  create_obj();
  cout<<time(NULL)<<endl;
  return 0;
}
```

Задание 2.1

Создайте класс с именем Complex для выпонения арифметических операций с комплексными числами.

Задание 2.2

Создайте класс с именем Rational для выпонения арифметических операций с дробями.

Концепция отделения интерфейса от реализации

Одним из наиболее фундаментальных принципов систематической разработки программного обеспечения является отделение интерфейса от реализации. Это упрощает модификацию программ, а также поощряет поставку независимыми производителями программного обеспечения библиотек классов для продажи и лицензирования. Для этого производители разделяют свой код на две части: заголовочный файл с описанием прототипов функций, объектный файл - скомпилированная библиотека реализации прототипов функций. При этом пользователь может использовать функции из заголовочного файла в своей программе, но посмотреть их реализацию в объектном файле не может. Таким образом, не раскрывается никакой информации, связанной с авторским правом, а сообщество пользователей C++ выигрывает от большого числа доступных библиотек классов, произведенных независимыми фирмами. Рассмотрим такой подход на простом примере.

Создаем заголовочный файл Time.h (интерфейс класса)

```
class Time
{
public:
  Time();
  void setTime(int, int, int);
  void printTime();
private:
  int hour;
  int minute;
  int second;
};
```

Сохраняет данный файл с именем Time.h.

Создаем библиотеку Time.o (реализацию класса)

```
#include<iostream>
#include "Time.h"
using namespace std;
Time::Time() {hour = minute = second = 0;}
void Time::setTime(int h, int m, int s){
  hour = (h>=0 && h<24)? h : 0;
  minute = (m>=0 && m<60)? m : 0;
  second = (s>=0 && s<60)? s : 0;}
void Time::printTime(){
  cout<< ((hour == 0 || hour == 12) ? 12 : hour%12)
      <<":"<< (minute<10 ? "0" : "") << minute
      <<":"<<(second<10 ? "0" :"")<<second
      <<(hour<12 ? "AM" : "PM");}
```

Сохраняем данный файл с именем Time.cpp.

Данный файл компилируем с ключем -c, который создает объектный файл Time.o, но не линкует его:

```
g++ -c Time.cpp -o Time.o
```

Основная программа

Теперь создаем свою программу, в которой используем класс Time.

```
#include<iostream>
#include "Time.h"
using namespace std;
int main()
{
  Time t;
  cout<<"Начальное значение t"<<endl;
  t.printTime(); cout<<endl;
  t.setTime(14, 63, 17);
  cout<<"Изменили значение"<<endl;
  t.printTime(); cout<<endl;
  return 0;
}
```

Сохраним эту программу с именем UseTime.cpp. Данную программу нужно компилировать с библиотекой Time.o следующим образом:

```
g++ UseTime.cpp  Time.o
```

Задание 2.3

Отделить интерфейс от реализации в классе Complex.

Композиция классов

Одной из форм повторного использования программного кода является *композиция*, когда класс содержит в качестве элементов объекты других классов. Например, определим класс узла и создадим пул из n узлов.

Создаем заголовочный файл pull.h (интерфейс класса)

```
class node
{
public:
  node();
  void set_next(node* );
  void set_prev(node* );
  void set_value(int value);
  node* get_next();
  node* get_prev();
  int get_value();
private:
  int value;
  node* next;
  node* prev;
};
```

Создаем заголовочный файл pull.h (интерфейс класса)

```
class pull
{
public:
   pull(int);
   ~pull();
   node get_pull(int);
private:
   int n;
   node* ar;
};
```

Создаем файл pull.cpp - реализацию класса

```
#include<iostream>
#include "pull.h"
using namespace std;
node::node()
{
  value = 0;
  next = NULL;
  prev = NULL;
}
int node::get_value()
{
  return value;
}
node* node::get_prev()
{
  return prev;
}
```

Создаем файл pull.cpp - реализацию класса

```
node* node::get_next()
{
  return next;
}

void node::set_next(node* p)
{
  next = p;
}

void node::set_prev(node*  p)
{
  prev = p;
}

void node::set_value(int x)
{
  value = x;
}
```

Создаем файл pull.cpp - реализацию класса

```
pull::pull(int x)
{
  if (x>0)
    {
      n=x;
      ar = new node [n];
      for (int i=0; i<n; i++)
{
  if (i== 0)
    {
      ar[0].set_prev(NULL);
      if (n !=1)
        ar[0].set_next(&ar[1]);
      else
        ar[0].set_next(NULL);
      ar[0].set_value(0);
      continue;
    }
```

Создаем файл pull.cpp - реализацию класса

```
   if (i == (n-1))
     {
       if (n != 1)
{
   ar[n-1].set_next(NULL);
   ar[n-1].set_prev(&ar[n-2]);
   ar[n-1].set_value(n-1);
}
       continue;
     }
   ar[i].set_next(&ar[i+1]);
   ar[i].set_prev(&ar[i-1]);
   ar[i].set_value(i);
}
     }
 else
    cout<<"Ошибка создания пула";
}
```

Создаем файл pull.cpp - реализацию класса

```
node pull::get_pull(int x)
{
  if ((x>=0) &&(x<n))
    return ar[x];
}

pull::~pull()
{
  delete [] ar;
}
```

Компилируем данный файл:
g++ -c pull.cpp -o pull.o

Создаем основной файл main.cpp, использующий определенные классы

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

int main()
{
  pull T(10);
  for (int i =0; i<10; i++)
    cout<<T.get_pull(i).get_value()<<endl;
  cout<<endl;
  node* p = T.get_pull(1).get_prev();
  while (p != NULL)
    {
      cout<<p->get_value()<<endl;
      p=p->get_next();
    }
  return 0;
}
```

Компилируем данный файл: g++ main.cpp pull.o

Дружественные функции, классы

Дружественная функция класса опредена вне области действия этого класса, однако она имеет право доступа к его закрытым элементам. Функция или целый класс могут быть объявлены в качестве друзей класса. Чтобы объявить функцию в качестве друга некоторого класса, следует поместить перед прототипом этой функции в определении класса ключевое слово friend. Чтобы объявить класс Class2 в качестве друга класса Class1, следует поместить объявление вида:
friend Class2;
в качестве элемента класса Class1.
Рассмотрим следующий пример:

Пример использования дружественных функций

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

class class_test
{
  friend void set(class_test &,int);
public:
  int get_value();
  void set_value(int);
private:
  int value;
};
```

Пример использования дружественных функций

```
int class_test::get_value()
{
  return value;
}
void class_test::set_value( int x)
{
  value=x;
}

void set(class_test &c, int x)
{
  c.value = x;
}
```

Пример использования дружественных функций

```
int main()
{
  class_test T;
  T.set_value(1);
  cout<<T.get_value()<<endl;
  set(T, 2);
  cout<<T.get_value()<<endl;
  return 0;
}
```

Дружественные функции, классы

Дружба даруется, но не навязывается. Чтобы класс A дружил в классом B, в классе A должно быть это объявлено. Однако от этого класс B автоматически не становится другом класса A. Кроме этого, если A дружит с B, и B дружит с A, а также B дружит с C, то отсюда автоматически не следует, что A дружит с C.

Задание 2.4

Определить с помощью классов двухсвязный список: добавление элемента в начало, добавленние элемента в конец, добавление в заданную позицию, удаление из заданной позиции, поиск заданного элемента, сортировка элементов списка, визуализацию списка.

Задание 2.5

Определить с помощью классов стек.

Указатель this

Каждый объект поддерживает указатель на самого себя - это так называемый указатель this, который является неявным аргументом во всех ссылках на элементы внутри этого объекта. Указатель this может также использоваться явно, в этом случае каждый объект получает свой собственный адрес. Рассмотрим пример использования указателя this.

Пример использования this

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

class test
{
public:
    test(int);
    ~test();
    void set(int);
    int get();
    void print();
    void get_address();
    void get_size();
private:
    int value;
};
```

Пример использования this

```
test::test(int a){this->value = a;}
test::~test(){}
int test::get(){return this->value;}
void test::set(int x){this->value = x; }
void test::print(){cout<<(*this).value<<endl; }
void test::get_address(){cout<<this<<endl; }
void test::get_size(){ cout<<sizeof(*this)<<endl; }
int main()
{
    test T(777), D(666); T.print(); D.print();
    T.get_address(); D.get_address();
    test F(T.get()+D.get()); F.print();
    F.get_size();
    return 0;
}
```

Статические элементы класса

Обычно каждый объект класса имеет собственную копию всех элементов данных класса. Однако в некоторых случаях все экземпляры класса должны иметь общий атрибут класса. Такой атрибут может быть объявлен статическим полем класса. Использование статических элементов в классе позволяет также экономить память, если достаточно одной копии данных. Может казаться, что статические переменные подобны глобальным переменным, однако они доступны только внутри класса. Статические элементы класса существуют, даже если не существует ни одного объекта этого класса. Для обращения к открытому статическому элементу класса, когда не существует ни одного экземпляра класса, следует просто поместить перед именем элемента имя класса с операцией разрешения области действия. Для доступа к закрытому статическому элементу, должна быть предусмотрена соответствующая открытая функция класса. Продемонстрируем эти возможности на примере.

Пример использования статических полей

```
#include <iostream>
using namespace std;
class test
{
public:
    test(int);
    ~test();
    static int count_objects;
    int get_instruction();
    static void set_instruction(int);
    int get_number();
    int get_value();
    void set_value(int);
private:
    int value;
    int number_object;
    static int instruction;
};
```

Пример использования статических полей

```
int test::count_objects=0;
int test::instruction = 777;
test::test(int a){
    this->value = a;
    count_objects++;
    number_object = count_objects;
}
test::~test(){}
int test::get_instruction(){
    return instruction;
}
void test::set_instruction(int x){
    instruction = x;
}
int test::get_number(){
    return number_object;
}
```

Пример использования статических полей

```
void test::set_value(int x){
    value = x;
}
int test::get_value(){
    return value;
}
int main()
{
    test A(5),B(7);
    cout<<"Номер объекта A: "<<A.get_number()<<endl;
    cout<<"Номер объекта B: "<<B.get_number()<<endl;
    cout<<"Количество объектов: "<<A.count_objects<<endl;
    cout<<"Инструкции для объектов: "<<A.get_instruction()<<endl;
    test::set_instruction(888);
    cout<<"Инструкции для объектов: "<<B.get_instruction()<<endl;
    return 0;
}
```

Шаблоны классов

Мы уже рассматривали возможность использования шаблонов функций. Эта же возможность существует и для классов, что открывает широкие перспективы в повторном использовании кода. Например, можно написать класс работы с двухсвязным списком, которых хранит в себе то целые числа, то вещественные числа, то символы, а также любой другом тип данных, определенный пользователем. Шаблоны классов часто называют *параметризованными типами*. Программист, желающий использовать шаблоны классов, просто пишет одно обобщенное определение шаблона класса, используя абстрактное определение типа. Для этого перед определение класса нужно написать заголовок:

```
template <class T>
```

Параметризованные типы могут принимать более одного параметра, например:

```
template <class T, class S, class R>
```

Пример использования шаблонов классов

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

template <class T>
class test
{
public:
    test(T);
    bool find(T);
    void print();
private:
    T value;
};
```

Пример использования шаблонов классов

```
template <class T>
test<T>::test(T a){
value = a;
}

template <class T>
bool test<T>::find(T x){
    if (x==value)
        return true;
    else
        return false;
}

template <class T>
void test<T>::print(){
    cout<<value<<endl;
}
```

Пример использования шаблонов классов

```
int main()
{
    test<int> int_my(111);
    test<float> float_my(3.14);
    test<char> char_my('q');
    cout<<int_my.find(555)<<endl;
    cout<<int_my.find(111)<<endl;
    int_my.print();
    float_my.print();
    char_my.print();
    return 0;
}
```

Задание 2.6

Определить с помощью шаблона класса структуру стек.