

Языки программирования. Часть 2
Лекция 1
Основы создания классов

ПМИ

Демяненко Я.М.

2024

В чем же состоит принципиальное отличие языка C++ от языка C?

Брюс Эккель:

Включение функций в структуры составляет подлинную суть того, что язык C++ добавил в C

При внесении функций в структуры к характеристикам (как у структур в C) добавляется **поведение**. Так возникает концепция *класса*. В результате чего его **поля (характеристики)** и **функции (поведение)** рассматриваются как единое целое. Такое объединение получило название *инкапсуляция (encapsulation)*.

При этом существует возможность регламентировать доступ к членам класса (полям и функциям).

Три спецификатора доступа

private (закрытый) – доступен только для членов класса;

protected (защищенный) – доступен для членов класса и их наследников;

public (открытый) – доступен для всех.

Рекомендации

В основном рекомендуется делать

поля – закрытыми,

интерфейсные функции – открытыми,

служебные функции – закрытыми

Различие в доступе по умолчанию

Описание	<pre>class <имя класса> { <поля и функции> };</pre>	<pre>struct <имя класса>{ <поля и функции> };</pre>
Спецификатор доступа по умолчанию	<code>private</code>	<code>public</code>

Следующие два объявления эквивалентны

```
struct A{  
    int f();  
    void g();  
private:  
    int i, j, k;  
};
```

```
int A::f() {  
    return i+j+k;  
}
```

```
void A::g() {  
    i=j=k=0;  
}
```

```
int main(){  
    A a;  
    B b;  
    a.f(); a.g();  
    b.f(); b.g();  
}
```

```
class B{  
    int i, j, k;  
public:  
    int f();  
    void g();  
};
```

```
int B::f() {  
    return i+j+k;  
}
```

```
void B::g() {  
    i=j=k=0;  
}
```

Типы данных и переменные

Класс определяет новый **тип данных**.

Переменная данного типа называется **объектом**.

Объект также называют **экземпляром класса**.

Класс Дата

```
/* date.h */

class Date {
    private: // это слово писать не обязательно, но желательно
        int y, m, d;

    public:
        // Конструктор класса
        Date(int d, int m, int y) {
            this->d = d;
            this->m = m;
            this->y = y;
        }

        // Конструктор класса без параметров
        Date() {
            d = 0;
            m = 0;
            y = 0;
        }

        ...

        void add_days(int n);
};
```

Указатель на объект

```
Date(int d, int m, int y) {  
    this->d = d;  
    this->m = m;  
    this->y = y;  
}
```

```
Date(int dd, int mm, int yy)  
{  
    d = dd;  
    this->m = mm;  
    this->y = yy;  
}
```

Ключевым словом **this** обозначается указатель на объект в функциях-членах класса.

Если коллизии имён не возникают, то при обращении к членам класса его обычно опускают.

Указатель на объект передаётся как **неявный параметр** во все функции, которые могут вызываться только для экземпляров классов (в том числе, конструкторы и деструкторы).

Модификатор `inline`

Все функции, размещённые (определённые) внутри класса, автоматически помечаются модификатором **`inline`**.

Определение внешних функций класса осуществляется в `*.cpp` файлах.

```
/* date.cpp */
```

```
#include "date.h"
```

```
// Определение некоторой функции вне интерфейса класса
```

```
void Date::add_days(int n)
```

```
{
```

```
    ...
```

```
}
```

Новый тип должен выглядеть как встроенный

При надлежащей реализации класса Date, в C++ будут возможны действия следующего рода:

```
Date d(17, 10, 14), d1 = d;  
cin >> d1;  
Date d2(31, 12, 14);  
Date d3;
```

```
d += 7;  
d1 = d1 - 7;  
int n = d2 - d;  
if (d == d1)...  
d++; ++d; d1--; --d1;  
d2++;  
cout << d << ' ' << d1
```

Как видим, это код выглядит так, будто Date это встроенный тип.

Класс — круг

Задача

Реализовать класс для описания геометрической фигуры «круг».

Определить:

- конструкторы,
- функции-члены класса для вычисления площади,
- функции доступа к радиусу.

Реализовать сравнение объектов на равенство.

```
//circle.h
#ifndef CIRCLE_H
#define CIRCLE_H
class circle {
private:
    double x,y,r;
public:
    circle();
    circle(double x1, double y1, double r1);
    double s();
    void set_r(double r1);
    double get_r();
    bool equal(circle a);
    bool operator ==(circle a);
};
#endif
```

Конструкторы

```
//circle.cpp
#include <cmath>
#include "circle.h"

circle::circle() {
    x=10;
    y=10;
    r=10;
}
circle::circle(double x1, double y1, double r1){
    x=x1;
    y=y1;
    r=r1;
}
```

ИЛИ

```
circle::circle(double x1=10, double y1=10, double r1=10){
    x=x1;
    y=y1;
    r=r1;
}
```

Функции доступа

```
double circle::s(){
    return 3.14*r*r;

void circle::set_r(double r1){
    if (r1<0)
        r=0;
    else
        r=r1;
}

double circle::get_r(){
    return r;
}
```

Сравнение на равенство

```
bool circle::equal(circle a){
    const double eps=0.0001;
    if (abs(x-a.x)<eps && abs(y-a.y)<eps && abs(r-a.r)<eps)
        return true;
    else
        return false;
}
```

```
bool circle::operator==(circle a){
    return equal(a);
}
```

Вызовы функций - членов класса

```
//главный файл - main.cpp
#include "circle.h"
#include <iostream>

using namespace std;

int main() {
    circle a,b(0,0,1);
    cout<<a.s()<<endl<<b.s()<<endl;
    cout<<a.equal(b)<<endl;
    cout<<a.operator==(b)<<endl;
    cout<<(a==b)<<endl;
    return 0;
}
```

Перегрузка операций

В C++ для пользовательских типов данных возможна собственная реализация стандартных операций, которая называется *перегрузкой операций*.

Перегруженная операция – это **функция**, имя которой начинается со слова **operator**, за которым следует **символ операции**.

Использовать перегруженную операцию можно двумя способами:

- ✓ традиционное использование операции

```
cout<< (a==b) <<endl;
```

- ✓ вызов функции

```
cout<<a.operator==(b) <<endl;
```

Конструкторы и деструкторы

Чтобы обеспечить должную **инициализацию** каждого **объекта**, разработчик класса должен включить в него **специальную функцию**, которая называется **конструктором**.

Имя конструктора должно совпадать с именем класса.

Конструктор предназначен для **инициализации полей** начальными значениями.

Конструктор без параметров

```
circle(); //конструктор без параметров
```

```
circle(double x1, double y1, double r1); //конструктор с параметрами
```

Конструктор без параметров называется конструктором по умолчанию

Конструктор по умолчанию

В случае если **явно не описан ни один конструктор**, компилятор **автоматически (неявно) сгенерирует конструктор по умолчанию**.

Поведение неявно созданного конструктора будет таким же, как если бы он был объявлен явно без списка параметров и с пустым телом.

Необходимость конструктора без параметров

Если в дальнейшем предполагается **размещать объекты** класса **в массиве** или **в любом другом контейнере**, **необходимо объявлять конструктор без параметров**.

Когда программа достигает точки выполнения, в которой определяется объект

```
circle a,b(0,0,1);
```

происходит автоматический вызов конструктора.

Конструктор по умолчанию

Конструктор по умолчанию — это конструктор без параметров, который генерируется автоматически, только тогда, когда явно не определено ни одного другого конструктора в классе (в том числе, конструктора копии).

```
class Student {
    string name;
}

main() {
    Student s;
}
```

Этот код компилируется и работает (хотя не очень полезен).
В `main()` работает конструктор по умолчанию.

Причина неочевидных ошибок

Это свойство конструктора по умолчанию может стать причиной неочевидных ошибок.

```
class Student {
    string name;
}

main() {
    Student s;
}
```

Добавим в класс Student конструктор с инициализацией поля.

```
class Student {
    string name;
public:
    Student(string const & name) : name(name) {}
}
```

После этого main() перестанет компилироваться, потому что исчезнет конструктор по умолчанию.

Массив объектов класса Student

Аналогичная ошибка возникнет и при попытке объявить массив объектов класса Student:

```
main() {  
    Student students[3];  
}
```

Это происходит потому, что при создании массива объектов компилятор вставляет вызовы конструктора без параметров для каждого элемента массива.

Чтобы такой массив можно было создать, нужно либо добавить в класс конструктор без параметров, либо проводить инициализацию явно:

```
main() {  
    Student students[] = {Student("Vasya"), Student("Petya"), Student("Sasha")};  
}
```

Деструктор

Синтаксис **деструктора** в целом схож с синтаксисом конструктора: **имя** функции тоже определяется **именем класса**. Но чтобы деструктор отличался от конструктора, его имя **начинается с префикса ~** (тильда).

Кроме того, **деструктор всегда один** и у него **нет аргументов**.

В случае отсутствия явного задания деструктора, компилятор неявно создаёт деструктор с пустым телом.

```
class Student {
    string name;
public:
    Student(string const & name) : name(name) {}
    ~Student() {}
}
```

Вызов деструктора

Деструктор автоматически вызывается компилятором при выходе объекта из области видимости или при уничтожении объекта, размещённого в динамической памяти.

При этом очищается память, занимаемая объектом.

Если перед уничтожением объекта необходимо освободить используемые ресурсы (например, динамическую память для размещения полей объекта, открытые файлы и т.д.), то необходимо явно определить деструктор, выполняющий эти действия.

Особенности

Конструкторы и деструкторы обладают одной уникальной особенностью:
они не имеют возвращаемого значения.

В этом они принципиально отличаются от функций, возвращающих пустое значение (значение типа `void`).

Конструктор со списком инициализаторов

```
Date(int d, int m, int y) {  
    this->d = d;  
    this->m = m;  
    this->y = y;  
}
```

В C++ для инициализации полей в конструкторе используются списки инициализаторов

```
class Date {  
    int d, m, y;  
public:  
    Date(int d, int m, int y) : d(d), m(m), y(y) {}  
};
```

`d(d), m(m), y(y)` — список инициализаторов

Как избежать двойной работы

Если список инициализаторов не используется, то в случае **полей классовых типов** всё равно **вначале** будут **вызваны их конструкторы**,

а **затем** в теле конструктора класса **значения полей** будут **перезаписаны** на основе аргументов конструктора.

То есть выполняется двойная работа.

Классы — круг и прямоугольник

Задача.

Реализовать два класса для описания геометрических фигур «круг» и «прямоугольник».

Описать функцию, позволяющую совместить центры круга и прямоугольника (переместить круг), если круг можно поместить в прямоугольник.

Класс — круг

```
//shape.h
#ifndef SHAPE_H
#define SHAPE_H

class rectangle;

class circle {
private:
    double x, y, r;
public:
    circle();
    circle(double x1, double y1, double r1);
    void print();
    friend bool tofit(circle &a, rectangle &b);
};
```

Класс — прямоугольник

```
class rectangle {  
private:  
    double x1, y1, x2, y2;  
public:  
    rectangle();  
    rectangle(double x1, double y1, double x2, double y2);  
    double width();  
    double height();  
    void print();  
    friend bool tofit(circle &a, rectangle &b);  
};
```

Дружественные функции

```
bool tofit(circle & a, rectangle &b){
    if (b.width() >= 2 * a.r && b.height() >= 2 * a.r){
        a.x = (b.x1 + b.x2) / 2;
        a.y = (b.y1 + b.y2) / 2;
        return true;
    }
    return false;
}
```

В функции **tofit** используются не только функции этих классов, но и их поля.

При этом **поля** описаны как **private**, т.е. защищены от внешнего доступа.

Чтобы разрешить внешней функции доступ к защищённым полям объектов, необходимо её сделать **дружественной** для этих классов.

Для этого описание функции со спецификатором **friend** помещается в интерфейс каждого из классов:

```
friend bool tofit(circle &a, rectangle &b);
```

Проблема очередности описания

```
class circle {  
...  
    friend bool tofit(circle &a, rectangle &b);  
};  
  
class rectangle {  
...  
    friend bool tofit(circle &a, rectangle &b);  
};
```

Параметрами функции `tofit` являются объекты обоих классов. Поскольку объявление дружественности для `tofit` расположено в интерфейсах обоих классов, возникает **проблема очередности** их описания.

Решением является использование **предварительного описания** одного из классов.

```
class rectangle;
```

Работа с объектами

```
//главный файл - main.cpp
#include "shape.h"
#include <iostream>

using namespace std;

int main() {
    setlocale(0, "Russian");
    circle a(0, 0, 5);
    rectangle b(10,10,20,20);
    a.print();
    b.print();
    if (tofit(a, b)) {
        cout << "перемещение выполнено" << endl;
        a.print();
        b.print();
    }
    else
        cout << "круг не помещается в прямоугольник" << endl;
    return 0;
}
```

Доступ к защищённым членам класса

В соответствии с **принципом инкапсуляции** работа с **защищёнными членами** класса должна быть организована через **открытые функции**.

В рассматриваемом примере можно было бы организовать **геттеры** и **сеттеры** для доступа к закрытым членам классов.

Если больше ни для каких других целей он не нужны, то вместо них **можно и нужно** использовать **механизм дружественности**.

Важно, что при этом **сокращается количество вызовов функций**.

Дружественность

Объявление **функции дружественной классу** разрешает **доступ** к защищённым полям класса **только для этой функции**.

Дружественными по отношению к рассматриваемому классу могут быть **не только внешние функции, но и функции другого класса**, или даже **другой класс** (все его функции).

На функции, объявленные дружественными, **не распространяются действия спецификаторов** (public, private, protected).

Объявление функций дружественными рекомендуется располагать либо в самом начале, либо в самом конце описания класса.

Объекты типа круг и прямоугольник

Задача.

Используя классы, описанные ранее, создать объект «прямоугольник» и массив объектов класса «круг».

Посчитать количество кругов, которые можно поместить внутри прямоугольника.

Для таких кругов совместить их центры с центром прямоугольника.

Создание массива кругов

```
#include "shape.h"
#include <iostream>
#include <cstdlib> //содержит srand() и rand()
#include <ctime>   //содержит time()

using namespace std;

int main() {
    rectangle b(10, 10, 40, 40);

    circle* c[5];

    srand((unsigned)time(0));

    for (int i = 0; i < 5; ++i) {
        c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);
        c[i]->print();
    }
}
```

Основные вычисления и освобождение

```
int count = 0;

for (int i = 0; i < 5; ++i)
    if (tofit(*c[i], b))
        count++;

cout << "количество перемещённых = " << count << endl;

for (int i = 0; i < 5; ++i)
    c[i]->print();

for (int i = 0; i < 5; ++i)
    delete c[i];
}
```

Проблемы статического массива кругов

В условии задачи требуется создать массив объектов класса `circle`.

Если бы описание массива выглядело таким образом

```
circle c[5];
```

то все объекты были бы созданы с помощью конструктора по умолчанию, т.е. с одинаковыми значениями переменных-членов класса.

Создание статического массива кругов

Чтобы иметь возможность для каждого элемента массива вызывать конструктор с параметрами, необходимо разместить их в динамической памяти, т.е. при объявлении использовать массив указателей.

```
circle* c[5];
```

Затем для каждого элемента массива создать объект класса circle.

```
for (int i = 0; i < 5; ++i) {  
    c[i] = new circle(rand()%100, rand()%100, rand()%10 + 10);  
    c[i]->print();  
}
```

В конце программы динамическая память освобождается.

```
for (int i = 0; i < 5; ++i)  
    delete c[i];
```

Передача объектов в функцию

В C++ передача параметров в функцию осуществляется по значению, то есть в функцию передается копия передаваемой переменной.

Этот факт справедлив и для очень больших объектов, таких как `string` или `vector`.
Чтобы не происходило полного копирования объектов, нужно передавать их по ссылке:

```
void f(vector<int> &v);
```

```
void f(const string &s);    // Изменить строку не получится
```

Передача объектов в функцию `tofit`

```
friend bool tofit(circle &a, rectangle &b);
```

Лучше так

```
friend bool tofit(circle &a, const rectangle &b);
```