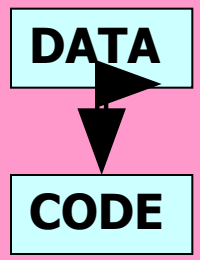
Объектное программирование

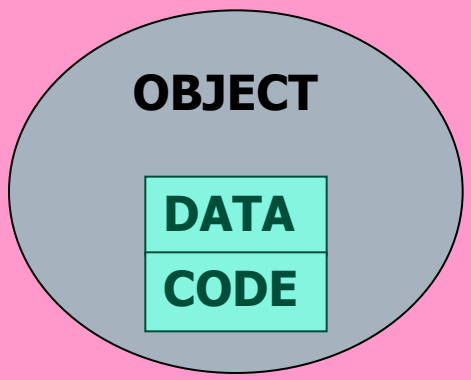
Было:



Процедурное программирование

Объектно-ориентированное

Стало:



Объект: некоторая структура, которая включает в себя данные и код, который взаимодействует с этими данными.

Для создания объектов используют специальные структуры данных, называемые *классами*.

```
class <имя класса>
{
      [описание данных]
      [описание кода]
};
```

Класс – тип данных, объект – экземпляр класса:

<имя класса> <объект1>, <объект2>,…

Вычисление в ООП рассматривается как модель поведения

Как было: структуры

Что делать, если объект описывается сложным по составу набором атрибутов?

Можно использовать структуры

Данные о сотруднике: фамилия, имя, день рождения

```
typedef  struct {
  char d;
  char m;
  short y;
} date;

struct Emp{
 char * FNam;
 char * SNam;
 date Bday;
};
…
Emp exampl;
exampl.FNam="aaa";
…
```

Определение структуры

Структура – это объединенное в единое целое множество уникально именованных элементов данных, которые могут быть разных типов.

Например, состав зачетной ведомости группы:

```
struct InspectSheet {
int Number;          //Номер в списке
char Names[40];      //Фамилия ИО
int Mark;            //Отметка
};
```

Формат определения структурного типа:

struct <имя_структурного_типа>
{<определения_элементов>**};**

Внимание! <имя_структурного_типа> в некотором смысле эквивалентна типу (int, double и т.п.), а не имени переменной.

Пример использования структуры

```
struct List {// Определяем тип данных
     // обычно в заголовочном файле
int n;
char Name[40];
};
…

void Function()
{
    struct List Student;
// Определяем переменную структурного типа
    struct List *pStudent;
// Определяем указатель на переменную структурного типа
    struct List Students[30];
// Определяем массив переменных структурного типа
…
}
```

Определение структуры - ещё

typedef struct {<определения_элементов>}
<обозначение_структурного_типа>;

В этом случае используется *безымянный структурный тип*.

```
typedef struct
{
double real;
double imag;
} complex;
```

При таком описании структурного типа определение соответствующих переменных в программе выглядит как:

```
…
    complex first;
// Определяем переменную типа complex
    complex *pfirst;
// Определяем указатель на переменную типа complex
    complex arrayN[100];
// Определяем массив переменных типа complex
```

Доступ к элементам структур

Обеспечивается с помощью уточненных имен. *Уточненное имя* – это выражение с двумя операндами и операцией «точка» между ними. Операция «точка» называется операцией доступа к элементу структуры. У нее самый высокий приоритет наряду со скобками.

Уточненное имя используется для выбора правого операнда операции «точка» из структуры, задаваемой левым операндом. Уточненные имена элементов структур обладают всеми правами объектов соответствующих типов. Их можно использовать в выражениях, их значения можно вводить с клавиатуры и т.д.

```
if (delta.real > 0) …
sigma.real +=2;
n = sigma.imag * sigma.imag;
strcpy(work, Var1.Name);
```

Элементы структур - массивы

В качестве элементов структуры могут указываться массивы. Предположим мы хотим описывать изображение, представленное совокупностью разноцветных шаров в пространстве. Каждый шар тогда определяется структурой

```
struct BALL{
char color;
double radius;
float coord[3];
} ball ={'r', 3.4, {1.0, 2.2, -3.5}};
```

Удаление центра шара от начала координат может быть записано как

```
int i;
double s=0.0;
for (i=0; i<3; i++)
  s += ball.coord[i]* ball.coord[i];
printf("%8.2lf", sqrt(s));
```

Массивы структур

Структуры могут группироваться в массивы.

```
complex V1 [100];
 struct COMPLEX V2[200];
```

Для доступа к элементам структур, представленных массивом, используются уточненные имена с индексированием первого имени (массива структур).

```
struct BALL VarBalls[100];
…
…VarBalls[2].color      //Цвет третьего шара
…VarBalls[k+1].coord[0]
          // Первая координата k+1-го шара
```

Указатели на структуры

```
struct COMPLEX *pC;
complex *pcmpl;
```

Можно вводить указатели в качестве обозначений структур

```
struct birth {
char Where[40];
struct date When;
} *pB1, *pB2;
```

Возможно также:

```
typedef struct COMPLEX
{
double real;
double imag;
} complex, *ptr_comp;
```

для определения переменной типа указатель на структуру:

```
ptr_comp px [12];
complex * px [12];
```

- одинаково определяют массивы из 12-ти указателей на структуру complex.

Доступ к элементам через указатель

Доступ к элементам структуры, определенной через указатель

осуществляется одним из двух способов:

****указатель на структуру.имя элемента***

указатель на структуру -> имя элемента

Первый способ традиционный, он основан на равенстве

если **pAdr == &adr**, то ***pAdr == adr.**

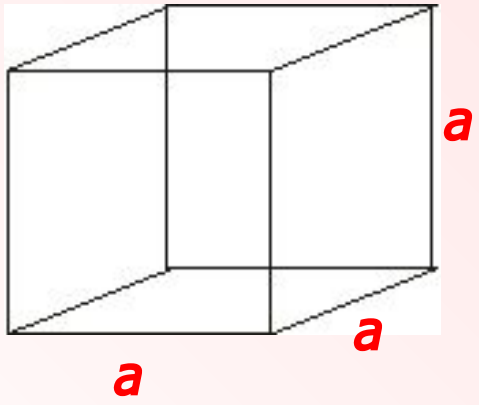
Обычно применяют второй способ, связанный с использованием специальной операции «->». Эта операция, наряду со скобками и операцией «.» имеет наивысший приоритет. Операция «стрелка» обеспечивает доступ к элементу структуры через адресующий ее указатель того же структурного типа. Операция «стрелка» иногда называется *операцией косвенного выбора элемента структурного объекта, адресуемого указателем*.

```
if (pdelta->real > 0) …
psigma->real +=2;
d = ptr_c->imag * ptr_c->imag;
strcpy(work, pVar1->Name);
```

Определение класса

Программируя, мы всегда строим некоторую специальную модель фрагмента реального мира!

Объект – ***куб***



В описание объекта входят данные и методы (в виде функций)

Данные: – сторона куба ***a***

Вычисления = методы:

– объем $V = a^3$

– площадь поверхности $S = 6 a^2$

Один из простейших видов описания соответствующего класса в языке C++:

```
class Cube {
public:
   double a;

   double Volume() { return a*a*a;}
   double Surface () { return 6*a*a;}
};
```

Здесь мы имели дело со статическим объектом, т.е. проводимые вычисления трудно ассоциировать с поведением

Объявление и определение методов класса

Объявление (declaration) методов класса обычно выносится в заголовочный файл. При объявлении класса достаточно определить только интерфейс встроенных в него методов. При объявлении прототипов функций возможно употребление спецификаторов **friend**, **static**, **virtual** and **inline** и квалификатора **const**. (Хотя **friend** делает функцию внеклассовой).
Определение (definition) методов (их программная реализация) обычно осуществляется в файле исходного текста на языке C++.

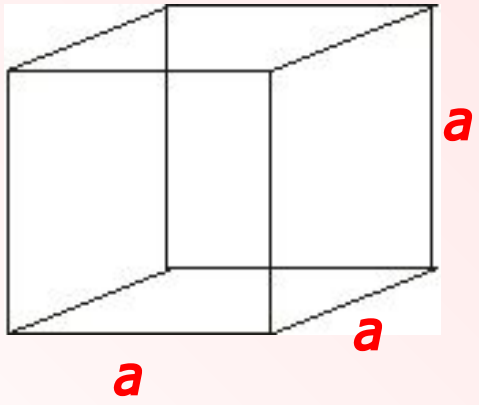
```
// Это в файле cube.h
class Cube {
public:
   double a;

   double Volume();
   double Surface ();
};
```

```
// Это в файле cube.cpp
#include <cube.h>

double Cube::Volume()
{
   return a*a*a;
}

 double Cube::Surface ()
{
  return 6*a*a;
}
```

Использование класса

Как определить объект куб в программе на языке С++?

Объект – ***куб***



Один из вариантов использования класса:

Предположим, что описание класса Cube выполнено в файле с именем cube.h.

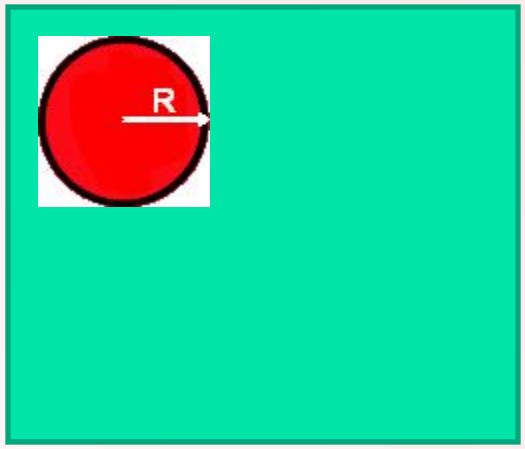
Файл cube.cpp может быть таким:

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

void main()
{
  Cube c1;
  c1.a =2;
 cout<<" Volume = " << c1.Volume() << endl;
 cout<<" Surface = " << c1.Surface() << endl;
};
```

Объект и поведение

А теперь динамический объект!

Объект – ***цветной круг***, двигающийся на плоскости



В описание объекта входят:

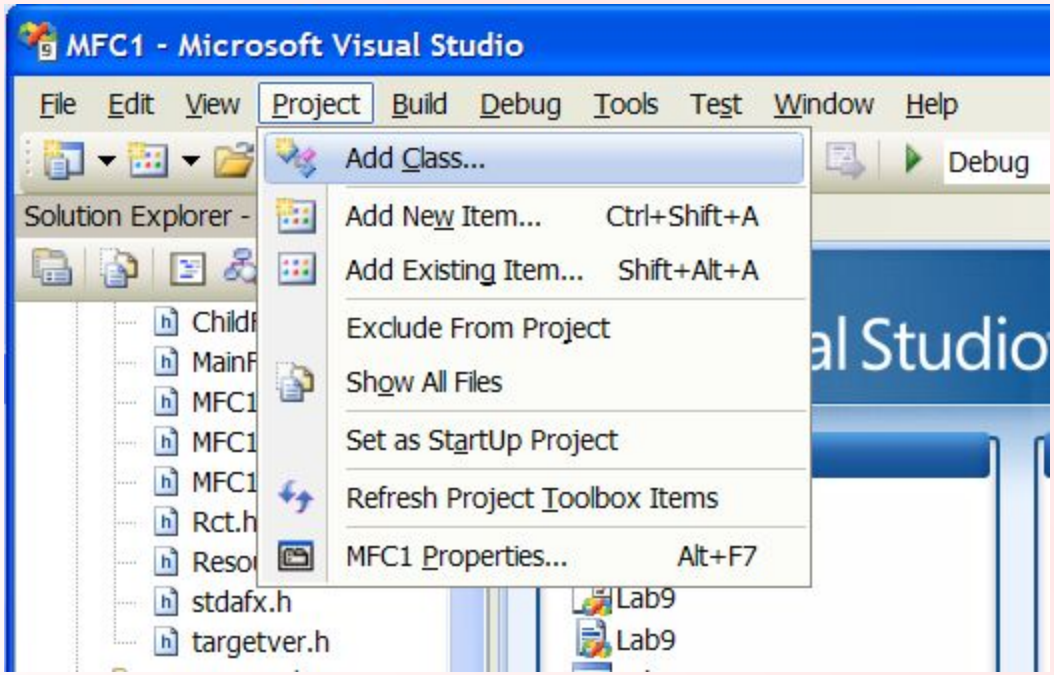
данные: – радиус круга ***R***
– координаты центра круга (***X*** и ***Y***)
– цвет круга ***Color***

методы: – площадь круга $S = \pi R^2$
– сдвиг круга на плоскости
– замена цвета круга

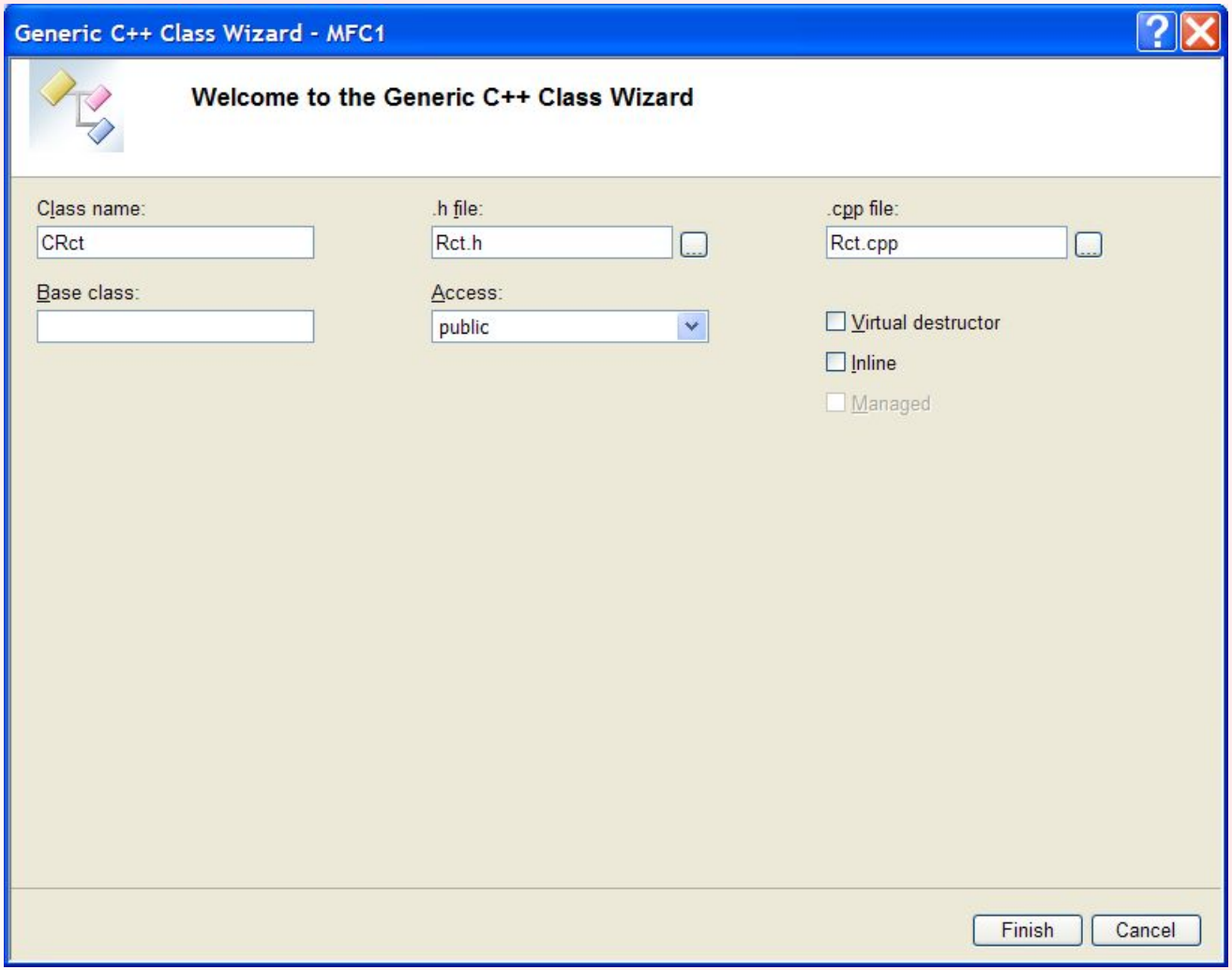
Описание соответствующего класса в языке C++:

```
class circle
 {
public:
   double R;
   int X, Y;
   long Color;

   double Area() { return 3.14 * R * R; }
   void Shift(int dX, int dY) { X += dX; Y += dY; }
   void ChangeColor(long NewColor)
   { Color = NewColor; }
 };
```

Создание класса при помощи Мастера 1



Создание класса при помощи Мастера 2



Создание класса при помощи Мастера 3

Файл Rct.h:

```
#pragma once

class CRct
{
public:
  CRct(void);    // Конструктор
  ~CRct(void); // Деструктор
};
```

Файл Rct.cpp:

```
#include "StdAfx.h"
#include "Rct.h"

CRct::CRct(void)
{
}

CRct::~CRct(void)
{
}
```

Конструктор

Как описать действия, выполняемые при создании нового объекта?

Конструктором называется метод, одноименный с именем класса и выполняемый при создании объекта данного класса.

Конструктор – это метод. Метод – это функция. Функция может иметь набор аргументов (параметров).

Существует 2 формы определения конструктора с параметрами:

Cube (double d) { a=d;}

это эквивалентно

Cube (double d): a(d) { }

Функция может не иметь параметров – это значит, что конструктор может быть задан в форме:

Cube () { a=1.0; }

При выполнении такого конструктора создается объект куб, размер ребра которого автоматически будет установлен в 1.

Конструктор по умолчанию

Почему введенные ранее нами классы Cube и circle не имели в своем описании конструкторов?

Наряду с перечисленными формами существует конструктор либо не имеющий параметров, либо все аргументы которого заданы по умолчанию – ***конструктор по умолчанию***:

Cube ( ) { a = 1.0; } или
Cube(double = 1.0)) { } **эквивалентно** **Cube ( ): a(1.0) { }**

Каждый класс может иметь только один конструктор по умолчанию.

Если в описании класса такой отсутствует, то компилятор генерирует его ***автоматически***.

Однако!
Генерация проводится только в случае отсутствия описания других конструкторов класса!

Конструктор реализуется как функция, не возвращающая никакого значения. Это означает, что в конструкторе нельзя использовать операторы return (выражение)

Если класс не имеет конструктора, то массивы объектов конкретного типа размещаются системой автоматически. Если класс имеет конструктор, но не имеет конструктора по умолчанию, то распределение массива приведет к синтаксической ошибке!

Множество конструкторов класса

В одном классе может быть множество конструкторов?

Все конструкторы класса должны отличаться друг от друга, чтобы компилятор мог определить, какой из них использовать в конкретной ситуации.

Но имя конструктора должно совпадать с именем класса.

Это означает, что единственной возможностью для различия альтернативных конструкторов является состав принимаемых параметров.

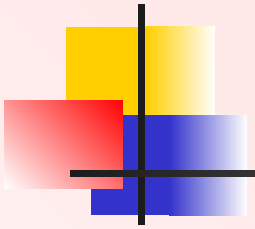
В качестве примера предположим, что в состав класса **circle** дополнительно включены конструкторы:

1) circle ( ) { R = 0.0; X = Y = 0; Color = 0; } //Конструктор по умолчанию

2) circle (double r);

3) circle (double r, int x, int y);

4) circle (double r , int x, int y, long Color);



Деструкторы

Деструктором называется метод, вызываемый при разрушении объекта данного класса.

Имя деструктора совпадает с именем класса и начинается с символа ~.

Пример: **~Cube( ) { }**.

В каждом классе может быть только один деструктор.

Деструктор не имеет параметров.

Деструктор не может быть перегружен.

Если в описании класса деструктор явно не определен, то компилятор будет генерировать его автоматически.

Деструктор не выполняет действия, связанные с освобождением занятой объектом памяти, а предшествует этому освобождению.

Чаще всего потребность в деструкторе возникает при необходимости освобождения захваченной объектом динамической оперативной памяти!

Деструкторы - пример

```
class Cl
{
public:
  Cl(int number) { n = number; ptr = new int[n]; };
  ~Cl() { delete [] ptr; };

private:
  int n;
  int *ptr;
};

Cl ccll(10);
```

Конструкторы и деструкторы - пример

```
class cl {
    int id;
public:
    cl(int n) {id = n; cout << "Constructor " << id << endl; }
    ~cl() {cout << "Destructor " << id << endl;}
} obj1(1);

void func () { static  cl  obj5(5); }

void main()
{
   cout << "Begin\n";
   cl obj2(2);
   {
       cl obj3(3);
   }
   cl obj4(4);    func (); func (); func ();
       cout << "End\n";
}
```

```
Constructor 1
Begin
Constructor 2
Constructor 3
Destructor 3
Constructor 4
Constructor 5
End
Destructor 4
Destructor 2
Destructor 5
Destructor 1
```

Конструкторы и деструкторы - правило

Деструкторы могут вызываться из программы. Однако на практике этого применять не стоит, особенно, если работа деструктора связана с освобождением памяти.

Конструкторы глобальных объектов вызываются в первую очередь, до выполнения первого действия в функции main(). Деструкторы глобальных объектов вызываются самыми последними – после того, как отработала функция main().

Жизнь локальных объектов, как и жизнь обычных переменных встроенных типов ограничивается областью видимости. Конструкторы таких объектов вызываются в месте создания объекта, деструкторы – после выхода из блока (области видимости).

Конструкторы статических объектов вызываются один раз при первом достижении команды создания объекта, а деструкторы – после завершения функции main().

Если несколько объектов создаются в одном блоке, то конструкторы таких объектов вызываются в порядке создания объектов, а деструкторы – в обратном порядке.

Конструктор копирования

Как можно построить (создать) объект как копию уже существующего?

Среди всего множества альтернативных конструкторов класса выделим еще один, тот, который описан с единственным параметром – объектом создаваемого при помощи этого конструктора класса. В качестве примера такого конструктора рассмотрим

circle (const circle& Exist);

Такой конструктор носит название ***конструктора копирования.***

Предполагается, что семантика этого конструктора связана с построением копии объекта-параметра.

Если в описании класса такой конструктор отсутствует, то компилятор будет генерировать его ***автоматически***.

Конструктор копирования - примеры

```
#include "iostream"
using namespace std;
class Cl {
public:
   int *ptr;
   Cl(int numb) { ptr = new int(numb); cout << "Constructor\n"; }
   ~Cl() { delete ptr; cout << "Destructor\n"; }
   void print () { cout << "*ptr = " << *ptr << endl; }
};

void main()
{
   Cl obj1(5);
   {
      Cl obj2 = obj1;
      obj2.print();
   }
   Cl obj3(10);
   obj1.print();
}
```

Результат

```
Constructor
*ptr = 5
Destructor
Constructor
*ptr = 10
```

Первый раз конструктор вызывается для obj1. При создании объекта obj2 конструктор не вызывается, т.к. происходит побитное копирование obj1 в obj2. Это означает, что значение указателя ptr в обеих объектах одно и то же. При выходе из блока вызывается деструктор obj2, который освобождает память под переменную целого типа – сразу для двух объектов!

Так получилось, что в программе сразу после блока строится новый obj3, который в конструкторе получает тот же адрес памяти, который был освобожден – это и объясняет приведенный результат выполнения программы

Конструктор копирования – примеры 2

Добавим:

```
Cl(const Cl & obj)
{
    ptr = new int(*obj.ptr);
    cout << "Copy constructor\n";
}
```

```
#include "iostream"
using namespace std;
class Cl {
public:
   int *ptr;
   Cl(int numb) { ptr = new int(numb); cout << "Constructor\n"; }
   ~Cl() { delete ptr; cout << "Destructor\n"; }
   void print () { cout << "*ptr = " << *ptr << endl; }
};

void main()
{
   Cl obj1(5);
   {
      Cl obj2 = obj1;      // Cl obj2 (obj1);
      obj2.print();
   }
   Cl obj3(10);
   obj1.print();
}
```

Эквивалентно !

Результат

```
Constructor
Copy constructor
*ptr = 5
Destructor
Constructor
*ptr = 5
Destructor
Destructor
```

Конструктор копирования – примеры 3

Конструктор копирования вызывается только при явной инициализации объектов, когда одному объекту присваивается другой. В противном случае компилятор воспринимает оператор (=) как обыкновенную операцию присваивания и производит побитное копирование содержимого одного объекта в другой. Существует еще одна ситуация, при которой вызывается конструктор копирования. Если описана функция, которая возвращает объект (т.е.временный объект, созданный внутри функции).

Начало …

```
#include "iostream"

using namespace std;
class C {
  int n;
public:
  C(int numb) { n = numb; cout << "Constructor\n"; }
  C(const Cl & obj){ n = obj.n; cout << "Copy constructor\n"; }
  ~C() { cout << "Destructor\n"; }
  void print () { cout << "n = " << n << endl; }
};
```

Конструктор копирования – примеры 3

```
Cl fn()
{
  cout << "begin:fn" << endl;
  Cl tmp(0);
  cout << "end:fn" << endl;
  return tmp;
}

void main()
{
  cout << "begin:main" << endl;
  Cl obj1(5); obj1.print();
  obj1 = fn(); obj1.print();
  cout << "end:main" << endl;
}
```

Окончание …

Результат

```
begin:main
Constructor
n = 5
begin:fn
Constructor
end:fn
Copy constructor
Destructor
Destructor
n = 0
end:main
Destructor
```

При вызове функции *fn* создается временный объект, предназначенный для хранения возвращаемого объекта.

Когда происходит инициализация временного объекта локальным объектом *tmp* внутри функции *fn()*, происходит вызов конструктора копирования.

После этого работают деструкторы для временного объекта и объекта *tmp*.

Значение временного объекта копируется в объект obj1 без вызовов конструктора копирования и деструктора.

Тесты ...

Вопрос: **Скомпилируется ли следующий код:**

```
class cls
{
public:
   cls() { }
   ~ cls() { }
   void f(cls * p){ delete p; }
};

void main()
{
        cls* p = new cls;
        p->f(p);
}
```

Тесты …

Вопрос: **Скомпилируется ли следующий код:**

```
class cls
{
public:
   cls() { }
   ~ cls() { }
   void f(cls * p){ delete p; }
};

void main()
{
        cls* p = new cls;
        p->f(p);
}
```

Ответ «да».
Не смотря на то, что объект уничтожает сам себя. Метод f нормально отработает и будет вызван деструктор.

Тесты …

Вопрос: **Что выведет следующая программа:**

```
#include "iostream"
#include <stdio.h>

void main()
{
    std::cout << printf("boom!");
}
```

Варианты ответа:

1. printf("boom!")
2. Нет правильного ответа
3. boom!
4. Ошибка компиляции (stdio только в стандартной библиотеке языка C)
5. Ошибка компиляции (несоответствие типов)

Тесты …

Вопрос: **Что выведет следующая программа:**

```
#include "iostream"
#include <stdio.h>

void main()
{
    std::cout << printf("boom!");
}
```

должно быть выведено "boom!5", т.к. printf() возвращает int, равный количеству символов, выведенных в stdout. std::cout по умолчанию синхронизирован с stdout, поэтому вначале в stdout попадет "boom!", а затем - "5».

Варианты ответа:

1. printf("boom!")
2. Нет правильного ответа
3. boom!
4. Ошибка компиляции (stdio только в стандартной библиотеке языка C)
5. Ошибка компиляции (несоответствие типов)

Тесты …

Вопрос: **Какое утверждение о следующем коде верно:**

```
int main(int argc, char* argv[])
{
  int a[3] = { 1, 2, 3 };
  int b[2] = { 1, 2 };
  a = b;
  return 0;
}
```

Варианты ответа:

1. Все нормально
2. Ошибка времени выполнения
3. Ошибка компиляции

Тесты …

Вопрос: **Какое утверждение о следующем коде верно:**

```
int main(int argc, char* argv[])
{
  int a[3] = { 1, 2, 3 };
  int b[2] = { 1, 2 };
  a = b;
  return 0;
}
```

Варианты ответа:

1. Все нормально
2. Ошибка времени выполнения
3. Ошибка компиляции

```
error C2440: '=' : cannot convert from 'int [2]' to 'int [3]'
```