Языки программирования
Лекция 3

ПМИ Семестр 2

Демяненко Я.М.

2024

Зачем создавать конструктор копии?

Допустим такое определение класса

```
class myvector {
    int size;
    int * data;
    string name;

public:
    myvector(string const & name = "id1"): size(8), name(name) {
        data = new int [size];
        cout << name << " created\n";
    }

    ~myvector()     {
        delete [] data;
        cout << name << " killed\n";
    }
};
```

Ожидание ⟶ Реальность

Проверим работу класса, выполнив следующий код в функции main:

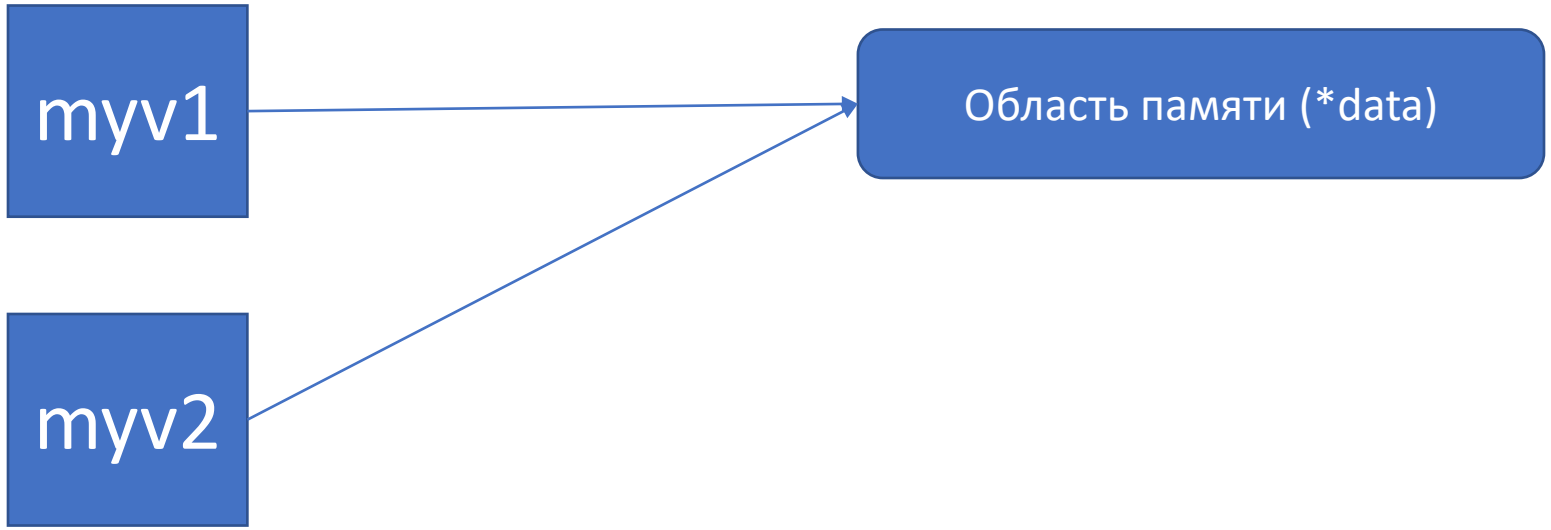
```
myvector myv1;
myvector myv2 = myv1;
```

В строке `myvector myv2 = myv1;` мы подразумеваем, что происходит копирование объекта `myv1`, а в конце выполнения функции `main()` — удаление двух объектов: `myv1` и `myv2`.

В действительности, при запуске программы происходит ошибка:

id1 created
id1 killed
*** Error in `./main': double free or corruption (fasttop): …

Что происходит в случае использования конструктора копии, создаваемого автоматически?



Вместо копии массива будет создана копия ссылки на него, т. е. ссылки двух разных объектов будут указывать на одно и то же место в памяти.

Реализация конструктора копии для класса Array

```
Array::Array (const Array &B){
  n=B.n;
  A=new int[n];
  for (int i=0; i<n; i++)
    A[i]=B.A[i];
  cout << "copy from "  <<name<< " to " <<B.name;
}
```

Теперь вывод программы выглядит следующим образом:

id1 created

copy from id1 to id2

id2 killed

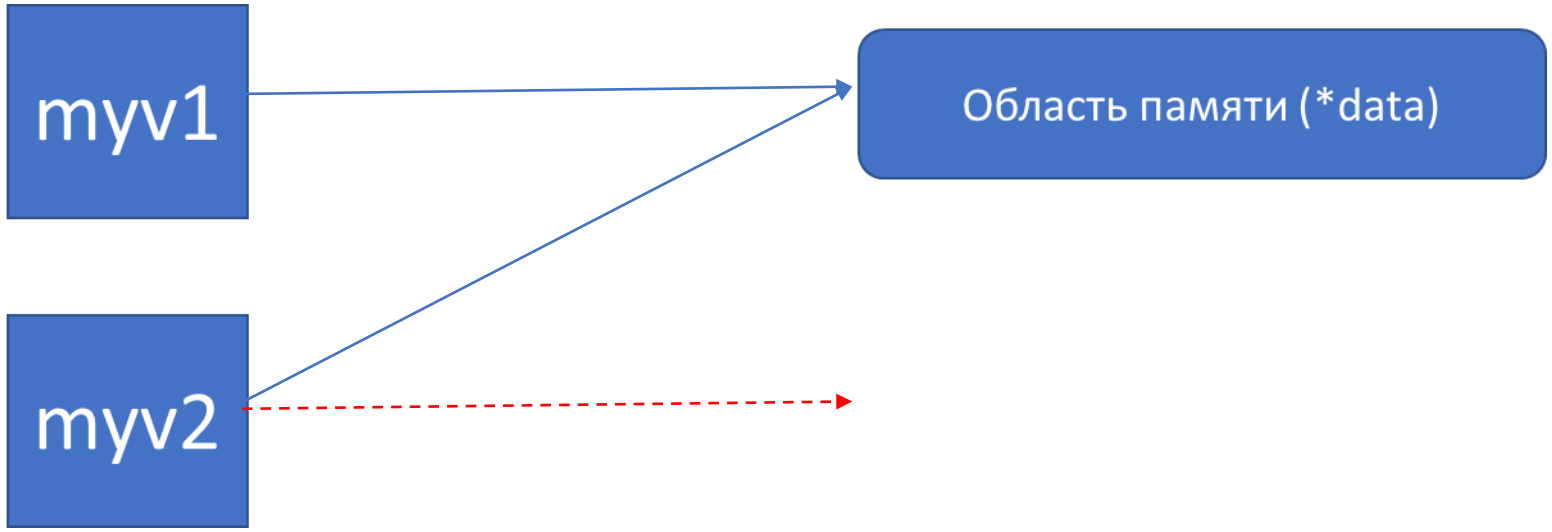
id1 killed

В каких случаях необходимо создавать конструктор копии?

Для классов, размещающих данные в динамической памяти, необходим конструктор копии.

Для классов, использующих размещение данных в динамической памяти, отсутствие конструктора копии является грубой ошибкой.

Ошибки двойного освобождения памяти




```
{
  Array myv1;
  Array myv2;
  myv2 = myv1;
...
}
```

//вызываются деструктор для объекта myv1 и деструктор для myv2
Здесь произойдёт ещё и утечка памяти, так как старое значение указателя A в объекте myv2 потеряется.

Функции-члены, которые генерируются «молча»

Рассмотрим следующий (не очень полезный) класс.

```
class Empty {};
```

На самом деле, такое описание класса эквивалентно следующему:

```
class Empty{
public:
    Empty() {}                          //конструктор без параметров

    Empty(Empty const &) {/* … */}  //конструктор копий

    Empty & operator=(Empty const &) {/* … */}
    //операция копирующего присваивания

    ~Empty() {/* … */}                  //деструктор
};
```

Операция копирующего присваивания

```
Array& Array::operator = (const Array &B) {
  if (this != &B) {
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

Функция `operator=` должна быть обязательно функцией класса.

Игнорирование проверки самоприсваивания `a=a`
в случае использования динамической памяти в определении класса может привести к потере данных.

Какие могут быть проблемы?

```
Array Array::operator = (const Array &B) {
  if (this != &B) {
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

В данной реализации операции могут появиться проблемы при возникновении исключения в операции new.

Поскольку к этому моменту уже выполнена операция delete[] A, объект после исключения в new останется в «полуразрушенном» состоянии.

Уровни гарантии безопасности кода

В C++ выделяют три уровня гарантий безопасности кода при возникновении исключений:

- базовый – при возникновении исключения не возникает утечек ресурсов, однако объекты могут находиться в непредсказуемом состоянии;
- строгий – если во время операции произошло исключение, то объект будет находиться в том же состоянии, что до начала операции;
- без исключений – в данном коде не может возникнуть исключений.

Какая гарантия?

```
Array Array::operator = (const Array &B) {
  if (this != &B) {
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

Базовая гарантия

```
Array Array::operator = (const Array &B) {
  if (this != &B) {
    delete[] A;
    n=B.n;
    A=new int[n];
    for (int i=0; i<n; i++)
      A[i]=B.A[i];
  }
return *this;
}
```

Строгая гарантия

```
Array & Array::operator = (const Array &B) {
  if (this != &B) {
    n=B.n;
    int * newdata = new int[n];
    for (int i=0; i<n; i++)
      newdata[i]=B.A[i];
    delete[] A;
    A = newdata;
  }
return *this;
}
```

Повторяющиеся действия

Обе версии реализации operator= выполняют действия, которые используются в других функциях, а именно в деструкторе и в конструкторе копии

Идиома *copy-and-swap*

Предполагает реализацию операции копирующего присваивания с использованием конструктора копий. При этом требуется вначале создать вспомогательную функцию-член `swap(Array & other)`, для обмена содержимого текущего объекта с объектом `other`.

```
void Array::swap(Array & other) {
  swap(n, other.n);
  swap(A, other.A);
}

Array& Array::operator=(Array other) {//вызов конструктора копии
  this -> swap(other);
  return *this;
}
```

Идиома copy-and-swap позволяет разрабатывать **устойчивые к исключениям операторы присваивания** и сокращает количество кода в них ценой определения полезной вспомогательной функции swap.

Рекомендации

В случае использования **динамической памяти** в реализации класса – настоятельно рекомендуется **реализовывать конструкторы и деструктор**.

Среди конструкторов обязательно должен присутствовать **конструктор копии**.

И обязательно определять **операцию присваивания**

Реализация преобразования типов

- **Конструктор преобразования** (конструктор **с единственным аргументом**) может быть использован для преобразования объектов разных типов (включая встроенные типы) **в объекты данного класса**.

- **Операция преобразования** (называемая также операцией **приведения**) может быть использована для преобразования **объекта одного класса в объект другого класса или в объект встроенного типа**. Такая операция преобразования должна быть нестатической функцией-членом. Операция преобразования этого вида не может быть дружественной функцией.

Класс frac дроби

Экземпляр класса frac хранит число f в виде отношения m/n, где m и n целые числа типа int.

В данном классе необходимо реализовать преобразование объекта `frac` к типу `double`.

```
frac f(1, 3);
double d = 1/3.0; //эквивалентно d=f;

frac f(3);

frac(int a, int b):ch(a),zn(b){}
frac(int a):ch(a),zn(1){}
```

Конструктор преобразования

В реальной программе, работая с классом `frac`, мы хотим писать так: `frac f = 2;`
То есть запись вида `f = 2;` должна быть эквивалентна `f = frac(2);`

Для решения этой задачи существует **специальный конструктор**, который называется **конструктором преобразования**.

Любой конструктор, который **может быть вызван с одним** параметром является конструктором преобразования и служит для неявного преобразования параметра к типу объекта данного класса.

Это значит, что в нашем случае этот конструктор уже реализован.

То есть запись `f = f1 + 3;` эквивалентна `f = f1 + frac(3);`

Ещё одна перегрузка операции *

Рассмотрим
`f = 2 * f2;` преобразуется к `f = frac(2) * f2;`
эквивалентно `f = frac(2,1) * f2;`

Однако обратим внимание на запись `frac(2) * f2`, здесь умножить целое число на дробь эффективнее, чем делать преобразование `frac(2,1)` и умножать дробь на дробь.

Поэтому, для ускорения работы перегрузим `operator*`:

```
…
friend frac operator*(int n, const frac &f) {
    return frac(n * f.m, f.n);
}
```

Подвох в конструкторе с одним параметром (конструкторе преобразования)

Рассмотрим такой код:

```
myvector<int> v(10), v1(10); // 10 нулей
v1 = v + 10;
```

Скорее всего, делая запись `v1 = v + 10`; мы подразумеваем увеличение размера вектора на 10 и как следствие получение в `v1` нового вектора из 20 нулей.

На деле `v1 = v + 10`; эквивалентно `v1 = v + myvector<int>(10);`
То есть мы получим еще один вектор из 10 нулей.

Для того чтобы избежать данной ошибки, необходимо запретить преобразовывать 10 к `myvector<int>(10)`.

Для этого существуют **явные конструкторы преобразования**.

Явные конструкторы преобразования

Явный конструктор преобразования задается с помощью ключевого слова **explicit**, поэтому его еще называют **explicit-конструктор**.

```
class myvector {
    …
public:
    explicit myvector(int n) {…}
}
```

Теперь компилятор запретит выражения типа `v1 = v + 10;`

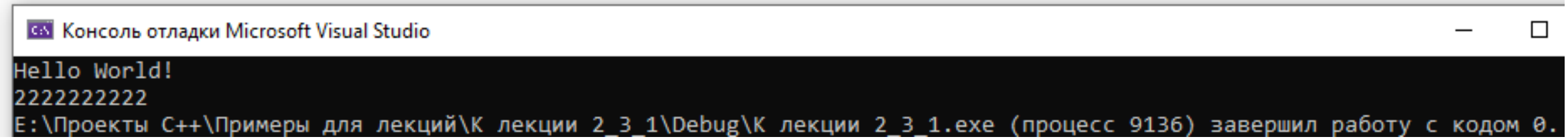
Если же необходимо сложить два вектора, тогда надо явно указывать выполняемую операцию как

```
v1 = v + myvector(10);
```

```
class myvector {
private:
   int size;
   int* a;
public:
myvector(int n) : size(n) {
       a = new int[n];
       for (int i = 0; i < size; ++i)  a[i] = 1;
   }

   myvector(const myvector& c) : size(c.size) {
       a = new int[size];
       for (int i = 0; i < c.size; ++i)
           a[i] = c.a[i];
   }
   ~myvector() {    delete[] a;  }

   friend ostream& operator<< (ostream & out, const myvector & c) {
       for (int i = 0; i < c.size; ++i)
           out << c.a[i];
       return out;
   }
```

```
    friend myvector operator+ (const myvector& b,const myvector& c) {
        myvector d(5);
        for (int i = 0; i < c.size; ++i)
            d.a[i] = b.a[i] + c.a[i];
        return d;
    }
};


int main()
{
    std::cout << "Hello World!\n";
    myvector a(5);
    myvector f(a);
    cout << a+5;
    cout << 5+a;
    return 0;
}
```

```cpp
int main()
{
    std::cout << "Hello World!\n";
    myvector a(5);
    myvector f(a);
    cout << a+5;
    cout << 5+a;
    return 0;
}
```



Операции приведения типа

```
frac f = frac(2, 3);
double d = f;
```

В данном случае операция `double d = f` не сработает.
Необходимо определить операцию приведения типа `operator double()`

```
class frac {
    int m, n;
…
public:
    operator double()      {
        return m/(double)n;
    }
};
```

Поскольку `operator double()` является `inline`, запись `double d = f;` будет заменяться на `double d = f.m/(double)f.n;`

Ограничения на перегрузку операций

Для некоторых операций перегрузка запрещена

(sizeof, ., ::, ?: и пр.)

Следующие операции можно перегружать только в классе (методом класса)

присваивание =
вызов функции ()
индексация [ ]
доступ через указатель ->
приведение типа

Операции можно перегружать только как внешние:
ввода/вывода

Конструктор преобразования char* в Str

```
Str::Str(const char* s) {
  length=(int)strlen(s);
  sPtr=new char[length + 1];
  strcpy(sPtr,s);
}
```

Операция приведения к char*

```
Str::operator char*() const {
  char *c=new char[length];
  strcpy(c,sPtr);
  return c;
}
```

Если `s` – объект класса `Str`, то когда компилятор встречает выражение `(char*)s`, он порождает вызов

`s.operator char*()`

Для этого вызова операнд `s` – это объект класса `Str`, для которого была активизирована функция-член класса `operator char*`

Неявный вызов операции приведения к char*

Если в программе в том месте, где ожидается `char*`, появился объект `s` определенного пользователем типа `Str`, то в этом случае компилятор для преобразования объекта в `char*` вызывает перегруженную функцию-операцию приведения `operator char*` и использует в выражении результирующий `char*`. Например:

```
cout << s;
```

Операция приведения для класса `Str` позволяет не перегружать операцию `<<` (поместить в поток), предназначенную для вывода `Str`

explicit

В C++11 и позже ключевое слово **explicit** применимо и к операциям преобразования.
По аналогии с конструкторами, оно защищает от непредвиденных неявных преобразований.

Статические члены класса

Если данные-члены объявлены с квалификатором **static**, то для всех объектов класса поддерживается только одна копия таких данных.

Статический член используется совместно всеми объектами данного класса.

Для того чтобы существовал статический член, не обязательно, чтобы существовали объекты такого класса.

Пример. Реализовать класс, позволяющий подсчитывать количество существующих объектов данного класса.

Объявление статических членов класса

```
class Counter {
private:
  static int count;
public:
  static int getCount() {
    return count;
  }
  Counter() {
    ++count;
  }
  ~Counter() {
    --count;
  }
};
```

Инициализация и использование статического члена класса

```
int Counter::count = 0;

int main() {

  cout << "before " << Counter::getCount() << endl;

  Counter first;
  cout << "after first " << first.getCount() << endl;

  Counter * second = new Counter();
  cout << "after second " << second->getCount() << endl;

  delete second;
  cout << "after delete second " << Counter::getCount() << endl;

  return 0;
}
```

Правила для статических членов класса

Внутри класса возможно **только объявление** статического члена, но **не** его **определение**.

```
private:
  static int count;
```

Статические данные можно **использовать** только **после определения**.

Это делается путем **нового объявления статической переменной**, причем используется оператор разрешения области видимости для того, чтобы идентифицировать тот класс, к которому принадлежит переменная.

```
int Counter::count = 0;
```

Статические члены-данные подчиняются **правилам доступа** к членам класса.

Однако **определение** статических членов-данных выполняется **вне зависимости** от спецификатора доступа.

Рекомендации и правила для статических членов

Определение статических членов-данных класса рекомендуется располагать вместе с определением самого класса в заголовочном файле.

Статические функции-члены не могут обращаться к нестатическим данным или вызывать нестатические функции этого же класса.
Это связано с тем, что в статические функции не передаётся указатель **this** на объект, для которого вызвана функция.

Реализация статической функции записывается так же, как и реализация любой другой функции-члена класса.
При этом если реализация вне класса, то ключевое слово static не указывается.

Например, определение статической функции getCount вне класса могло бы выглядеть следующим образом

```
int Counter::getCount() {
    return count;
}
```

Семантика перемещения

Семантика перемещения появилась в стандарте C++11.

Она нацелена на **уменьшение количества создаваемых копий** объектов при выполнении конструктора копии и операции присваивания, которые вызываются для **rvalue** выражений.

lvalue- и rvalue выражения

Любое выражение в C++ является или левосторонним (lvalue), или правосторонним (rvalue).

Выражение **lvalue** — это объект, который имеет имя. Все переменные являются lvalue.

А выражение **rvalue** — это временный безымянный объект, не существующий за пределами того выражения, которое его создало.

Пример. Реализовать move-семантику на примере упрощённого класса для динамического массива.

Для этого достаточно:

- определить конструктор с параметрами по умолчанию;
- определить конструктор копии;
- определить деструктор;
- перегрузить операцию сложения двух массивов одинакового размера;
- перегрузить операцию присваивания.

```
class myvector {
private:
  static int f;
  string name;
  int size;
  int * vect;
public:
  myvector(int s = 1, string nm ="noname"): size(s), name(nm){
    f++;
    cout << "constructor > " << name<<" "<<f<< endl;
    vect = new int[s];
  }
```

```
myvector(const myvector & v): size(v.size){
    f++;
    name = "Copy ( "+v.name+" )";
    cout << "copy > " << name << " " <<  f << endl;
    vect = new int[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }


~myvector(){
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
```

```
myvector& operator= (const myvector &v){
    cout << name <<" operator= > " << v.name <<" "<< f << endl;
    if (&v != this) {
      delete[] vect;
      vect = new int[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }
```

```
 myvector operator+(const myvector& v){
    if (size == v.size) {
      myvector v1(size, name + " + "+v.name);
      for (int i = 0; i < size; ++i)
        v1.vect[i] = vect[i] + v.vect[i];
      return v1;
    }
    return *this;//лучше использовать исключение
  }
};
```

```
#include "vect.h"
int myvector::f = 0;
int main() {

  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```

Командная строка

```
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy ( A ) 3
constructor > A + B 4
copy > Copy ( A + B ) 5
destructor > A + B 4
A operator= > B 4
constructor > A + Copy ( A ) 5
copy > Copy ( A + Copy ( A ) ) 6
destructor > A + Copy ( A ) 5
B operator= > Copy ( A + Copy ( A ) ) 5
destructor > Copy ( A + Copy ( A ) ) 4
destructor > Copy ( A + B ) 3
destructor > Copy ( A ) 2
destructor > B 1
destructor > A 0
```

Именно для них будут создаваться дополнительные временные объекты, которые после использования в конструкторе копии и операции присваивания тут же будут удалены.

Выражениями **rvalue** являются

a + b внутри вызова конструктора копии для объекта cc

и **a + D** в операторе присваивания.