



# Программирование на C++ и Python

## Лекция 2. Классы 2

Воробьев Виталий Сергеевич (ИЯФ, НГУ)

2 октября 2019, Новосибирск

# План лекции

- Продвинутые классы
  - Список инициализации полей
  - Перегрузка операторов
  - Статические поля и методы
- Передача аргументов по ссылке
- Ключевое слово `const`
- `lvalue`, `rvalue` и семантика перемещения
- Работа с динамической памятью
  - Умные указатели

# Продвинутые конструкторы

- Что происходит при создании объекта класса:
  - выделение памяти для полей
  - инициализация полей *в порядке их объявления*. Для полей-объектов вызываются их конструкторы
  - выполняется тело конструктора (код внутри фигурных скобок)
- В текущей реализации происходит двойная работа: сначала поля инициализируются, а потом выполняется присваивание
- Для сложных объектов такая двойная работа может стоить дорого

```
class Foo {  
    int _a;  
    int _b;  
  
public:  
    Foo(int a, int b) {  
        // _a и _b уже проинициализированы  
        _a = a;  
        _b = b;  
    }  
};
```

# Продвинутые конструкторы

- Избежать двойной работы при создании объекта (инициализация, а потом присваивание) позволяет специальный механизм: *список инициализации полей*
- Кроме того, такой способ инициализации позволяет
  - использовать одинаковые имена для полей и параметров конструктора
  - инициализировать константные поля

```
class Foo {  
    const int a;  
    int b;  
  
public:  
    Foo(int a, int b) :  
        a(a), b(b) {}  
};
```

# Перегрузка операторов

Пусть у нас есть класс для работы с матрицами `Matrix`, в котором определен метод сложения `plus()`. Тогда часть кода, в которой складываются две матрицы, могла бы выглядеть так:



```
Matrix A, B;  
Matrix C = A.plus(B);
```



```
Matrix A, B;  
Matrix C = A + B;
```

Такая программа выглядит гораздо естественней

- C++ позволяет это сделать — это называется *перегрузкой операторов* (*operator overloading*)
- Перегрузка операторов — определение стандартных операторов C++ для работы с нестандартными типами (классами)

# Перегрузка операторов

- В C++ перегрузить можно почти любой оператор:

- А вот эти нельзя:

- . (выбор поля)
- \* (указатель)
- :: (область видимости)
- ?: (тернарный условный оператор)

+	-	*	/	%	^
&		~	!	,	=
<	>	<=	>=	++	--
<<	>>	==	!=	&&	
+=	-=	/=	%=	^=	&=
=	*=	<<=	>>=	[]	()
->	->*	new	new []	delete	delete []

- Нельзя также
  - создавать новые символы для операций
  - менять приоритеты операций
  - менять интерфейс: унарный оператор остается унарным, бинарный – бинарным, сложение возвращает такой же тип, сравнение – логический тип (bool) и т.п.

# Перегрузка операторов: пример I

```
class Time {
    int h, m, s;
public:
    Time() : h(0), m(0), s(0) {}
    Time(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}
    void print() const {
        cout << "Time is " << h << ':' << m << ':' << s;
    }

    Time operator+(Time t) {
        t.h += h + (m+t.m+(s+t.s)/60)/60;
        t.m += (m + (s+t.s)/60) % 60;
        t.s += s % 60;
        return t;
    }
};
```

\*Перегруженный оператор не обязан быть членом класса

```
int main() {
    Time now(12,40,00);
    Time pair(1,30,00);

    Time after = now + pair;
    after.print();

    (now+pair).print();

    (now+Time(1,30,00)).print();
};
```

# Разберем подробнее...

- Что происходит при выполнении `now + pair`: это эквивалентно вызову функции

```
now.operator+(pair);
```

- Создается копия объекта `pair`, которая передается в функцию `operator+`
- После `return` из функции `operator+` копия объекта `pair` удаляется
- В случае, если `pair` – большой объект (например, матрица 1000x1000), копировать его при выполнении любой операции страшно неэффективно! Мы хотим работать с исходным объектом `pair` внутри функции, не создавая его копии.
- В языке `C` мы добивались этого, передавая указатель на объект (массив, структуру, ...). Но тогда вместо `now+pair` нам пришлось бы писать `now+(&pair)` – очень некрасиво и непонятно.
- Для решения подобных проблем в `C++` были добавлены ссылки



# Ссылки (references)

- *Ссылка* – это специальный тип данных, который позволяет создавать псевдоним для переменной

```
int i = 1;
int& n = i;  // n - это ссылка на i
n = 10;      // теперь i равно 10
cout << i << endl;  // 10
```

- Ссылки похожи на указатели, но:
  - нельзя создавать нулевые ссылки, они обязательно инициализируются при создании
  - ссылку нельзя изменить

**Ссылки часто используются для передачи аргументов в функцию**

# Передача аргументов по ссылке

- Передача аргументов по значению:

```
void add_five(int i) {  
    i = i + 5;  
}
```



```
n = 1;  
add_five(n);  
cout << n << endl; // 1
```

- Передача аргументов через указатель:

```
void add_five(int* i) {  
    *i = *i + 5;  
}
```



```
n = 1;  
add_five(&n);  
cout << n << endl; // 6
```

- Передача аргументов через ссылку:

```
void add_five(int& i) {  
    i = i + 5;  
}
```



```
n = 1;  
add_five(n);  
cout << n << endl; // 6
```

```
class Sensor {  
    double value;  
public:  
    void setValue(double x) {value = x;}  
    virtual void print() const {  
        cout << "Sensor value is " << value << endl;}  
};
```

```
class HumiditySensor : public Sensor {  
public:  
    void print() const override {  
        cout << "Humidity is " << getValue() << endl;  
    }  
};
```

```
void print_sensor1(Sensor s) { s.print(); }  
void print_sensor2(Sensor& s) { s.print(); }
```


Пример: передача по ссылке и наследование

```
HumiditySensor h;  
h.setValue(10.0);  
h.print(); // Humidity is 10  
print_sensor1(h); // Sensor value is 10  
print_sensor2(h); // Humidity is 10
```

# Перегрузка операторов: пример II

```
class Time {
    int h, m, s;
public:
    Time() : h(0), m(0), s(0) {}
    Time(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}
    void print() const {
        cout << "Time is " << h << ':' << 'm' << ':' << s;
    }

    Time operator+(Time& t) {
        t.h += h + (m+t.m+(s+t.s)/60)/60;
        t.m += m + (s+t.s)/60;
        t.m = t.m % 60;
        t.s += s;
        t.s = t.s % 60;
        return t;
    }
};
```



```
int main() {
    Time now(12,40,00);
    Time pair(1,30,00);

    Time after = now + pair;
    after.print();

    (now+pair).print();

    (now+Time(1,30,00)).print();
};
```

# Целостность аргументов

- С помощью ссылок мы научились избегать лишнего копирования при передаче аргументов в функции (методы).
- Но теперь переменные и объекты, которые передаются в функцию, можно испортить!

```
Time after = now + pair;
```

```
Time operator+(Time t) {  
    t.h += h + (m+t.m+(s+t.s)/60)/60;  
    t.m += m + (s+t.s)/60;  
    t.m = t.m % 60;  
    t.s += s;  
    t.s = t.s % 60;  
    return t;  
}
```

Все нормально:  
испорчена копия **pair**

```
Time operator+(Time& t) {  
    t.h += h + (m+t.m+(s+t.s)/60)/60;  
    t.m += m + (s+t.s)/60;  
    t.m = t.m % 60;  
    t.s += s;  
    t.s = t.s % 60;  
    return t;  
}
```



Ошибка: испорчена  
копия **pair**

# const

- С помощью ключевого слова `const` C++ позволяет указать, что конкретную переменную или объект запрещено модифицировать
- Это позволяет гарантировать сохранность объектов, которые передаются в функции по ссылкам

```
const int i = 100;
i = 1; // Ошибка!

void add_five(const int& i) {
    i = i + 5;    // Ошибка!
}
```

```
const int i = 100; // i - константа
```

```
const char* s = "abc";
// s - указатель на константную строку
// s[0] = 'v'; - нельзя
// s = "vbn"; - можно
```

```
char* const s = "abc";
// s - константный указатель на строку
// s[0] = 'v'; - можно
// s = "vbn"; - нельзя
```

```
const int* func(int);
// int, указатель на который вернет функция,
// менять нельзя
```

```
int func(int) const;
// если func - метод класса, ей запрещено
// менять любые* поля класса
```

# Перегрузка операторов: пример III

```
class Time {
    int h, m, s;
public:
    Time() : h(0), m(0), s(0) {}
    Time(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}
    void print() const {
        cout << "Time is " << h << ':' << 'm' << ':' << s;
    }

    Time operator+(const Time& t) const {
        Time tnew;
        tnew.h = t.h + h + (m+t.m+(s+t.s)/60)/60;
        tnew.m = (t.m + m + (s+t.s)/60) % 60;
        tnew.s = (t.s + s) % 60;
        return tnew;
    }
};
```

```
int main() {
    Time now(12,40,00);
    Time pair(1,30,00);

    Time after = now + pair;
    after.print();

    (now+pair).print();

    (now+Time(1,30,00)).print();
};
```

# Перегрузка операторов: пример IV

```
class Time {
    const int h, m, s;
public:
    Time() : h(0), m(0), s(0) {}
    Time(int hh, int mm, int ss) : h(hh), m(mm), s(ss) {}
    void print() const {
        cout << "Time is " << h << ':' << 'm' << ':' << s;
    }

    Time operator+(const Time& t) const {
        return {
            h + t.h + (m+t.m+(s+t.s)/60)/60,
            (m + t.m + (s+t.s)/60)%60,
            (s + t.s)%60
        };
    }
};
```

```
int main() {
    Time now(12,40,00);
    Time pair(1,30,00);

    Time after = now + pair;
    after.print();

    (now+pair).print();

    (now+Time(1,30,00)).print();
};
```



# Суммируем I

1. С помощью *списка инициализации полей* можно создавать выразительные и эффективные конструкторы класса
2. C++ позволяет определять для любых классов операции типа сложения (+), умножения (\*) и др. Это называется *перегрузкой операторов*
3. C++ позволяет определить псевдоним любой переменной или объекта – *ссылку*. Ссылки нужны, чтобы избежать копирования объектов при передаче их в качестве аргументов функций
4. Ключевое слово **const** C++ позволяет защитить переменные от изменения
5. С помощью ключевого слова **const** можно объявлять методы класса, которые не изменяют состояние объекта (*константные методы*)

# Напоминание: стек и куча

- В программе всегда есть две принципиально разных области памяти: *стек* (stack, автоматическая память) и *куча* (heap, динамическая память)

## Стек

- все локальные переменные создаются в стеке
- переменные автоматически уничтожаются при выходе из области видимости, в порядке, обратном порядку создания
- стек очень быстрый

```
int i;  
Date d;
```

## Куча

- для создания переменных в куче надо использовать специальные функции
- работа с кучей ведется через указатели
- пользователь ответственен за удаление объектов из кучи
- куча заметно медленнее стека (но и заметно больше)

```
int *pi = malloc(1*sizeof(int));
```

# Динамическая память

- В языке C для работы с динамической памятью использовались функции malloc() и free(). В C++ для этого есть специальные операторы: new, new[], delete и delete[]

```
int *pi = (int *)malloc(sizeof(int));
*pi = 5;
free(pi);

int *a = (int *)malloc(5*sizeof(int));
free(a);
```

```
int *pi = new int(5);
delete pi;

Date *p = new Date(18,9,2018);
delete p; // вызывается ~Date()

int *a = new int[5]; // array
delete[] a;
```

```

class Matrix {
    const int dim;
    double *data;

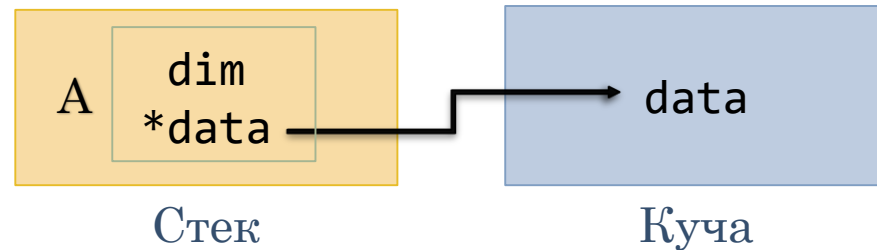
public:
    Matrix(int n) : dim(n) {
        data = new double[dim*dim];
    }
    ~Matrix() { delete[] data; }

    double operator()(int i, int j) const {
        return data[i*dim+j];
    }
    double& operator()(int i, int j) {
        return data[i*dim+j];
    }
};

```



- В конструкторе выделяем память, в деструкторе – освобождаем



```

#include <iostream>
using namespace std;

int main() {
    Matrix A(5); // матрица 5*5

    A(1,1) = 5.0;
    cout << A(1,1) << endl;
}

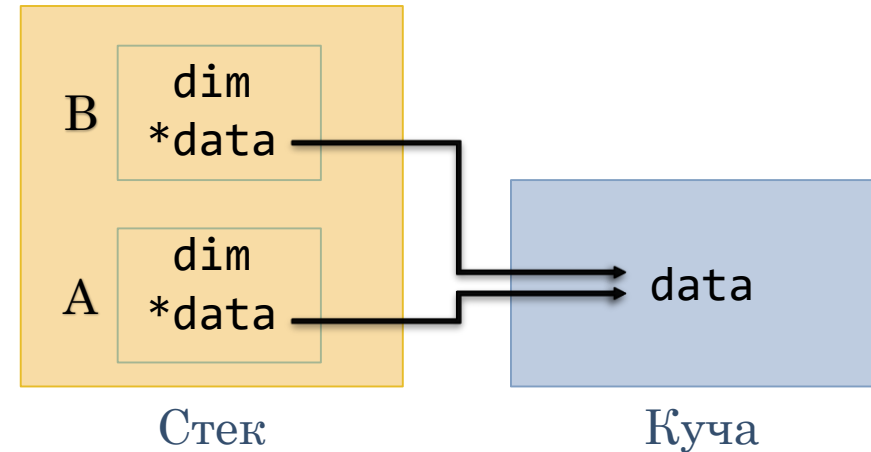
```

# Проблема копирования

- Создадим две матрицы:

```
int main() {  
    Matrix A(5); // матрица 5*5  
  
    Matrix B = A;  
    B(1,1) = 10;  
    cout << "B[1,1]=" << B(1,1) << endl;  
  
    A(1,1) = 5;  
    cout << "A[1,1]=" << A(1,1) << endl;  
    cout << "B[1,1]=" << B(1,1) << endl;  
}
```

```
B[1,1]=10  
A[1,1]=5  
B[1,1]=5
```



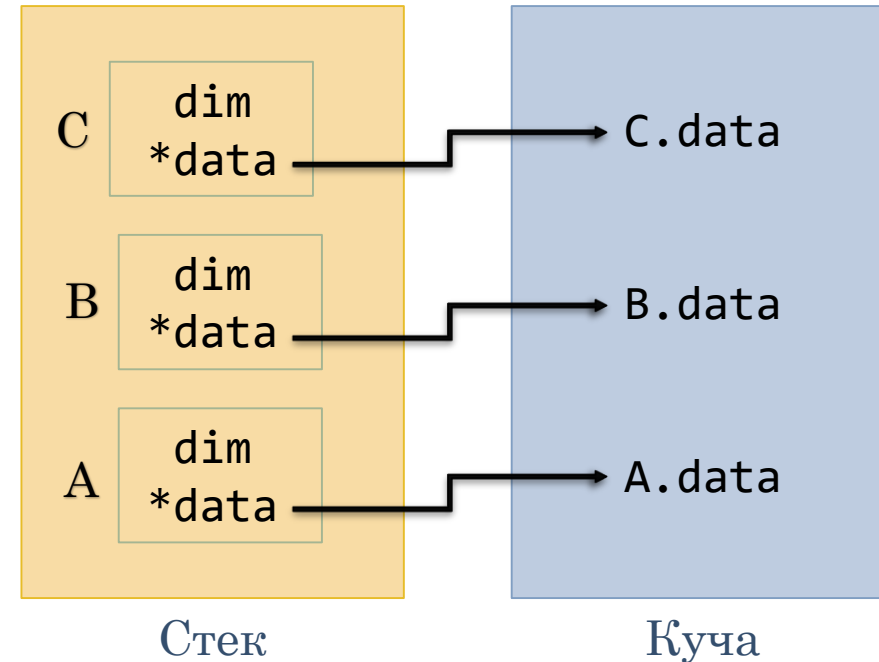
- При копировании `B = A` выполнялось копирование каждого поля: `B.dim = A.dim`, `B.data = A.data`, и теперь оба объекта A и B указывают на один и тот же массив в памяти!
- Деструкторы объектов A и B два раза попытаются освободить одну и ту же память. Это приведёт к *неопределённому поведению*

# Конструктор копирования

```
class Matrix {  
...  
public:  
    Matrix(const Matrix& M) : dim(M.dim) {  
        data = new double[dim*dim];  
        for (int i = 0; i < (dim*dim); ++i) {  
            data[i] = M.data[i];  
        }  
    }  
};
```

```
int main() {  
    Matrix A(5); // матрица 5*5  
    Matrix B = A; // конструктор копирования  
    Matrix C(A); // конструктор копирования  
}
```

- Конструктор копирования отвечает за создание нового объекта на основе старого
- Для создания копии матрицы необходимо выделить новую память и скопировать туда содержимое исходной матрицы



# Оператор присваивания

- Конструктор копирования вызывается только при создании объекта (как и любой другой конструктор):
- А как правильно реализовать присваивание?
- Необходимо перегрузить оператор присваивания =
  - очистить уже используемую память.
  - правильно обработать самоприсваивание:  $A = A$ ;
  - вернуть ссылку на объект, чтобы можно было присваивать по цепочке:  $A = B = C$ ;

```
Matrix& operator=(const Matrix& M) {  
    // самоприсваивание  
    if (this == &M) return *this;  
    // удаление старой матрицы  
    if (data != nullptr) delete[] data;  
    if (M.data != nullptr) {  
        data = new double[dim*dim];  
        for (int i = 0; i < (dim*dim); i++) {  
            data[i] = M.data[i];  
        }  
    }  
    return * this;  
}
```

```
// конструктор копирования  
Matrix B = A;
```

```
Matrix B;  
B = A; // присваивание
```

# Виртуальный деструктор

- Деструктор, как и другие методы класса, могут быть виртуальными. Если в классах-наследниках происходит нетривиальная работа с памятью, то конструктор базового класса следует объявлять виртуальным. В этом случае всегда будет вызываться верный деструктор

```
> A dtor  
> A dtor
```



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

class A {
public:
    ~A () {
        cout << "A dtor" << endl;
    }
};

class B : public A {
public:
    ~B () {
        cout << "B dtor" << endl;
    }
};

int main() {
    vector<A*> vec {
        new A(),
        new B()
    };
    for (auto ptr : vec) delete ptr;
    return 0;
}
```





# Виртуальный деструктор

- Деструктор, как и другие методы класса, могут быть виртуальными. Если в классах-наследниках происходит нетривиальная работа с памятью, то конструктор базового класса следует объявлять виртуальным. В этом случае всегда будет вызываться верный деструктор

```
> A dtor  
> B dtor  
> A dtor
```



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

class A {
public:
    virtual ~A () {
        cout << "A dtor" << endl;
    }
};

class B : public A {
public:
    ~B () {
        cout << "B dtor" << endl;
    }
};

int main() {
    vector<A*> vec {
        new A(),
        new B()
    };
    for (auto ptr : vec) delete ptr;
    return 0;
}
```

# Другая проблема копирования

- Как работает выражение `Matrix C = A + B`:
  1. Вызывается оператор сложения матриц, внутри которого *создается* новая временная матрица и заполняется поэлементной суммой
  2. Создается матрица C, в которую *копируется* содержимое временной матрицы с помощью *конструктора копирования*
  3. Временная матрица удаляется
- Память для матрицы C выделялась дважды. Сначала при создании временной матрицы, потом при копировании ее в C.
- Добиться оптимальной эффективности можно с помощью *конструктора перемещения*
- Также существует *оператор перемещения*, позволяющий избежать лишнего копирования при присвоении *временных объектов*

```
Matrix A(5);  
Matrix B(5);  
// Заполняем A и B  
Matrix C = A + B;
```

# lvalue и rvalue

- *lvalue* — это выражение, для которого существует адрес. Все выражение, которые при присваивании находятся *слева*, являются lvalue
- *rvalue* — это безымянное выражение, которое существует только до тех пор, пока оно вычисляется
- **Оператор &** обозначает ссылку на lvalue выражение
- **Оператор &&** обозначает ссылку на rvalue выражение

```
int x = 1;    // x - lvalue, 1 - rvalue
int y = x + 1; // y - lvalue, x + 1 - rvalue
int z = x + y; // z - lvalue, x + y - rvalue
```

Многие функции возвращают rvalue:

```
string getName() {
    return "Name";
}

// string& sref = getName(); // error
string&& sref = getName();
```

# Конструктор перемещения, оператор перемещения

```
Matrix::Matrix(Matrix&& M) :  
    dim(M.dim), id(count), data(M.data) {  
    M.data = nullptr;  
}  
  
Matrix& Matrix::operator=(Matrix&& M) {  
    if (this == &M) return *this;  
    if (data != nullptr) delete[] data;  
    data = M.data;  
    M.data = nullptr;  
    return *this;  
}
```

```
Matrix A(3);  
Matrix B(3);  
Matrix C = A + B;    // move ctor  
Matrix D(C + A);     // move ctor  
Matrix E = move(A);  // move ctor  
A(1, 0) = 0;         // error!
```

# Огласите весь список!

- Есть шесть специальных методов, которые компилятор старается создать по умолчанию

```
class Matrix {  
public:  
    Matrix();           // конструктор по умолчанию  
    Matrix(const Matrix&); // копирующий конструктор  
    Matrix(Matrix&&);     // перемещающий конструктор  
    Matrix& operator=(const Matrix&); // оператор присваивания  
    Matrix& operator=(Matrix&&); // оператор перемещающего присваивания  
    ~Matrix(); // деструктор  
};
```

- Если в классе происходит нетривиальная работа с памятью, скорее всего вам следует реализовать все эти методы самостоятельно

# this, nullptr и другие мелочи

- В любом методе можно использовать predefined указатель `this`. Он всегда указывает на тот объект, для которого вызван метод.
- В C++ есть специальное ключевое слово для нулевого указателя: `nullptr`
- C++ позволяет запретить любой метод, используя ключевое слово `delete`. Например, если вы хотите запретить копирование объектов класса `Matrix`, можно написать:

```
class Matrix {  
    ...  
public:  
    Matrix& operator=(const Matrix& M) = delete;  
};
```

В стандартной библиотеке C++ есть контейнеры, в которых аккуратно решены все вопросы работы с памятью. В частности, для класса `Matrix` можно использовать стандартный контейнер `vector`.

# Статические члены класса

Каждый объект класса занимает свою область памяти. Значения полей класса для разных объектов независимы. Но что, если мы хотим определить поле класса, которое используется всеми объектами этого класса?

Например, мы хотим идентифицировать каждый объект его порядковым номером. Тогда нам нужен счетчик объектов класса, и этот счетчик должен быть общим для класса и доступен всем объектам

Для создания таких общих полей класса можно использовать ключевое слово **static**. Инициализировать статические члены класса необходимо после определения класса

Можно определять не только *статические поля*, но и *статические методы* класса. Для вызова статических методов не нужно создавать объект класса, их можно вызывать с использованием имени класса

# Пример: счетчик объектов

```
class Matrix {  
    const int dim;  
    double *data;  
    static int count;  
    int id;  
  
public:  
    Matrix(int n) : dim(n) {  
        data = new double[dim*dim];  
        id = count;  
        ++count;  
    }  
    static int get_count() {return count;}  
    int get_id() const {return id;}  
};
```

```
int Matrix::count = 0;  
  
int main() {  
    Matrix A(3);  
    Matrix B(3);  
  
    cout << A.get_id() << endl; // 0  
    cout << B.get_id() << endl; // 1  
  
    cout << Matrix::get_count()  
        << endl; // 2  
};
```



# Современная работа с динамической памятью

- Идея: поручить работу с динамической памятью объекту, который в конструкторе выделяет память, а в деструкторе ее освобождает (RAII)
- *Умные указатели* реализуют эту идею в C++ (заголовочный файл `<memory>`)

## Устаревающий подход

```
Matrix* M = new Matrix(3);  
Matrix* M2 = M;  
delete M;
```

## Современный подход

```
unique_ptr<Matrix> M1 = make_unique<Matrix>(3);  
// unique_ptr<Matrix> M5 = M1; // error
```

```
shared_ptr<Matrix> M2 = make_shared<Matrix>(3);  
shared_ptr<Matrix> M3 = M2;
```

# Современная работа с динамической памятью

- Идея: поручить работу с динамической памятью объекту, который в конструкторе выделяет память, а в деструкторе ее освобождает (RAII)
- *Умные указатели* реализуют эту идею в C++ (заголовочный файл `<memory>`)

## Устаревший подход

```
Matrix* M = new Matrix(3);  
Matrix* M2 = M;  
delete M;
```

## Современный подход

```
auto M1 = make_unique<Matrix>(3);
```

```
auto M2 = make_shared<Matrix>(3);  
auto M3 = M2;
```

# Суммируем

1. Для работы с динамической памятью в *C++* определены специальные операторы **new**, **new[]**, **delete** и **delete[]**. Умные указатели делают работу с динамической памятью более удобной и безопасной
2. В случае, если при создании объекта используется динамическая память, необходимо определить следующие методы (**правило пяти**): конструкторы присвоения и перемещения, операторы присваивания и перемещения, деструктор
3. Работа с *rvalue*-ссылками и семантика перемещения позволяет избегать лишнего копирования
4. Внутри метода предопределен указатель **this** на объект, для которого вызван метод
5. Для определения полей, общих для всех объектов класса, используется ключевое слово **static**. Также можно определить статические методы класса, не «привязанные» к объекту
6. Для обозначения нулевого указателя в *C++* есть специальное ключевое слово **nullptr**