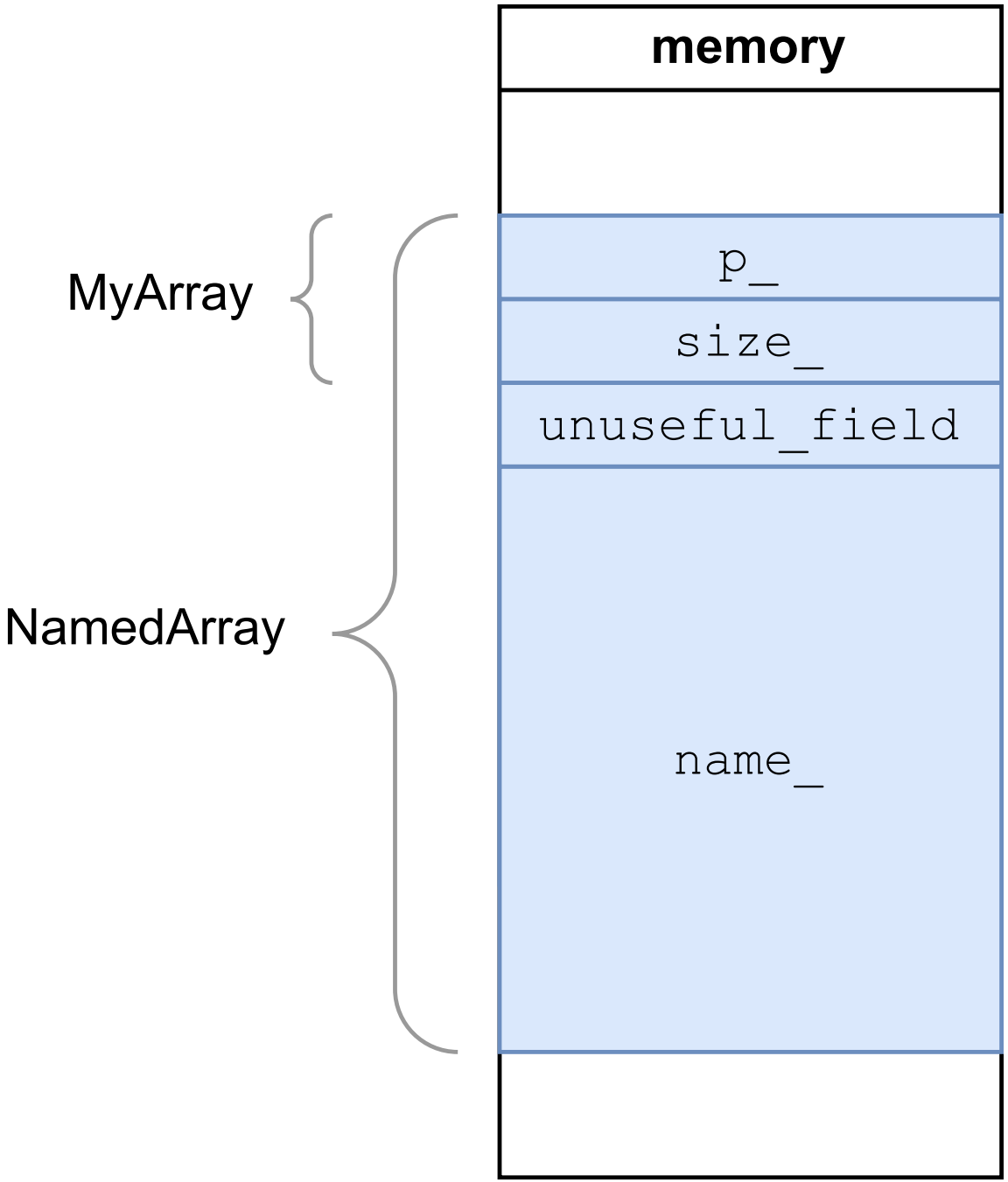
Классы, динамический полиморфизм.

Наследование (не полиморфное)

```
class MyArray{
  protected :
  int * ptr_;
  size_t size_;
  // эти два поля являются
  // инвариантом, то есть они
  // зависят друг от друга
  // изменение одного поля
  // должно делаться с
  // оглядкой на другое
}

class NamedArray: public MyArray{
  protected :
  uint64_t unuseful_field;
  std::string name_;
}
```



Модификаторы доступа (Access specifiers)

- спецификатор членов класса
  - `public : member-declarations` - члены, объявленные после спецификатора доступа, имеют публичный доступ к членам. доступ возможен отовсюду.
  - `protected : member-declarations` - доступ имеют только потомки класса (считай доступ из методов потомков) или друзья класса
  - `private : member-declarations` - доступ возможен только из данного класса или из друзей. Закрытые члены базового класса всегда недоступны производному классу независимо от открытого, защищенного или закрытого наследования.
- базовый спецификатор (определяет доступность унаследованных членов)
  - `public base-class` - не меняет доступ к членам родительского класса
  - `protected base-class` - делает открытые члены базового класса защищенными
  - `private base-class` - делает члены базового класса закрытыми

Модификаторы доступа (Access specifiers)

```
class Base {
public:
    void pub() {std::cout<< "Base\n";}
    void anotherPub() {std::cout<< "Base\n";}
protected:
    void prot() {}
    int i , j;
private:
    void priv() {}

};

class Derived: public Base {
    public:
    void pub() {
        Base::pub(); // OK parent class method call
        anotherPub();  // OK parent class method call since there is no method with the same name
        prot(); // OK
        //priv(); // ошибка
        i = 10;
        Base::i = 20;
        j = 30 ;
    }
    int i;
};


int main(){
    Derived d;
    d.pub();  // Derived::foo call
    d.Base::pub();
    d.anotherPub();
    //d.prot(); // ошибка
    //d.priv(); // ошибка
    return 0;
}
```

Модификаторы доступа (Access specifiers)

```
class AnotherDerived: private Base {
    public:
    using Base::prot;
    void thirdPub () {
        Base::pub(); // OK parent class method call
        anotherPub();  // OK parent class method call
        // since there is no method with the same name
        std::cout<< "Derived\n";
        prot(); // OK
    }
};

class TwiceDerived: public AnotherDerived {
    public:
    void foo () {
        prot() ; // можно вызывать, был открыт в секции public
        // с помощью using
        //anotherPub(); // ошибка, был закрыт в родительско классе
        // базовым спецификатором
    }
};

int main(){
    TwiceDerived d;
    d.prot(); // OK
    // d.anotherPub(); // ошибка
   return 0;
}
```

Определение типов внутри классв

```
class MyClass{
    public:
    class AuxilaryClass{
      public:
      int * begin_;
      int * end_;
      int * begin(){return begin_;}
      int * end(){return end_;}
    };
    void func() {AuxilaryClass a; }
    protected:
    AuxilaryClass c_;
    int * p_ = nullptr;
};

// содание псевдонимов типа можно использовать и внутри тела класса/функции
typedef MyClass::AuxilaryClass NewName; // способ на древнем наречии
using  SomeName =  MyClass::AuxilaryClass ; // способ начиная с++11.
// можно делать шаблонные псевдонимы в отличии от typedef

int main(){
    SomeName a;
    MyClass::AuxilaryClass b;
    NewName c;
    return 0;
}
```

Шаг в сторону. Namespace (useful link)

```
namespace MyNamespace {
    class SomeTypeName {
      public:
      typedef int NestedType;
      NestedType a;
    };
}
using namespace std;
int main(){
    cout<< "Hello world";
    MyNamespace::SomeTypeName a; // имя SomeTypeName находится в пространстве имен   MyNamespace
    MyNamespace::SomeTypeName::NestedType b; // имя NestedType находится в типе SomeTypeName
    {
        using namespace MyNamespace; // будет действовать в пределах блока {}
        SomeTypeName a1;
        SomeTypeName::NestedType b1;
        // using SomeTypeName::NestedType; // так низя
        using  MyPseudonym = SomeTypeName::NestedType ; // будет работать далее в пределах блока
        MyPseudonym b2;
    }
    // SomeTypeName a2; //ошибка. т.к. вне блока
    return 0;
}
```

Промежуточные итоги

- структуры отличаются от классов модификатором доступа по умолчанию.
  для классов это private, для структур public
- модификаторы доступа не влияют на организацию объекта в памяти,
  лишь накладывают ограничения на области видимости. используйте private или
  или хотябы protected для инвариантов класса.
- Если хотим открыть защищенный член класса можно использовать using.
  (немного забегая вперед)если вы наследуете шаблонный класс от шаблонного класса, доступ к членам родительского класса можно упростить с помощью using
- классы и структуры можно объявлять внутри тела другого класса/функции
  но область видимости будет ограничена телом. для доступа к данному классу
  нужно использовать оператор разрешения области `::`
- именам можно создавать псевдонимы используя using или typedef

Список инициализации

```
#include <iostream>
struct Compose {
    Compose(int i): i_(i) { std::cout <<"make Compose "<< i << std::endl; }
    int i_;
    int i2_ ;
};

struct Parent {
    Parent( ):  c1_(1), c_(0) { std::cout <<  "make Parent" << std::endl; }
    Compose c_;
    Compose c1_ = 100;
};

struct Child : Parent{
    Child( ):  c2_(1){  std::cout <<  "make Child" << std::endl; }
    Compose c2_;
};
// Инициализация полей происходитт в порядке обьявления этих полей в теле класса !!!!
// порядок в списке инициализации (то что перед телом конструктора )
// не имеет значения
int main(){Child a; }
// output :
//make Compose 0
//make Compose 1
//make Parent
//make Compose 1
//make Child
```

Список инициализации

```
class Parent{
    public:
    Parent() = default;
    explicit Parent(int val): public_int_(val) { }
    int public_int_ ;
};

class Child: public Parent {
   public:
   Child(int & value, float fl) :Parent(value), immutable_(fl) , ref_(value)
   //, public_int_(10)   error нельзя инициализировать
    { name_ = "some"; // bed practice
    }
   public:
    const float immutable_ ;
   private:
    std::string name_;
    int& ref_;
};
```

Инициализация полей класса

- определяется после заголовка конструктора, перед его телом, через двоеточие :
- члены инициализируются в порядке их объявления в классе, а не в порядке списка инициализации
- присваивание в теле конструктора не оптимально
  даже если вы не использовали поле в списке инициализации,
  будет неявно вызван default constructor (а он может что-то делать),
  потом уже в деле будет выполнено присваивание (двойная работа, а оно нам надо?)
- const поля, ссылки и объекты инициализируем только через список
- в списке может быть вызвать констрруктор родительского класса или будет вызван default
- мы можем вызвать другой конструктор собственного класса (delegating constructors)

Автоматический вызов деструктора родительского класса

```
#include <iostream>
class Parent {
public:
    ~Parent() { std::cout << "Parent destructor\n"; }
};
class Child : public Parent {
public:
    ~Child() { std::cout << "Child destructor\n"; }
};
int main() {
    Child c;
}
```

```
Child destructor
Parent destructor
```

Ключевое слово explicit конструкторы

`explicit` предотвращает неявные преобразования, делая код более безопасным и понятным.

```
class IntVector {
    public: IntVector(size_t num, double def = 0.){/* */};
    // чтобы неявного приведения типов
    // explicit IntVector(size_t num, double def = 0.){/* */};
};

int sumArr(IntVector const& arr) { /*...*/}

int main(){
    Child a;
    sumArr(10); // без explicit произойдет неявное приведение типа
    // будет создан временный объект IntVector с параметром num = 10 и def = 0. .
    //
    // с explicit компилятор выдаст ошибку
}
```

Статические поля класса

- статическое поле (static) – это член класса, который общий для всех объектов данного класса и не привязан к конкретному экземпляру.
- объявляется в классе, но определяется отдельно
- статические поля полезны для хранения глобального состояния, подсчёта объектов и общих конфигурационных параметров.
- доступ вне класса через `ИмяКласса::имя_переменной`

```
\\storage.h
class Storage {
public:
    static std::vector<int> data;
};
\\storage.cpp
std::vector<int> Storage::data = {1, 2, 3};
```

Функции-члены класса (не статические)

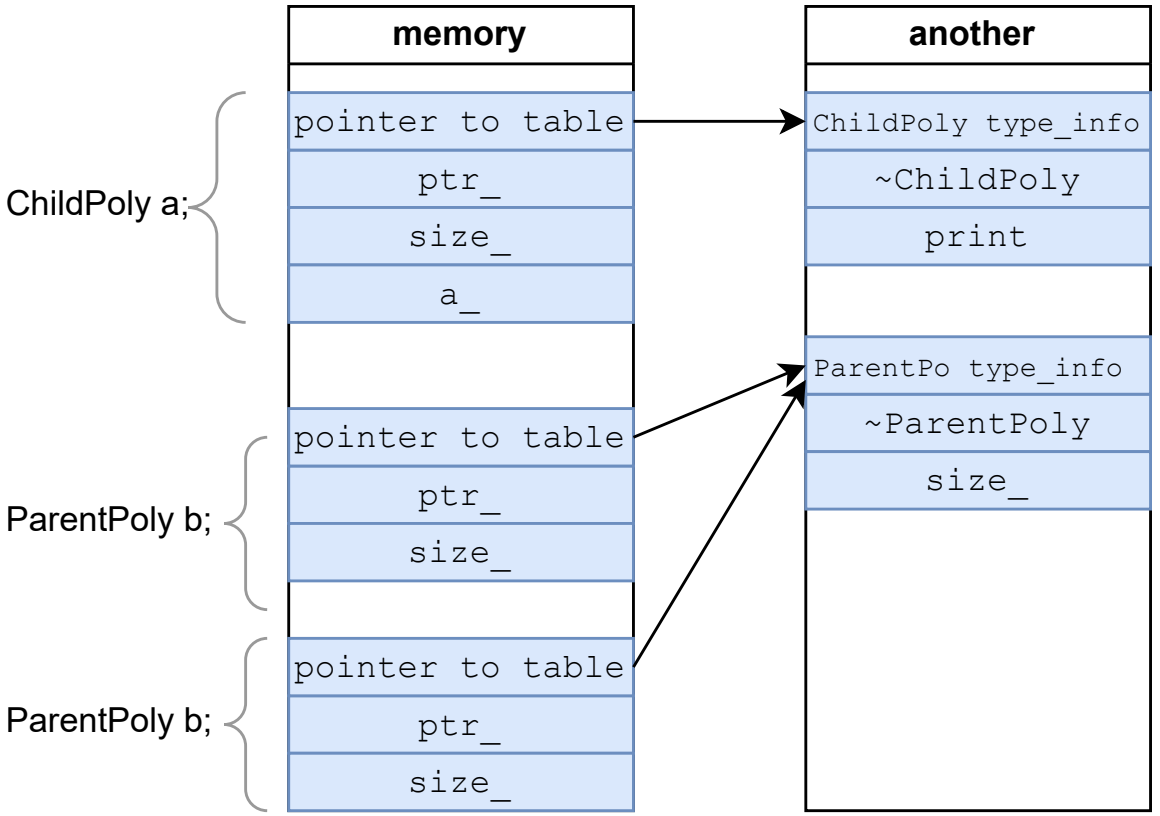
- Определяются внутри или вне класса
- имеют неявный аргумент this (аноалог self в python)
- Могут быть const если не меняют объект (компилятор делает проверку)
- если определены в классе то неявно используется inline
  (копилятор пытается встроить ее без вызова,
  что в случае маленьких функций сокращает накладные расходы)

```
class MyClass {
public:
    void print() const; \\ OK
    void set(int *ptr) const  { ptr_ = ptr; } \\ приведет к ошибке компиляции
private:
  int * ptr_ = nullptr;
};
void MyClass::print() const {
    std::cout << "Ptr value "<< ptr_ << std::endl;
}
```

Полиморфные классы

```
#include <iostream>
struct ParentPoly {
    virtual ~ParentPoly() {
      std::cout << "Parent destructor\n";
    };
    virtual void print() {
      std::cout << "Parent\n";
    }
  private:
  int* ptr_ = nullptr;
  int size_ = 0;
};

struct ChildPoly : public ParentPoly {
    ~ChildPoly() override  {
      std::cout << "Child destructor\n"; };
    void print() override {
      std::cout << "Child\n"; }
    uint64_t a_ ;
};
```



Полиморфные классы

```
//Если базовый класс предназначен для полиморфного использования,
// его деструктор должен быть виртуальным,
// иначе могут возникнуть утечки памяти при удалении через указатель на базовый класс:
#include <iostream>
struct ParentPoly {
    virtual ~ParentPoly() {  std::cout << "Parent destructor\n"; };
    virtual void print() { std::cout << "Parent\n"; }
};

struct ChildPoly : public ParentPoly {
    ~ChildPoly() override  { std::cout << "Child destructor\n"; };
    void print() override { std::cout << "Child\n"; }
};

void show(ParentPoly* ptr){ ptr->print();}

int main() {
    ParentPoly* a = new ParentPoly ;
    ParentPoly* b = new ChildPoly ;
    show(a); show(b);  // Output:
    // Parent
    return 0 ;
}
```

```cpp
// NVI example
class Animal {
  public:
   void speak() const { std::cout << getSound() << std::endl; }
  private:
   virtual std::string getSound() const = 0;
};

class Dog : public Animal {
  private:
   std::string getSound() const { return "Woof!"; }
};

class Cat : public Animal {
  private:
   std::string getSound() const { return "Meow!"; }
};
```

Статические Функции-члены класса

- Принадлежат классу, а не объекту.
- Не имеют доступа к нестатическим членам (без передачи объекта).
- Вызываются через класс (ClassName::method()), но могут вызываться и через объект.

```
class MyClass {
public:
  static void doSomething() ;
};
void MyClass::doSomething() {
}

int main(){
  MyClass::doSomething();
  return 0;
}
```