# 16. Конструкторы. Назначение конструкторов. Список инициализаторов конструктора. Делегирующий конструктор. Деструктор. Порядок вызова конструктора и деструктора при наследовании.

* **Конструкторы. Назначение конструкторов (инфа – с metanit.com).**

Конструкторы представляют собой специальный метод, который имеет то же имя, что и класс, которая не возвращает никакого значения и которая позволяют инициализировать объект класса во время его создания и таким образом гарантировать, что поля класса будут иметь определенные значения. При каждом создании нового объекта класса вызывается конструктор класса.

Конструктор может быть либо автоматически создан компилятором (не принимает никаких параметров и, по сути, ничего не делает), т.е. задан неявно, либо же пользователь может самостоятельно определить его явно:

class Person

{

public:

std::string name;

unsigned age;

void print()

{

std::cout << "Name: " << name << "\tAge: " << age << std::endl;

}

Person(std::string default\_name = “Unknown”, unsigned default\_age = 18)

{

name = default\_name;

age = default\_age;

std::cout << "Person has been created" << std::endl;

}

};

В данном примере мы делаем так, чтоб в классе Person при создании объекта поля name и age уже имели некоторые значения по умолчанию (default\_name и default\_age в нашем случае).

Назначение конструкторов:

· Инициализация членов класса.

· Выделение ресурсов, таких как память.

· Установка начальных значений для объекта (см. пример выше).

* **Список инициализаторов конструктора.**

Список инициализаторов конструктора позволяет инициализировать члены класса до входа в тело конструктора. Он записывается после заголовка конструктора, разделенного двоеточием.

class Person {

private:

std::string name;

unsigned age;

public:

MyClass(std::string default\_name, unsigned default\_age) : name(default\_name), age(default\_age) {}

};

В данном примере список инициализаторов конструктора «: name(default\_name), age(default\_age)» инициализирует члены класса «name» и «age» значениями «default\_name» и «default\_age» соответственно. Это более эффективный способ инициализации членов класса, чем назначение значений в теле конструктора (как в самом первом примере).

* **Делегирующий конструктор.**

Делегирующий конструктор позволяет вызывать другой конструктор того же класса для выполнения общей работы по инициализации. Он используется для предоставления различных способов инициализации объектов.

class 3D\_BoxModel {

private:

float length;

float width;

float height;

public:

MyClass(float box\_length, float box\_width) : MyClass(box\_length, box\_width, 0.0) {}

MyClass(float box\_length, float box\_width, float box\_height) : length(box\_length), width(box\_width), height(box\_height) {}

};

В данном примере в классе 3D\_BoxModel (3D модель прямоугольной коробки) реализован делегирующий конструктор, который, если пользователь при вызове конструктора укажет два параметра вместо трёх [т.е. пропишет в main «**3D\_BoxModel Box(1.3, 2.0);**»], автоматически выдаёт объекту третье значение как 0, т.е. генерирует как бы «картонку», т.е. двумерный объект на основе класса трёхмерного объекта.

Иначе говоря, делегирующий конструктор просто выдаёт объекту недостающие значения.

* **Деструктор (инфа – с metanit.com).**

Деструктор – специальный метод, выполняющий освобождение использованных объектом ресурсов и удаление нестатических переменных объекта. Деструктор автоматически вызывается, когда удаляется объект. Удаление объекта происходит в следующих случаях:

* когда завершается выполнение области видимости, внутри которой определены объекты;
* когда удаляется контейнер (например, массив), который содержит объекты;
* когда удаляется объект, в котором определены переменные, представляющие другие объекты;
* динамически созданные объекты удаляются при применении к указателю на объект оператора delete.

Деструктор называется по имени класса (как и конструктор), перед которым добавляется тильда (~):

#include <iostream>

class Person

{

public:

Person(std::string p\_name)

{

name = p\_name;

++count;

std::cout << "Person " << name << " created. Count: " << count << std::endl;

}

~Person()

{

--count;

std::cout << "Person " << name << " deleted. Count: " << count << std::endl;

}

private:

std::string name;

static inline unsigned count{}; // счётчик объектов

};

int main()

{

{

Person tom{"Tom"};

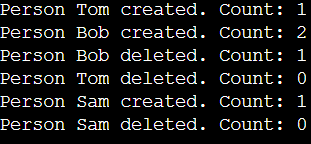
Person bob{"Bob"};

} // <- здесь объекты Tom и Bob уничтожаются, т.к. завершается выполнение области видимости;

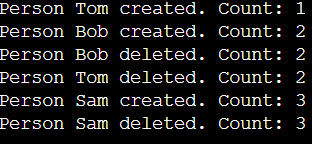
Person sam{"Sam"};

} // <- здесь объект Sam уничтожается по той же причине.

В этом примере у нас создаётся счётчик объектов Person (переменная count). В теле конструктора мы увеличиваем этот счётчик, т.к. объект создаётся, в теле деструктора же уменьшаем его. Зачем это делается? Посмотрим консольный вывод программы:



Если бы мы не уменьшали счётчик count в деструкторе, вывод был бы следующим:



Т.е. логика серьёзно нарушена, т.к. в конце счётчик равен 3-м, хотя программа совершила удаление всех объектов, и он должен быть равен нулю.

Также, для любого класса компилятор, по аналогии с конструктором по умолчанию, создаёт и «пустой» деструктор по умолчанию: **~<имя класса>(){}**.

* **Порядок вызова конструктора и деструктора при наследовании (инфа с bestprog.net).**

Если два класса образуют иерархию наследования, то при создании экземпляра производного класса сначала вызывается конструктор базового класса. Затем этот конструктор становится недоступен и дополняется кодом конструктора производного класса.

Таким образом, сначала происходит инициализация данных базового класса, затем инициализация данных производного класса.

Если классы образуют иерархию, деструкторы этих классов вызываются в обратном порядке по отношению к вызову конструкторов. Сначала вызывается деструктор производного класса, затем вызывается деструктор базового класса.

Пример:

#include <iostream>

using namespace std;

class A

{

public:

A()

{

cout << "Constructor A::A()" << endl;

}

~A()

{

cout << "Destructor A::~A()" << endl;

}

};

class B : public A

{

public:

B()

{

cout << "Constructor B::B()" << endl;

}

~B()

{

cout << "Destructor B::~B()" << endl;

}

};

class C :public B

{

public:

C()

{

cout << "Constructor C::C()" << endl;

}

~C()

{

cout << "Destructor C::~C()" << endl;

}

};

int main()

{

C obj; // A() => B() => C()

} // ~C() => ~B() => ~A()

Консольный вывод:

