Конструктор и деструктор

Конструктор

- Представляет собой специальную функцию, которую ваша программа автоматически вызывает каждый раз при создании объекта
- Имеет такое же имя, как и класс объекта
- Не имеет возвращаемого значения, но вы не указываете ему тип void. Вместо этого вы просто не указываете возвращаемое значение вообще

Конструктор

Когда ваша программа создает объект, она может передать параметры конструктору во время объявления объекта.

Конструктор

C++ позволяет вам перегружать конструкторы и разрешает использовать значения по умолчанию для параметров.

Деструктор

- Представляет собой специальную функцию, которую ваша программа вызывает автоматически каждый раз при уничтожении объекта
- Имеет такое же имя, как и класс объекта но его имя предваряется символом тильды (~)

Объявление и реализация конструктора

У класса Human есть конструктор, который объявляется так:

```
class Human
{
public:
    Human(); // объявление конструктора
};
```

Объявление и реализация конструктора

Конструктор может быть реализован в классе или вне объявления класса.

Объявление и реализация конструктора

Реализация (определение) в классе выглядит следующим образом:

```
class Human
{
public:
    Human()
    {
        // код конструктора здесь
    }
};
```

Объявление и реализация конструктора

Вариант определения конструктора вне объявления класса выглядит следующим образом:

```
class Human
{
public:
    Human(); // объявление конструктора
};
// определение конструктора (реализация)
Human::Human()
{
    // код конструктора здесь
}
```

Объявление и реализация конструктора

Конструктор всегда вызывается при создании объекта.
Это делает конструктор наилучшим местом для инициализации исходными значениями переменных-членов класса, таких как целые числа, указатели и т.д.

Использование конструктора для инициализации переменных-членов класса

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     // Закрытые данные-члены:
 8:     string Name;
 9:     int Age;
10:
```

Использование конструктора для инициализации переменных-членов класса

```
11: public:
12:     // конструктор
13:     Human()
14:     {
15:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
16:         cout << "Constructed an instance of class Human" << endl;
17:     }
```

Использование конструктора для инициализации переменных-членов класса

```
18:
19:     void SetName (string HumansName)
20:     {
21:         Name = HumansName;
22:     }
23:
24:     void SetAge(int HumansAge)
25:     {
26:         Age = HumansAge;
27:     }
28:
29:     void IntroduceSelf()
```

Использование конструктора для инициализации переменных-членов класса

```
    {
        cout << "I am " + Name << " and am ";
        cout << Age << " years old" << endl;
    }
};

int main()
{
    Human FirstMan;
    FirstMan.SetName("Adam");
    FirstMan.SetAge(30);

    Human FirstWoman;
    FirstWoman.SetName("Eve");
    FirstWoman.SetAge (28);

    FirstMan.IntroduceSelf();
    FirstWoman.IntroduceSelf();
}
```

Использование конструктора для инициализации переменных-членов класса

Результат

```
Constructed an instance of class Human
Constructed an instance of class Human
I am Adam and am 30 years old
I am Eve and am 28 years old
```

Конструктор

- Конструктор, который может быть вызван без аргумента, называется *стандартным конструктором* (*default constructor*).
- Собственноручно создавать стандартный конструктор необязательно
- Если вы не создали конструктор сами, то компилятор предоставит его автоматически

Перегрузка конструкторов

Поскольку конструкторы могут быть перегружены как функции, вполне можно получить конструктор, позволяющий создать объект класса Human, с именем в качестве параметра, например:

Перегрузка конструкторов

```
class Human
{
public:
    Human()
    {
        // здесь код стандартного конструктора
    }

    Human(string HumansName)
    {
        // здесь код перегруженного конструктора
    }
};
```

Класс **Human** с несколькими конструкторами

Продемонстрирует применение перегруженных конструкторов при создании объекта класса Human с предоставленным именем:

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:      // Закрытые данные-члены:
 8:      string Name;
 9:      int Age;
10:
```

Класс **Human** с несколькими конструкторами

```
11: public:
12:     // конструктор
13:     Human()
14:     {
15:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
16:         cout << "Default constructor creates an instance of Human"
                 << endl;
17:     }
18:
19:     // перегруженный конструктор, получающий Name
20:     Human(string HumansName)
21:     {
22:         Name = HumansName;
23:         Age = 0; // инициализация гарантирует отсутствие
                     // случайного значения
24:         cout << "Overloaded constructor creates " << Name << endl;
25:     }
```

Класс **Human** с несколькими конструкторами

```
    // перегруженный конструктор, получающий Name и Age
    Human(string HumansName, int HumansAge)
    {
        Name = HumansName;
        Age = HumansAge;
        cout << "Overloaded constructor creates ";
        cout << Name << " of " << Age << " years" << endl;
    }

    void SetName (string HumansName)
    {
        Name = HumansName;
    }

    void SetAge(int HumansAge)
    {
        Age = HumansAge;
    }

    void IntroduceSelf()
    {
        cout << "I am " + Name << " and am ";
        cout << Age << " years old" << endl;
    }
};
```

Класс **Human** с несколькими конструкторами

```
53: int main()
54: {
55:     Human FirstMan; // использование стандартного конструктора
56:     FirstMan.SetName("Adam");
57:     FirstMan.SetAge(30);
58:
59:     Human FirstWoman ("Eve"); // использование перегруженного
                                  // конструктора
60:     FirstWoman.SetAge (28);
61:
62:     Human FirstChild ("Rose", 1);
63:
64:     FirstMan.IntroduceSelf();
65:     FirstWoman.IntroduceSelf();
66:     FirstChild.IntroduceSelf();
67: }
```

Класс **Human** с несколькими конструкторами

Результат

```
Default constructor creates an instance of Human
Overloaded constructor creates Eve
Overloaded constructor creates Rose of 1 years
I am Adam and am 30 years old
I am Eve and am 28 years old
I am Rose and am 1 years old
```

Конструктор

- Вы можете решить не реализовать стандартный конструктор, чтобы заставить создавать экземпляры объектов с определенным минимальным набором параметров

Класс без стандартного конструктора

Класс с перегруженным конструктором, но без стандартного конструктора:

```
 0: #include <iostream>
 1: #include <string>
 2: using namespace std;
 3:
 4: class Human
 5: {
 6: private:
 7:     // Закрытые данные-члены:
 8:     string Name;
 9:     int Age;
10:
```

Класс без стандартного конструктора

```
11: public:
12:
13:     // перегруженный конструктор (без стандартного конструктора)
14:     Human(string HumansName, int HumansAge)
15:     {
16:         Name = HumansName;
17:         Age = HumansAge;
18:         cout << "Overloaded constructor creates " << Name;
19:         cout << " of age " << Age << endl;
20:     }
21:
```

Класс без стандартного конструктора

```
22:     void IntroduceSelf()
23:     {
24:         cout << "I am " + Name << " and am ";
25:         cout << Age << " years old" << endl;
26:     }
27: };
28:
29: int main()
    {
        // Закомментирована следующая строка, пытающаяся создать объект
        // с использованием стандартного конструктора
        // Human FirstMan;

        Human FirstMan("Adam", 30);
        Human FirstWoman("Eve", 28);

        FirstMan.IntroduceSelf();
        FirstWoman.IntroduceSelf();
    }
```

Класс без стандартного конструктора

Результат

```
Overloaded constructor creates Adam of age 30
Overloaded constructor creates Eve of age 28
I am Adam and am 30 years old
I am Eve and am 28 years old
```

Параметры конструктора со значениями по умолчанию

Как и функции, конструкторы способны иметь параметры со значениями, определенными по умолчанию .

В следующем коде приведена немного модифицированная версия конструктора из строки 14 предыдущего листинга, но где у параметра Age есть значение по умолчанию 25:

Параметры конструктора со значениями по умолчанию

```
class Human
{
private:
    // Закрытые данные-члены:
    string Name;
    int Age;

public:
    // перегруженный конструктор (без стандартного конструктора)
    Human(string HumansName, int HumansAge = 25)
    {
        Name = HumansName;
        Age = HumansAge;
        cout << "Overloaded constructor creates " << Name;
        cout << " of age " << Age << endl;
    }
    // ... другие члены
};
```

Параметры конструктора со значениями по умолчанию

Объект такого класса может быть создан следующим образом:

```
Human Adam("Adam"); // Adam.Age присваивается значение по умолчанию 25
Human Eve("Eve, 18); // Eve.Age присваивается указанное значение 18
```

Конструктор

Стандартный конструктор - это тот, который позволяет создавать экземпляры без аргументов, причем не обязательно тот, который не получает параметров.

Ниже приведен конструктор с двумя параметрами, но оба они со значениями по умолчанию, поэтому он является стандартным конструктором:

```
class Human
{
private:
    // Закрытые данные-члены:
    string Name;
    int Age;

public:
    // Обратите внимание на значения по умолчанию для двух
входных параметров
    Human(string HumansName = "Adam", int HumansAge = 25)
    {
        Name = HumansName;
        Age = HumansAge;
        cout << "Overloaded constructor creates " << Name;
        cout << " of age " << Age << endl;
    }
};
```

Дело в том, что экземпляр класса Human вполне может быть создан без аргументов:

```
Human Adam; // Human со значениями по умолчанию Name "Adam", Age 25
```

Конструкторы со списками инициализации

Вы уже видели, насколько полезны конструкторы при инициализации переменных.

Другой способ инициализации членов — использование *списков инициализации (initialization list*).

Ниже показан вариант конструктора, получающего два параметра, но выглядящего со списком инициализации следующим образом:

Конструкторы со списками инициализации

```
class Human
{
private:
    string Name;
    int Age;

public:
    // конструктор получает два параметра для инициализации
    // членов Age и Name
    Human(string InputName, int InputAge)
        :Name(InputName), Age(InputAge)
    {
        cout << "Constructed a Human called " << Name;
        cout << ", " << Age << " years old" << endl;
    }
    // ... другие члены класса
};
```

Конструкторы со списками инициализации

Таким образом, список инициализации характеризуется двоеточием (:) с последующим объявлением параметров, содержащихся в круглых скобках (. . .) , индивидуальными переменными-членами и значениями для инициализации.

Это инициализирующее значение может быть параметром, таким как InputName, или даже фиксированным значением.

Списки инициализации могут также пригодиться при вызове конструкторов базового класса с определенными аргументами.

Стандартный конструктор, способный получать параметры, но со значениями по умолчанию и списком инициализации для установки значений членов:

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

class Human
{
private:
    int Age;
    string Name;

public:
    Human(string InputName = "Adam", int InputAge = 25)
        :Name(InputName), Age(InputAge)
    {
        cout << "Constructed a Human called " << Name;
        cout << ", " << Age << " years old" << endl;
```

Стандартный конструктор, способный получать параметры, но со значениями по умолчанию и списком инициализации для установки значений членов:

```
16:     }
17: };
18:
19: int main()
20: {
21:     Human FirstMan;
22:     Human FirstWoman("Eve", 18);
23:
24:     return 0;
25: }
```

Результат

```
Constructed a Human called Adam, 25 years old
Constructed a Human called Eve, 18 years old
```

Деструктор

Деструкторы (destructor), как и конструкторы, являются специальными функциями.
В отличие от конструкторов, деструкторы автоматически вызываются при удалении объекта.

Объявление и реализация деструктора

Имя деструктора, подобно конструктору, совпадает с именем класса, но предваряется тильдой (~).

Так, у класса Human может быть деструктор, объявленный следующим образом:

```
class Human
{
    ~Human(); // объявление деструктора
};
```

Объявление и реализация деструктора

Этот деструктор может быть реализован в объявлении класса или вне его.

Реализация или определение в классе выглядит следующим образом:

```
class Human
{
public:
     ~Human()
     {
          // здесь код деструктора
     }
};
```

Объявление и реализация деструктора

Определение деструктора вне объявления класса выглядит следующим образом:

```
class Human
{
public:
    ~Human();// объявление деструктора
};

// определение деструктора (реализация)
Human::~Human()
{
    // здесь код деструктора
}
```

Когда и как использовать деструкторы

Деструкторы всегда вызываются при выходе объекта класса из области видимости или при их удалении оператором d e l e t e .

Это делает деструкторы идеальным местом для сброса переменных, а также освобождения зарезервированной динамической памяти и других ресурсов.

Когда и как использовать деструкторы

Рекомендуется применять строки класса

s t d : : s t r i n g , а не символьные буфера в стиле С, где распределением, управлением и освобождением памяти придется заниматься самостоятельно.

Класс s t d : : s t r i n g и другие подобные классы обладают не только конструкторами и деструкторами, но и массой полезных утилит.

Проанализируем пример класса MyString, представленный в следующем листинге, который резервирует память для строки в конструкторе и освобождает ее в деструкторе:

Пример класса, инкапсулирующего буфер в стиле C для гарантии его освобождения при помощи деструктора

```
 0: #include <iostream>
 1: #include <string.h>
 2: using namespace std;
 3: class MyString
 4: {
 5: private:
 6:     char* buffer;
 7:
 8: public:
 9:     MyString(const char* initString)  // constructor
10:     {
11:         if(initString != NULL)
12:         {
13:             buffer = new char [strlen(initString) + 1];
14:             strcpy(buffer, initString);
15:         }
16:         else
17:             buffer = NULL;
18:     }
19:
```

Пример класса, инкапсулирующего буфер в стиле C для гарантии его освобождения при помощи деструктора

```
20:     // Деструктор: освобождает буфер, зарезервированный в
        // конструкторе
21:     ~MyString()
22:     {
23:         cout << "Invoking destructor, clearing up" << endl;
24:         if (Buffer != NULL)
25:             delete [] Buffer;
26:     }
27:
28:     int GetLength()
29:     {
30:         return strlen(Buffer);
31:     }
32:
33:     const char* GetString()
34:     {
35:         return Buffer;
36:     }
37: }; // конец класса MyString
```

Пример класса, инкапсулирующего буфер в стиле C для гарантии его освобождения при помощи деструктора

```
38:
39: int main()
40: {
41:     MyString SayHello("Hello from String Class");
42:     cout << "String buffer in MyString is " << SayHello.GetLength(
43:     cout << " characters long" << endl;
44:
45:     cout << "Buffer contains: ";
46:     cout << "Buffer contains: " << SayHello.GetString() << endl;
47: }
```

Результат

```
String buffer in MyString is 23 characters long
Buffer contains: Hello from String Class
Invoking destructor, clearing up
```