КОНСТРУИРОВАНИЕ ПРОГРАММ

Лекция № 18 — Конструкторы

Преподаватель: Поденок Леонид Петрович, 505a-5

+375 17 293 8039 (505a-5)

+375 17 320 7402 (ОИПИ НАНБ)

prep@lsi.bas-net.by

ftp://student:2ok*uK2@Rwox@lsi.bas-net.by/

Кафедра ЭВМ, 2021

2021.11.24

Оглавление

Классы

Класс — это тип.

Классы представляют собой расширенную концепцию структур данных.

Подобно структурам данных, они могут содержать элементы данных, но они также могут содержать функции в качестве членов.

Объект — это экземпляр класса.

С точки зрения переменных класс будет типом, а объект — переменной.

Классы определяются с использованием класса ключевых слов или структуры ключевых слов со следующим синтаксисом:

```
class имя_класса {
access_specifier_1:
    member1;
access_specifier_2:
    member2;
  ...
} имя_объекта;
```

имя_класса является допустимым идентификатором класса.

имя_объекта является необязательным списком имен для объектов этого класса.

member — тело объявления может содержать члены, которые могут быть либо объявлениями данных, либо объявлениями функций, и (необязательно) спецификаторами доступа.

Классы имеют тот же формат, что и простые структуры данных, за исключением того, что они могут также включать функции и иметь дополнительные особенности, называемые спецификаторами доступа.

Спецификатор доступа — это одно из следующих трех ключевых слов:

private
public
protected

Спецификаторы доступа изменяют права доступа для следующих за ними членов.

private — частные/закрытые члены класса доступны только для и из других членов того же класса (или от их «друзей (*friend*)»).

protected — защищенные члены доступны для других членов того же класса (или от их «друзей»), но также и для членов их производных классов.

public — открытые члены доступны из любого места, где виден объект.

По умолчанию все члены класса, объявленные с ключевым словом **class**, имеют частный доступ (**private**) ко всем его членам.

Таким образом, любой член, который объявлен перед любым другим спецификатором доступа, имеет частный доступ автоматически. Например:

```
class Rectangle {
    int width;
    int height;
public:
    void set_values(int, int);
    int area(void);
} rect;
```

Объявляет класс (то есть. тип) с именем **Rectangle** и объект (то есть переменную) этого класса, называемый **rect**.

```
class Rectangle {
    int width;
    int height;
public:
    void set_values(int,int);
    int area(void);
} rect;
```

Этот класс содержит четыре члена:

- два члена данных типа **int** (члены **width** и **height**) с закрытым доступом (поскольку закрытый уровень доступа является уровнем доступа по умолчанию);

- две функции-члена с открытым доступом — функции **set_values** и **area**, из которых на данный момент включена только их декларация (объявление), но не определение.

Следует обратить внимание на разницу между именем класса и именем объекта:

Rectangle — имя класса (т.е. имя типа),а **rect** — имя объекта типа **Rectangle**.

Это точно то же самое, что и для :

```
int a;
```

где **int** — это имя типа (класс), а **a** — имя переменной (объект).

После объявления **Rectangle** и **rect** можно получить доступ к любому из открытых членов объекта **rect**, как если бы они были обычными функциями или обычными переменными, просто вставив точку (**.**) между именем объекта и именем члена.

Здесь тот же синтаксис, что и доступ к членам простых структур данных.

Например, так:

```
rect.set_values (3,4);
myarea = rect.area();
```

Единственные члены объекта **rect**, к которым нельзя получить доступ извне класса, — это **width** и **height**, поскольку они имеют закрытый (**private**) доступ и на них можно ссылаться только из других членов этого же класса.

Полный пример класса Rectangle:

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

class Rectangle {
    int width, height;
public:
    void set_values(int, int);
    int area() {return width * height;}
};

void Rectangle::set_values(int x, int y) {
    width  = x;
    height = y;
}

int main () {
    Rectangle rect;
    rect.set_values(3,4);
    cout << "area: " << rect.area();
    return 0;
}
```

В этом примере вновь вводится оператор области видимости (`::`, два двоеточия), который рассматривался ранее в отношении пространств имен.

Оператор области видимости используется в определении функции **set_values** для определения функции-члена класса вне самого класса.

Следует обратить внимание, что, учитывая предельную простоту функции-члена **area**, ее определение включено непосредственно в определение класса **Rectangle**.

set_values(int, int) — это просто объявление прототипа функции-члена внутри класса, но его определение находится за его пределами.

В этом внешнем определении для указания того, что определяемая функция является членом класса **Rectangle**, а не обычной функцией, не являющейся членом класса, используется оператор разрешения области видимости (**::**).

В данном случае оператор области (**::**) указывает класс, к которому принадлежит определяемый член, предоставляя точно такие же свойства области видимости/действия, как если бы это определение функции было непосредственно включено в определение класса.

Например, функция **set_values** имеет доступ к переменным **width** и **height**, которые являются закрытыми членами класса **Rectangle** и, таким образом, доступны только из других членов класса, таких как **this**.

Единственная разница между полным определением функции-члена в определении класса или просто включением ее объявленияи последующим определением вне класса заключается в том, что в первом случае компилятор автоматически считает эту функцию встроенной (*inline*) функцией-членом, тогда как во втором это обычная (не встроенная) функция-член класса.

Это не вызывает никаких различий в поведении, но позволяет, возможно, компилятору сделать какие-нибудь оптимизации.

Члены **width** и **height** имеют закрытый (частный) доступ (если ничего не указано, все члены класса, определенного с помощью ключевого слова **class**, имеют закрытый доступ). Объявляя их закрытыми, запрещается доступ извне класса (на уровне языка).

Закрытие их имеет смысл, поскольку определена функция-член для установки значений этих элементов в объекте — функция-член **set_values**. Поэтому остальная часть программы прямой доступ к ним иметь не должна.

В таком простом примере, как этот, трудно понять, насколько полезным может быть ограничение доступа к этим переменным, но в больших проектах может быть очень важно, чтобы значения не могли быть изменены неожиданным образом (неожиданно с точки зрения объекта).

Наиболее важным свойством класса является то, что он является типом, и поэтому мы можем объявить несколько объектов этого класса. Например, следуя примеру, где определен класс **Rectangle**, мы можем в дополнение к объекту **rect** объявить объект **rectb**:

Пример — один класс, два объекта

```
#include <iostream>

class Rectangle {
    int width, height;
public:
    void set_values(int, int);
    int area() {return width * height;}
};

void Rectangle::set_values(int x, int y) {
    width  = x;
    height = y;
}

int main () {

    Rectangle recta, rectb;

    recta.set_values(3,4);
    rectb.set_values(5,6);
    std::cout << "recta area: " << recta.area() << std::endl;
    std::cout << "rectb area: " << rectb.area() << std::endl;
    return 0;
}
```

В данном конкретном случае классом (типом объектов) является **Rectangle**.

Также у нас есть два экземпляра этого класса (т.е. объекты) — **rect** и **rectb**. Каждый из них имеет свои собственные переменные-члены и функции-члены.

Следует обратить внимание, что вызов **rect.area()** не дает того же результата, что вызов **rectb.area()**, потому, что каждый объект класса **Rectangle** имеет свои собственные переменные **width** и **height.**

В то же время их собственные функции-члены **set_value** и **area**, которые работают с собственными переменными-членами объектов, являются общими для всех объектов.

Классы позволяют программировать с использованием объектно-ориентированных парадигм — данные и функции являются членами объекта, что уменьшает необходимость передавать и переносить обработчики или другие переменные состояния в качестве аргументов функций, поскольку они являются частью объекта, функция-член которого вызывается.

Следует обратить внимание, что при вызовах **rect.area** или **rectb.area** этим функциям не было передано никаких аргументов. Эти функции-члены напрямую используют элементы данных своих соответствующих объектов **rect** и **rectb**.

Что произойдет в предыдущем примере, если мы вызовем функцию-член **area** до вызова **set_values**?

Результатом будет «неопределенный результат», так как элементам **width** и **height** никогда не были присвоены значения.

Чтобы избежать этого, класс может включать в себя специальную функцию, называемую его конструктором, которая автоматически вызывается при создании нового объекта этого класса. Такое поведение позволяет классу инициализировать переменные-члены или выделить память.

Конструкторы

Функция-конструктор объявляется как обычная функция-член, но с именем, совпадающим с именем класса и без какого-либо возвращаемого типа, даже не **void**.

Однако, на самом деле конструктор не имеет имени, а его объявление и определение — это просто специальная синтаксическая форма (несколько их).

Приведенный выше класс **Rectangle** можно улучшить, реализовав конструктор:

```
#include <iostream>

class Rectangle {
    int width, height;
public:
    Rectangle(int, int);                          // Конструктор
    int area() {return (width * height);}
};

Rectangle::Rectangle(int a, int b) {
    width  = a;
    height = b;
}

int main () {
    Rectangle recta(3,4);
    Rectangle rectb(5,6);
    std::cout << "recta area: " << recta.area() << std::endl;
    std::cout << "rectb area: " << rectb.area() << std::endl;
    return 0;
}
```

Результирующий вывод данного примера будет идентичен результатам предыдущего примера. Но теперь класс **`Rectangle`** не имеет функции-члена **`set_values`** и вместо этого имеет конструктор, который выполняет аналогичное действие — он инициализирует значения ширины и высоты с помощью переданных ему аргументов.

Следует обратить внимание на способ, как эти аргументы передаются в конструктор в момент создания объектов этого класса:

```
Rectangle recta(3,4);
Rectangle rectb(5,6);
```

Конструкторы нельзя вызывать явно, как если бы они были обычными функциями-членами — они выполняются только один раз, когда создается новый объект этого класса.

Следует обратить внимание, что ни объявление прототипа конструктора (внутри класса), ни определение конструктора позже, не имеют никаких возвращаемых значений, даже **`void`** — **конструкторы никогда не возвращают значения, они просто инициализируют объект.**

Перегрузка конструкторов

Как и любая другая функция, конструктор также может быть перегружен разными версиями, принимающими разные параметры — с разным количеством параметров и/или с параметрами разных типов.

Компилятор автоматически вызовет тот, чьи параметры соответствуют аргументам:

Пример — перегрузка конструктора

```
class Rectangle {
    int width, height;
public:
    Rectangle( );                  // явный конструктор по умолчанию
    Rectangle(int,int);            // конструктор с двумя параметрами
    int area(void) {
        return (width * height);
    }
};

Rectangle::Rectangle( ) {
    width  = 5;
    height = 5;
}

Rectangle::Rectangle(int a, int b) {
    width  = a;
    height = b;
}
```

```
#include <iostream>

int main () {

    Rectangle recta(3,4);
    Rectangle rectb;

    std::cout << "recta area: " << recta.area() << std::endl;
    std::cout << "rectb area: " << rectb.area() << std::endl;

    return 0;
}
```

В приведенном выше примере построены два объекта класса **Rectangle** — **recta** и **rectb**.

Объект **recta** конструируется с двумя аргументами, как и в предыдущем примере.

В этом примере также представлен специальный конструктор вида — конструктор по умолчанию.

Конструктор по умолчанию

Конструктор по умолчанию — это конструктор, который не принимает параметров, и он является в некоторой степени особенным.

Конструктор по умолчанию всегда вызывается при объявлении объекта, если при объявлении параметры не передаются, при этом объект не инициализируется никакими аргументами.

Однако если конструктор класс имеет какие-либо конструкторы с параметрами, конструктор по умолчанию следует объявлять явно, иначе компилятор сообщает об ошибке.

```
class Rectangle {
    int width, height;
public:
    Rectangle(int, int);
    int area() {return width * height;}
};

Rectangle::Rectangle(int x, int y) {
    width  = x;
    height = y;
}

int main () {

    Rectangle rect; // ошибка: no matching function for call to «Rectangle::Rectangle()
    Rectangle rectb(3, 4);

    std::cout << "area: " << rect.area() << std::endl;
    return 0;
}
```

В приведенном ранее примере перегрузки конструктора для **rect** вызывается явный конструктор по умолчанию.

Следует обратить внимание, что **rect** создается без всяких скобок — никакие скобки, в том числе и пустые, для вызова конструктора по умолчанию не могут быть использованы

```
Rectangle rectb;   // ok, вызывается конструктор по умолчания
Rectangle rectc(); // oops, конструктор по умолчанию НЕ вызывается
                   // и объект не создается
```

Это связано с тем, что пустой набор круглых скобок сделал бы из **rectc** объявление функции вместо объявления объекта — это была бы функция, которая не принимает аргументов и возвращает значение типа **Rectangle**.

Однородная (равномерная) инициализация

Способ вызова конструкторов, заключая их аргументы в скобки, как это было показано выше, называется *функциональной формой*. Но конструкторы также могут вызываться с другим синтаксисом.

Во-первых, конструкторы с одним параметром можно вызывать с использованием синтаксиса инициализации переменной (знак равенства, за которым следует аргумент):

```
class_name object_name = initialization_value;
```

В C ++ есть возможность вызова конструкторов с использованием *равномерной инициализации*, которая, по сути, совпадает с функциональной формой, но использует скобки **{ }** вместо скобок **( )**:

```
class_name object_name { value, value, value, ... }
```

Опционально, этот последний синтаксис может содержать знак равенства перед фигурными скобками.

```
class_name object_name = { value, value, value, ... }
```

Вот пример с четырьмя способами создания объектов класса, конструктор которого принимает один параметр:

```
// Однородная инициализация
#include <iostream>

class Circle {
    double radius;
public:
    Circle(double r) { radius = r; }
    double circum( ) {return 2 * radius * 3.14159265;}
};

int main ( ) {

    Circle foo(10.0);    // функциональная форма
    Circle bar = 20.0;   // инициализация присвоением
    Circle baz {30.0};   // однородная форма
    Circle qux = {40.0}; // POD-подобная

    std::cout << "foo's circumference: " << foo.circum( ) << '\n';
    return 0;
}
```

Преимущество однородной инициализации по сравнению с функциональной формой заключается в том, что в отличие от круглых скобок, фигурные скобки не могут быть перепутаны с объявлениями функций и, следовательно, могут использоваться для явного вызова конструкторов по умолчанию:

```
Rectangle rectb;   // вызывается конструктор по умолчанию
Rectangle rectc(); // объявление функции (констр-р по умолчанию не вызывается)
Rectangle rectd{}; // вызывается конструктор по умолчанию
```

Выбор синтаксиса для вызова конструкторов во многом зависит от стиля.

Большая часть существующего кода в настоящее время использует функциональную форму, и некоторые более новые руководства по стилю предлагают выбрать равномерную инициализацию по сравнению с другими, даже при том, что он также имеет свои потенциальные ловушки.

Инициализация членов в конструкторах

Когда конструктор используется для инициализации других членов, эти другие члены могут быть инициализированы напрямую, не прибегая к операторам в его теле. Это делается путем вставки перед телом конструктора двоеточия (:) и списка инициализаций для членов класса. Например, рассмотрим класс со следующим объявлением:

```
class Rectangle {
    int width, height;
public:
    Rectangle(int,int);
    int area() {return width*height;}
};
```

Конструктор для этого класса может быть определен как обычно:

```
Rectangle::Rectangle(int x, int y) { width = x; height = y; }
```

Но он также может быть определен с помощью инициализации членов:

```
Rectangle::Rectangle(int x, int y) : width(x) { height = y; }
```

Или даже так:

```
Rectangle::Rectangle(int x, int y) : width(x), height(y) { }
```

Следует обратить внимание, что в этом последнем случае конструктор ничего не делает, кроме как инициализирует свои члены, поэтому он имеет пустое тело функции.

Пример инициализации членов класса

```
#include <iostream>

class Circle {
    double radius;
public:
    Circle(double r) : radius(r) { }
    double area() {return radius * radius * 3.14159265;}
};

class Cylinder {
    Circle base;
    double height;
public:
    Cylinder(double r, double h) : base(r), height(h) {}
    double volume() {return base.area() * height;}
};

int main () {
  Cylinder foo(10, 20);
  std::cout << "foo's volume: " << foo.volume() << '\n';
  return 0;
}
```

В этом примере класс **Cylinder** имеет объект-член, тип которого является другим классом (базовый тип — **Circle**). Поскольку объекты класса **Circle** могут конструироваться только с параметром, конструктору **Cylinder** необходимо вызвать конструктор **base**, и единственный способ сделать это — с помощью списка инициализации членов.

Эти инициализации также могут использовать однородный синтаксис инициализаторов, используя фигурные скобки **{}** вместо скобок **()**:

```
Cylinder::Cylinder(double r, double h) : base{r}, height{h} { }
```

Специальные члены класса

Специальные функции-члены – это функции-члены, которые неявно определяются как члены класса при определенных обстоятельствах.

Их шесть:

	Функция-член	**Типичная форма для класса C**
1)	Конструктор по-умолчанию	`C::C();`
2)	Деструктор	`C::~C();`
3)	Конструктор копирования	`C::C(const C&);`
4)	Оператор присваивания копированием	`C& operator =(const C&);`
5)	Конструктор перемещения	`C::C(C&&);`
6)	Оператор присваивания перемещением	`C& operator =(C&&);`

Конструктор по умолчанию

```
C::C( );
```

Конструктор по умолчанию – это конструктор, который вызывается при объявлении объектов класса, которые не инициализируются никакими аргументами.

```
class Example {
public:
    int total; // инициализации нет объект -> .bss
    void accumulate(int x) { total += x; }
};
```

компилятор предполагает, что класс **Example** имеет конструктор по умолчанию. Следовательно, объекты этого класса могут быть созданы простым их объявлением без аргументов

```
Example ex;
```

Но как только в классе есть какой-то конструктор, принимающий любое количество явно объявленных параметров, компилятор больше не предоставляет неявный конструктор по умолчанию и больше не позволяет объявлять новые объекты этого класса без аргументов.

Пример

```
class Example2 {
public:
    int total;
    Example2(int initial_value) : total(initial_value) { };
    void accumulate (int x) { total += x; };
};
```

Здесь мы объявили конструктор с параметром типа **int**. Поэтому следующее объявление объекта будет правильным:

```
Example2 ex(100);   // ok: calls constructor
```

Но следующее:

```
Example2 ex;        // not valid: no default constructor
```

будет недействительным, поскольку класс был объявлен с явным конструктором, принимающим один аргумент, который заменяет неявный конструктор по умолчанию, не принимающий ни одного.

Поэтому, если объекты этого класса должны создаваться без аргументов, в классе также должен быть объявлен надлежащий конструктор по умолчанию.

Например:

```
#include <iostream>
#include <string> // std::

class Example3 {
    string data;
public:
    Example3(const string& str) : data(str) {}
    Example3() {}
    const string& content() const {return data;}
};

int main () {

  Example3 foo;                 // без аргументов
  Example3 bar("Example");
  std::cout << "bar's content: " << bar.content() << '\n';
  return 0;
}
```

После компиляции и запуска:

```
bar's content: Example
```

Здесь, класс **Example3** имеет конструктор по умолчанию (т.е. конструктор без параметров), **Example3() {}**, определенный как пустой блок.

Это позволяет создавать объекты класса **Example3** без аргументов, как был объявлен **foo** в этом примере.

Обычно такой конструктор по умолчанию неявно определяется для всех классов, у которых нет других конструкторов, и поэтому явное определение не требуется.

Но в этом случае класс **Example3** имеет другой конструктор:

```
Example3 (const string& str);
```

И когда в классе явно объявлен какой-либо конструктор, неявные конструкторы по умолчанию автоматически не предоставляются.

Деструкторы

```
C::C();  -- конструктор по умолчанию
C::~C(); -- деструктор
```

Деструкторы выполняют противоположную функциональность конструкторов – они отвечают за очистку, необходимую классу, когда заканчивается его время жизни.

Классы, которые мы определили в предыдущих главах, не выделяли никаких ресурсов и, следовательно, не требовали никакой очистки.

Но теперь давайте представим, что класс в последнем примере выделяет динамическую память для хранения строки, которую он имел в качестве члена данных. В этом случае было бы очень полезно иметь функцию, вызываемую автоматически в конце срока службы объекта, которая отвечает за освобождение этой памяти.

Для этого мы используем деструктор.

Деструктор – это функция-член, очень похожая на конструктор по умолчанию. Он не принимает аргументов и не возвращает ничего, даже **`void`**.

Он также использует имя класса в качестве собственного имени, но перед ним стоит знак тильды (**~**).

Пример

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

class T {
    string* ptr;
public:
    // constructors:
    T() : ptr(new string) {} // конструктор по умолчанию
    T(const string& str) : ptr(new string(str)) {}
    // destructor:
    ~T () {delete ptr;}     // деструктор
    // access content:
    const string& content() const {return *ptr;} // возвращает строку
};

int main () {

  T foo;
  T bar("Example");
  // При построении класса T выделяется память для строки, которая позже
  // освобождается деструктором. Деструктор для объекта вызывается в конце
  // его жизни; в случае foo и bar это происходит в конце функции main.

  std::cout << "bar's content: " << bar.content() << '\n';
}
```

Конструктор копирования

```
C::C(const C&);
```

Когда при создании объекта ему в качестве аргумента для инициализации передается именованный объект его собственного типа, для создания копии вызывается его конструктор копирования.

Конструктор копирования – это конструктор, ***первый*** параметр которого относится к типу ссылки на сам класс (возможно, с cv-квалификацией) и который может быть вызван с помощью одного аргумента этого типа.

Например, для класса **T** конструктор копирования может иметь следующую сигнатуру:

```
T::T(const T&);
```

Если класс не имеет ни пользовательских конструкторов копирования, ни перемещения (или присваивания), предоставляется неявный конструктор копирования.

Этот конструктор копирования просто выполняет копирование всех своих собственных членов.

Например, для такого класса, как:

```
class SomeClass {
public:
    int a;
    int b;
    string c;
};
```

неявный конструктор копирования будет определен автоматически.

Предполагаемое для этой функции определение неявного конструктора копирования выполняет *поверхностное* копирование, примерно эквивалентное следующему:

```
SomeClass::SomeClass(const SomeClass& x) : a(x.a), b(x.b), c(x.c) {}
```

Этот конструктор копирования по умолчанию может удовлетворить потребности многих классов.

Но поверхностные копии копируют только сами члены класса, и это, вероятно, не то, что мы ожидаем для классов, подобных классу **T**, который мы определили выше, потому что он содержит указатели, с помощью которых он работает со своей памятью.

```
class T {
    string* ptr;
public:
    // constructors:
    T() : ptr(new string) {}                          // к-тор по умолчанию
    T(const string& str) : ptr(new string(str)) {} // инициализируется
                                                      // строкой <string>
    // destructor:
    ~T() {delete ptr;}
    // access content:
    const string& content() const {return *ptr;} // возвращает строку
};
```

Для этого класса выполнение поверхностного копирования означает, что будет скопировано значение указателя, но не самого содержимого. Это означает, что оба объекта (копия и оригинал) будут совместно использовать один **string** объект (они оба будут указывать на один и тот же объект), и в какой-то момент (при уничтожении) оба объекта будут пытаться удалить один и тот же блок памяти, что скорее всего вызывает сбой программы во время выполнения.

Проблема может быть решена путем определения следующего пользовательского конструктора копирования, который выполняет *глубокое* копирование:

```
// copy constructor: deep copy
#include <iostream>
#include <string>

class T {
    string* ptr;
public:
    T(const string& str) : ptr(new string(str)) {}
    ~T() { delete ptr; }
    // copy constructor:
    T(const T& x) : ptr(new string(x.content())) {}       // (*)
    // функция доступа к содержимому:
    const string& content() const { return *ptr; }
};

int main () {

  T foo("Example");
  T bar(foo);       // или так
  T baz = foo;      // или так

  std::cout << "bar's content: " << bar.content() << '\n';
  std::cout << "baz's content: " << baz.content() << '\n';
  return 0;
}
```

После компиляции и запуска:

```
bar's content: Example
baz's content: Example
```

Глубокая копия, выполняемая этим конструктором копирования, выделяет память для новой строки, которая инициализируется копией исходного объекта. Таким образом, оба объекта (копия и оригинал) имеют разные копии контента, хранящиеся в разных местах.

Копирование присваиванием (copy assignment)

```
C& operator =(const C&);
```

Объекты не только копируются, когда они инициализируются при конструировании – они также могут быть скопированы при любой операции присваивания.

Смотрим разницу:

```
MyClass foo;
MyClass bar(foo);  // инициализация объекта: вызывается конструктор копирования
MyClass baz = foo; // инициализация объекта: вызывается конструктор копирования
foo = bar;         // объект уже инициализирован: вызывается copy assignment
```

Обратите внимание, что **baz** инициализируется при построении с использованием знака равенства, но это не операция присваивания, хотя она и выглядит так.

Объявление объекта не является операцией присваивания, это просто еще один синтаксис для вызова конструкторов с одним аргументом.

Присваивание **foo = bar** является операцией присваивания. Здесь не объявляется ни один объект, а выполняется операция над существующим объектом **foo**.

Оператор присваивания копии является перегрузкой оператора **=**, который принимает значение или ссылку на сам класс в качестве параметра. Возвращаемое значение обычно является ссылкой на ***this** (хотя это не обязательно). Например, для класса **MyClass** присваивание копии может иметь следующую сигнатуру:

```
MyClass& operator =(const MyClass&);
```

Оператор копирования присваиванием является специальной функцией и также определяется неявно, если в классе не определены ни пользовательские функции копирования присваиванием, ни функции перемещения присваиванием, ни конструктор перемещения.

```
class T {
    string* ptr;
public:
    T(const string& str) : ptr(new string(str)) {}
    ~T() { delete ptr; }
    // copy constructor:
    T(const T& x) : ptr(new string(x.content())) {}       // (*)
    // функция доступа к содержимому:
    const string& content() const { return *ptr; }
};
```

Неявная версия выполняет поверхностное копирование, которое подходит для многих классов, но не для классов с указателями на обрабатываемые объекты, как это имеет место в случае класса **T**. В этом случае не только класс подвергается риску удаления указанного объекта дважды, но также присваивание создает утечки памяти, не удаляя объект, существовавший объектом до присваивания. Эти проблемы могут быть решены с помощью копирования при присваивании, которое удаляет предыдущий объект и выполняет глубокое копирование:

```
T& operator =(const T& x) {
  delete ptr;                      // delete currently pointed string
  ptr = new string(x.content());  // allocate space for new string, and copy
  return *this;
}
```

Или даже лучше, поскольку его строковый член не является константой, он может повторно использовать тот же строковый объект:

```
T& operator =(const T& x) {
  *ptr = x.content();
  return *this;
}
```

Конструктор перемещения и перемещение присваивания

```
C::C(C&&);            // конструктор перемещения
C& operator =(C&&);   // перемещение при присваивании
```

Подобно копированию, перемещение также использует значение объекта, чтобы установить значение другого объекта. Но, в отличие от копирования, контент фактически передается от одного объекта (источника) к другому (месту назначения), при этом источник теряет тот контент, который переходит к месту назначения.

Такое перемещение имеет место только в том случае, когда источником значения является безымянный объект.

Безымянные объекты – это объекты, которые носят временный характер, и поэтому им даже не было присвоено имя. Типичными примерами безымянных объектов являются возвращаемые значения функций или приведение типов.

Использование значения временного объекта для инициализации другого объекта или для использования его значения для присваивания на самом деле не требует копии – объект никогда не будет использоваться для чего-либо еще, и, таким образом, его значение может быть перемещено в объект места назначения. В таких случаях вызывается конструктор перемещения и конструктор перемещения присваиванием:

Конструктор перемещения вызывается, когда объект инициализируется в конструкции с использованием безымянного временного объекта.

Аналогично, перемещение присваиванием вызывается, когда существующему объекту присваивается значение безымянного временного объекта.

```
MyClass fn();             // функция, возвращающая объект класса MyClass
MyClass foo;              // конструктор по умолчанию
MyClass bar = foo;        // конструктор копирования
MyClass baz = fn();       // конструктор перемещения (move-constructor)
foo = bar;                // копирование при присваивании
baz = MyClass();          // перемещение при присваивании
```

Значение, возвращаемое **fn**, и значение, созданное с помощью **MyClass**, являются безымянными временными значениями. В этих случаях нет необходимости делать копию – безымянный объект недолговечен и может быть получен другим объектом – это более эффективная операция.

Конструктор перемещения и присвоение перемещением являются членами, которые принимают параметр типа *rvalue* со ссылкой на сам класс:

```
MyClass (MyClass&&);              // move-constructor
MyClass& operator =(MyClass&&);   // move-assignment
```

Ссылка на rvalue указывается типом с двумя амперсандами (&&)

Концепция перемещения наиболее полезна для объектов, управляющих памятью, которую они используют, например, для объектов, под которые выделяется память с помощью **new** и **delete**. В таких объектах копирование и перемещение – это действительно разные операции:

- Копирование из A в B означает, что для B выделяется новая память, а затем весь контент A копируется в эту новую память, выделенную для B.

- Перемещение из A в B означает, что память, уже выделенная для A, передается объекту B без выделения какой-либо новой памати. Это заключается просто в копировании указателя.

Например:

```
// move constructor/assignment
#include <iostream>
#include <string>
using namespace std;

class Example6 {
    string* ptr;
public:
    Example6(const string& str) : ptr(new string(str)) {}
    ~Example6() { delete ptr; }
    // move constructor
    Example6(Example6&& x) : ptr(x.ptr) {x.ptr = nullptr;}
    // move assignment
    Example6& operator= (Example6&& x) {
      delete ptr;
      ptr    = x.ptr;
      x.ptr  = nullptr;
      return *this;
    }
    // access content:
    const string& content() const {return *ptr;}
    // addition:
    Example6 operator+ (const Example6& rhs) {
      return Example6(content() + rhs.content());
    }
};
```

```
int main () {
  Example6 foo("Exam");
  Example6 bar = Example6("ple");   // move-construction

  foo = foo + bar;                  // move-assignment

  cout << "foo's content: " << foo.content() << '\n';
  return 0;
}
```

После компиляции и запуска:

```
foo's content: Example
```

Компиляторы оптимизируют многие случаи, которые формально требуют вызова конструкции перемещения в результате так называемой *оптимизации возвращаемого значения*.

В частности, это имеет место когда значение, возвращаемое функцией, используется для инициализации объекта. В этих случаях конструктор перемещения фактически может никогда не вызваться.

Обратите внимание, что хотя ссылки типа rvalue могут использоваться для типа любого параметра функции, это редко полезно для использования, отличного от конструктора перемещения.

Rvalue-ссылки хитры, и ненужное использование может быть источником ошибок, которые довольно сложно отследить.

Неявные члены

Шесть специальных функций-членов, которые описаны выше, являются членами, неявно объявленными для классов при определенных обстоятельствах:

Функция-член	Объявляются неявно:	Определение по умолчанию
Конструктор по умолчанию	если нет других конструкторов	ничего не делает
Деструктор	если нет деструктора	ничего не делает
Конструктор копирования	если нет конструктора перемещения и оператора присваивания перемещением	копирует все члены
Оператор копирования присваиванием	если нет конструктора перемещения и оператора присваивания перемещением	копирует все члены
Конструктор перемещения	если нет деструкторов, нет конструкторов копирования и перемещения, нет операторов копирования и присваивания перемещением.	перемещает все члены
Оператор перемещения присваиванием	если нет деструкторов, нет конструктора копированием, перемещением и операторов присваивания перемещением	перемещает все члены

Не все специальные функции-члены неявно определяются в одних и тех же случаях.

Это происходит в основном из-за обратной совместимости со структурами C и более ранними версиями C++, а на самом деле некоторые включают устаревшие варианты поведения.

К счастью, каждый класс может явно выбрать, какие из этих членов существуют с определением по умолчанию или которые удаляются — для этого используются ключевые слова **`default`** и **`delete`** соответственно. Синтаксис является одним из следующих:

```
function_declaration = default;
function_declaration = delete;
```

Например:

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

class Rectangle {
    int width, height;
public:
    Rectangle(int x, int y) : width(x), height(y) {}
    Rectangle() = default;
    Rectangle(const Rectangle& other) = delete;
    int area() {return width*height;}
};

int main () {
  Rectangle foo;
  Rectangle bar(10, 20);

  cout << "bar's area: " << bar.area() << '\n';
  return 0;
}
```

Здесь объект класса **Rectangle** может быть сконструирован либо с двумя аргументами типа **int**, либо сконструирован по умолчанию (без аргументов). Однако он не может быть создан из другого объекта **Rectangle**, потому что эта функция была удалена. Следовательно, если предположить объекты из последнего примера, следующий оператор будет недействительным:

```
Rectangle baz(foo);
```

Тем не менее, это можно сделать явно допустимым, определив конструктор копирования следующим образом:

```
Rectangle::Rectangle(const Rectangle& other) = default;
```

который, по существу, эквивалентен следующему:

```
Rectangle::Rectangle(const Rectangle& other) :
           width(other.width), height(other.height) {}
```

Обратите внимание, что ключевое слово **default** не определяет функцию-член, эквивалентную конструктору по умолчанию (т.е. конструктор по умолчанию означает конструктор без параметров), но эквивалентную конструктору, который был бы неявно определен, если бы не был удален.

В целом и для будущей совместимости классам, которые явно определяют один конструктор копирования/перемещения или одно присваивание копированием/перемещением, но не оба одновременно, для других специальных функций-членов, которые они явно не определяют, рекомендуется указывать либо удаление, либо настройку по умолчанию.