Конструктор копирования и оператор присваивания

В C++ (до стандарта C++11) у нового объекта по умолчанию есть возможность быть скопированным или присвоенным. За это отвечает неявно сгенерированный конструктор копирования и оператор присваивания. Иными словами по умолчанию копирование и присваивание разрешены. Поведение по умолчанию заключается в копировании всех атрибутов класса. Часто это не то, что требуется, например просто скопированные указатели на некоторые области памяти могут привести к повторному освобождению памяти. Конструктор копирования используется для инициализации класса путем создания копии необходимого объекта. Оператор присваивания копированием используется для копирования одного класса в другой (существующий) класс.

```
class Coord2D {
    int _x, _y;
public:
    A(): _x(int()) , _y(int()) {}  // конструктор
    ~A(){}  // деструктор

    // A a1;
    // A a2 = a1;
    A(const A& c): _x(c.x), _y(c.y){}  // конструктор копирования

    // A a1;
    // A a2;
    // a2 = a1;
    A& operator=(const A& copied)  // оператор присваивания
    {
        x = copied.x;
        y = copied.y;
        return *this;
    }
};
```

```
class Coord2D {
    int _x, _y;
public:
    Coord2D() {
        _x = 0;
        _y = 0;
        cout << "Coord2D(x = " << _x << ", y = " << _y << ") created"
<< endl;
    }

    Coord2D(int x, int y) {
        _x = x;
        _y = y;
        cout << "Coord2D(x = " << _x << ", y = " << _y << ") created"
<< endl;
    }

    ~Coord2D() {
        cout << "Coord2D(x = " << _x << ", y = " << _y << ")
destroyed" << endl;
    }

    Coord2D(const Coord2D &) = delete;   // объявляем запрет присваивания
    void operator=(const Coord2D &) = delete; // объявляем запрет копирования
};
```

В примере выше явно запрещены присваивание и копирование объекта с помощью ключевого слова delete:

```
Coord2D(const Coord2D &) = delete;
void operator=(const Coord2D &) = delete;
```

В тоже время можно явно оставить присваивание и копирование по умолчанию с помощью ключевого слова `default`.

Однако иногда есть необходимость переопределить конструктор копирования:

```
class Coord2D {
/* ... */
    Coord2D(const Coord2D &obj) {
        _x = obj._x; // конструктор копирования имеет доступ к
        _y = obj._y; // private полям другого экземпляра класса
        cout << "Coord2D("<<_x<<","<<_y<<") copied"
<< endl;
    }
/* ... */
}

…

Coord2D c1(1, 2); // создание экземпляра класса Coord2D
Coord2D c2(c1);   // вызов конструктора копирования
Coord2D c3 = c1;  // вызов конструктора копирования
```

Также возможно переопределить копирующий оператор присваивания:

```
class Coord2D {
/* ... */
    void operator=(const Coord2D &obj) {
        _x = obj._x;
        _y = obj._y;
        cout << "Coord2D("<<_x<<","<<_y<<") copy
assigned operator" << endl;
    }
/* ... */
}

…

Coord2D c1(1, 2); // создание экземпляра класса Coord2D
Coord2D c2(3, 2); // создание экземпляра класса Coord2D
c2 = c1;          // вызов оператора присваивания
```

Результат выполнения предыдущего кода будет:

```
>> Coord2D(x = 1, y = 2) created
>> Coord2D(x = 3, y = 2) created
>> Coord2D(x = 1, y = 2) copy-assigned
>> Coord2D(x = 1, y = 2) destroyed
>> Coord2D(x = 1, y = 2) destroyed
```

Деструктор при присвоении не вызывается. Это означает, что в реализации copy assignment следует освобождать старые ресурсы перед присвоением новых значений.

lvalue и rvalue ссылки в C++

Понятие `lvalue` и `rvalue` связано с различиями в выражениях справа и слева от знака оператора присваивания. Левые и правые операнды в выражении присваивания являются самостоятельными выражениями. Для того, чтобы присваивание было допустимым, левый операнд должен ссылаться на объект, который должен быть типа `lvalue`. Правый операнд может быть любым выражением, он не обязательно должен обладать свойствами `lvalue`. Например можно декларировать `n` как объект типа `int`, тогда можно использовать выражение присваивания, например:

```
int n;
n = 3;
```

В таком случае перестановка левого и правого операнда приведут к ошибке компиляции, кроме того такое выражение будет лишено смысла:

```
3 = n;
```

Хотя невозможно использовать `rvalue` в качестве `lvalue`, однако возможно использовать `lvalue` в качестве `rvalue`. Например, можно присвоить значение `n` объекту, который обозначен через `m`:

```
int m, n;
…
m = n;
```

У операнды бинарного оператора `+` могут быть `lvalue`, но результат у него всегда `rvalue`. Например, даны целые объекты `m` и `n`, и следующее выражение приведет к ошибке:

```
m + 1 = n;
```

По базовой концепции `rvalue` является просто значением, оно не ссылается на объект. На практике `rvalue` может ссылаться на объект, однако это не обязательно так. Поэтому необходимо писать программу так, как будто rvalue не ссылаются на объекты.

Изначально, когда понятие `lvalue` было введено в `C`, оно буквально означало «выражение, применимое с левой стороны оператора присваивания». Однако позже, когда `ISO C` добавило ключевое слово `const`, это определение видоизменились.

```
const int a = 10; // 'a' - lvalue
a = 10; // но ему не может быть присвоено значение
```

Таким образом не всем `lvalue` можно присвоить значение. Те, которым можно, называются изменяемые `lvalue` (`modifiable lvalues`).

Каждое выражение в C++ является либо `lvalue`, либо `rvalue`. l`value` является выражением, которое обозначает объект (ссылается на него с указателем или без). Каждое `lvalue` бывает, в свою очередь, или модифицируемым, или немодифицируемым. К `rvalue` относится любое выражение, которое не является `lvalue`. Различия между `rvalue` и `lvalue` можно свести к тезисам:

— `modifiable lvalue` является адресуемым (может быть операндом унара `&`) и присваиваемым (может являться левым операндом `=`).
— `non-modifiable lvalue` также является адресуемым. Но оно не является присваиваемым (например тип с модификатором `const`).
— `rvalue` не является ни адресуемым, ни присваиваемым.

Примеры `lvalue` и `rvalue` операндов:

```
int x;
int& ref_x = x;
x; // lvalue, так как валидна операция &x
5; // rvalue, нельзя получить адрес
&x; // rvalue, адрес переменой x
ref_x; // lvalue, так как это переменная
++x; // lvalue, так как валидна операция &++x
"abc"; // lvalue, так как валидна операция &"abc", строки хранятся в таблице строковых литералов
int& foo(){…};
foo(); // lvalue, так как возвращается ссылка на переменную в памяти
x + 1; // rvalue, так как операция &(x + 1) некорректна, возвращает временный объект без адреса
int bar(){…}; // rvalue, возврат результата по значению
bar; // lvalue, так как возможно взять адрес у функции
*(&x + 1); // lvalue
```

Обычные ссылки, с одинарным амперсантом можно именовать как lvalue ссылки, они имеют право связываться только со значениями lvalue выражений.

Например:

```
int x;
int& ref_x = x; // lvalue ссылка на x
const int& cref_x = x; // константная lvalue ссылка на x
int& foo(){…};
int& ref0 = foo(); // lvalue ссылка на результат foo

int& ref1 = 0; // ошибка, у нуля нет отдельного представления в памяти
int& ref2 = x++; // ошибка

const int& ref3 = 10; // исключение, константные lvalue ссылки можно связывать с временными объектами, "продлевая" тем самым им жизнь
```

В стандарте C++ 11 были добавлены ссылки на `rvalue`. Синтаксически они выглядят как `lvalue` ссылки, но с двойным амперсандом. Они обладают такой же семантикой, как и обычные `lvalue` ссылки, но связывать их можно только с временными объектами, продлевая им жизнь, в отличии от константных `lvalue` позволяют изменять объект.

Например:

```
int x = 5;
int foo(){ … }

int&& rref0 = 10; // присвоение rvalue ссылки
rref0 = 20; // выполнится
int&& rref1 = foo(); // выполнится
int&& rref2 = x++; // выполнится

int&& rref3 = x; // ошибка, rvalue свяжется только с временным объектом
int&& rref4 = ++x; // ошибка

int& lref0 = rref0; // выполнится
```

В так как стандарт подразумевает, что не смотря на схожую природу, rvalue ссылки отличаются от lvalue ссылок, то можно организовать перегрузку функций, чтобы избежать проблем с адресами временных объектов.

Например:

```
template<typename T>
const T* AddressOf( const T& value ){
    return &value;
}

…

int x;
const int* ptr0 = AddressOf(x); // возвращается корректный адрес x
const int* ptr1 = AddressOf(10); // возвращается "провисший" указатель
```

Чтобы избежать неопределённого поведения, в случае если функция, получив на вход временный объект не бросает ошибку времени компиляции
Функция может получить как ссылку на lvalue, так и ссылку на rvalue (lvalue константные ссылки могут принимать rvalue значения), возвращаемый адрес, в случае rvalue возвращает локальный для функции AddressOf адрес rvalue ссылки. Очевидно это приведёт к undefined behavior. Чтобы избежать подобного явления можно определить специализацию шаблона, запрещающего явно вызов AddressOf для rvalue ссылок.

```
template<typename T>
const T* AddressOf( const T& value ){
    return &value;
}

template<typename T>
const T* AddressOf( const T&& ) = delete;

…

int x;
const int* ptr0 = AddressOf(x); // возвращается корректный адрес x
const int* ptr1 = AddressOf(10); // ошибка времени компиляции
```

Если существует некоторое `lvalue`, и ему необходимо присвоить некое `rvalue`:

```
Coord2D c1(1, 2);
c1 = Coord2D(3, 4);

…

>> Coord2D(x = 1, y = 2) created
>> Coord2D(x = 3, y = 4) created
>> Coord2D(x = 3, y = 4) copy assigned operator
>> Coord2D(x = 3, y = 4) destroyed
>> Coord2D(x = 3, y = 4) destroyed
```

В данном случае происходит копирование, однако этого копирования можно избежать.

Если объект передаётся по `rvalue` ссылке, то это значит, что вызывающий код отказывается от владения объектом, и функция может делать с ним всё, что ей необходимо. Функция `std::move` сообщает об этом намерении явно. Можно перегрузить переносящий оператор присваивания, который принимает в себя `rvalue` ссылку на объект, внутреннее состояние которого следует заменить:

```
class Coord2D {
/* ... */
    Coord2D& operator=(Coord2D&& other)
    {
        cout << "move assignment operator" << std::endl;
        std::swap(_x, other._x); // std::swap "подменяет"
```

поля текущего объекта и объекта принятого по `rvalue` ссылке

```
        std::swap(_y, other._y);
        return *this;
    }
/* ... */
}
```

Переносящий оператор присваивания перегружен таким образом, чтобы подменить внутреннее состояние двух объектов. В таком случае первый объект, имеющий некоторый продолжительный срок жизни получит желаемое состояние, а свое, ненужное, он отдаст временному объекту. Так как по `rvalue` ссылке получен временный объект, то его деструктор сработает скорее, чем у `lvalue` объекта и уничтожит подмененное состояние.

```
Coord2D c1(1, 2);
c1 = Coord2D(3, 4);

…

>> Coord2D(x = 1, y = 2) created
>> Coord2D(x = 3, y = 4) created
>> move assignment operator
>> Coord2D(x = 1, y = 2) destroyed
>> Coord2D(x = 3, y = 4) destroyed
```

Таким же образом с помощью rvalue ссылок можно перегрузить конструктор переноса (`move constructor`) , то есть он так же нужен для перемещения содержимого временного объекта.

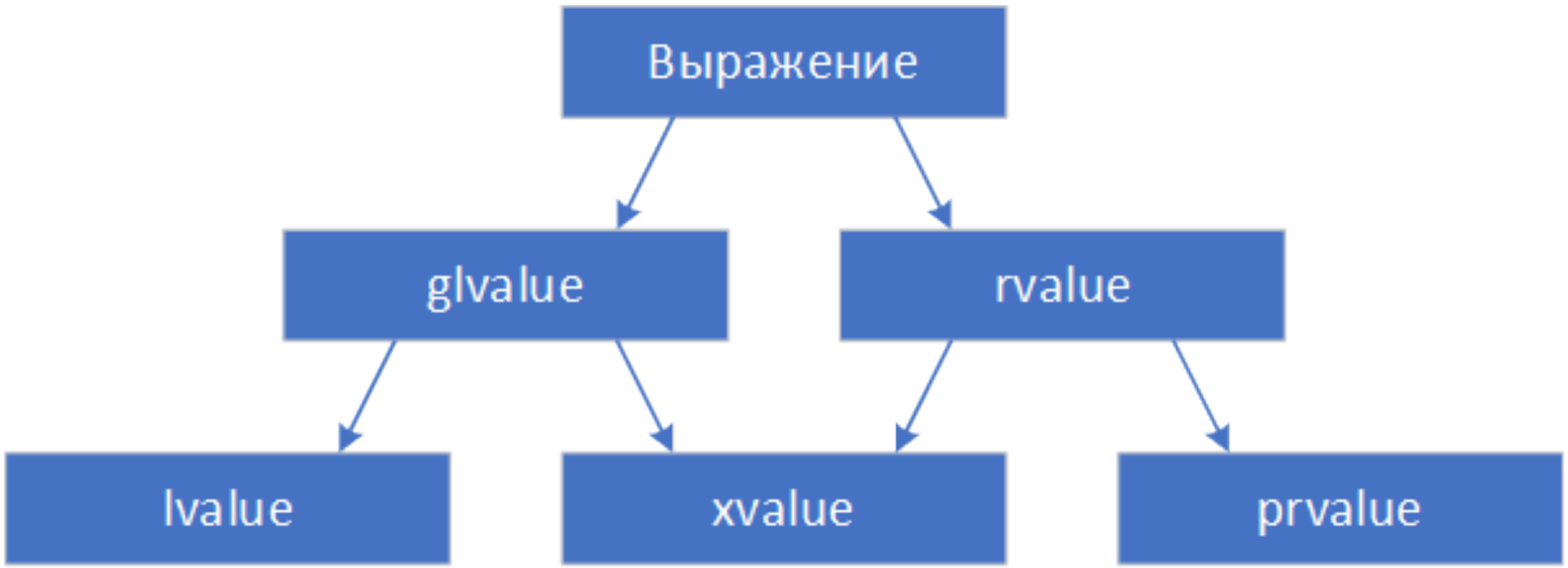
```
class Coord2D {
/* ... */
   Coord2D(Coord2D&& obj) {
      _x = obj._x;
      _y = obj._y;
      cout << "Coord2D(x = " << _x << ", y = " << _y <<
                           ") moved" << endl;
   }
}

…

int main() {
     Coord2D c0(1,2);
     Coord2D c1 = std::move(c0);
}
```

rvalue и lvalue выражения начиная с C++11

В языке C++ терминология категорий выражений достаточно сильно эволюционировала, в особенности после принятия стандарта C++11, в котором появились понятия rvalue-ссылок и семантики перемещения (move semantics).



`lvalue` — обозначает "адресуемое значение". Это выражение, которое может быть присвоено в переменную. Например, переменные и элементы массива являются `lvalue`. `Lvalue` можно использовать как левую часть присваивания.

`rvalue` — обозначает "значение, которое не является `lvalue`". Это выражение, которое обычно не имеет адреса. `Rvalue` представляет временные значения, результаты вычислений и литералы. `Rvalue` нельзя присвоить.

`xvalue` (`eXpiring value`) — обозначает "перемещающееся значение". Это выражение, которое имеет адрес, но может быть перемещено, как, например, результат вызова `std::move()` на `lvalue`. `Xvalue` используется для реализации семантики перемещения в C++.

`prvalue` (`pure rvalue`) — обозначает "временное значение, которое не имеет адреса". Это выражение, которое представляет временное значение, которое может быть сконструировано на месте. `Prvalue` включает в себя литералы и результаты временных вычислений.

`glvalue` (`generalized lvalue`) — это обобщенный термин, который охватывает как `lvalue` и `xvalue`.

Выражение является `lvalue` если ссылается на объект, уже имеющий имя, доступное вне выражения.

```
int a = 3;
a; // lvalue
int& b = a;
b; // lvalue
int* c = &a;
*c; // lvalue
int& foo() { return a; }
foo(); // lvalue
```

Выражение является `xvalue` если:

— результатом вызова функции является `rvalue` ссылка

```
int&& foo() { return 3; }
foo();  // xvalue
```

— явное приведение к `rvalue`

```
static_cast<int&&>(5);  // xvalue
std::move(5);  // эквивалентно static_cast<int&&>
```

— результат доступа к нестатическому члену, объекта `xvalue` значения

```
struct A {
    int i;
};
A&& foo() { return A(); }
foo().i;  // xvalue
```

Выражение `prvalue` не принадлежит ни к `lvalue`, ни к `xvalue`.

```
int x = 42;  // 42 — это prvalue

…

int getFive() {return 5;}
int z = getFive();  // результат вызова функции - prvalue, тк RVO

…

int y = 2 + 3;  // выражение 2 + 3 - это prvalue
```

Выражение является `rvalue` если его можно отнести к `xvalue` и `prvalue`.
Выражение является `glvalue` если его можно отнести к `xvalue` и `lvalue`.

Есть имя	Может быть перемещено	Тип
Да	Нет	glvalue, lvalue
Да	Да	rvalue, xvalue, glvalue
Нет	Да	rvalue, prvalue

Семантика перемещения std::move

Очень часто объекты, с которыми приходится работать, будут не `rvalues`, а `lvalues`. Например шаблон функции `swap()`:

```
template<class T>
void swap(T& x, T& y)
{
   T tmp(x);  // вызывает конструктор копирования
   x = y;  // вызывает оператор присваивания копированием
   y = tmp;  // вызывает оператор присваивания копированием
}

…

string x("String1");
string y("String2");
cout << "x: " << x << endl;
cout << "y: " << y << endl;
swap(x, y);
cout << "x: " << x << endl;
cout << "y: " << y << endl;
```

Принимая два объекта типа `T`, функция `swap()` меняет местами их значения, делая при этом три копии. Однако, если объекты достаточно большие, то это не очень эффективно, так как копирование происходит трижды. Этого можно избежать, так как цель функции `swap()` всего лишь поменять местами `x` и `y`, для этого можно использовать перемещение вместо копирования. Проблема состоит в том, что параметры `x` и `y` являются ссылками `lvalue`, а не ссылками `rvalue`, поэтому нет способа вызвать конструктор перемещения или оператор присваивания перемещением вместо конструктора копирования и оператора присваивания копированием. Начиная со стандарта C++ 11 в стандартной библиотеке присутствует функция `std::move()`, которая конвертирует передаваемый аргумент в `rvalue (xvalue)`. Для использования `std::move()` необходимо подключить заголовочный файл `utility`.

Программа, но уже с функцией `swap()`, которая использует `std::move()` для преобразования `lvalues` в `rvalues`, чтобы использовать семантику перемещения вместо семантики копирования.

```
template<class T>
void swap(T& x, T& y)
{
  T tmp(std::move(x)); // вызывает конструктор перемещения
  x = std::move(y); // вызывает оператор присваивания перемещением
  y = std::move(tmp); // вызывает оператор присваивания перемещением
}
```

…

```
string x("String1");
string y("String2");
cout << "x: " << x << endl;
cout << "y: " << y << endl;
swap(x, y);
cout << "x: " << x << endl;
cout << "y: " << y << endl;
```

Также можно использовать `std::move()` для заполнения контейнерных классов, таких как `std::vector`, значениями `lvalues`.

```
vector<string> v;
string str = "String";

cout << "Copying str" << endl;
v.push_back(str); // вызывает версию lvalue метода push_back(), которая копирует str в элемент массива

cout << "str: " << str << endl;
cout << "vector: " << v[0] << endl;
cout << "Moving str" << endl;

v.push_back( move(str) ); // вызывает версию rvalue метода push_back(), которая перемещает str в элемент массива

cout << "str: " << str << endl;
cout << "vector: " << v[0] << ' ' << v[1] << endl;
```

В первом случае передаётся `lvalue` в `push_back()`, поэтому используется семантика копирования для добавления элемента в вектор. По этой причине переменная `str` остается с прежним значением.
Во втором случае передаётся `rvalue` в `push_back()`, поэтому используется семантика перемещения для добавления элемента в вектор. Это более эффективно, так как элемент вектора может "украсть" значение переменной `string`, вместо его копирования. По этой же причине `str` лишается своего значения.

В стандартной реализации вектора, в функции `std::vector::push_back` для `rvalue` можно обнаружить:

```
void push_back(_Ty&& _Val) {
    emplace_back( std::move(_Val) );
}
```

Таким образом, код, добавляющий новый элемент в вектор через `rvalue` начинает работать через перемещение, а не копирование.

Нет необходимости использовать `std::move` при возврате из функции локального объекта:

```
std::string get_string(const size_t index)
{
    std::string my_string;
    …
    return std::move(my_string); // в move нет необходимости
}
```

Здесь нужно убрать `std::move`. Всю работу сделает `copy/move elision` – специальная оптимизация, которую выполняет компилятор, убирая лишние создания объектов.

Функция `std::move` не выполняет никаких перемещений, она выполняет приведение типа к `rvalue` ссылке.

```
template <class _Ty>
[[nodiscard]] constexpr remove_reference_t<_Ty>&& move(_Ty&& _Arg) noexcept {
    return static_cast<remove_reference_t<_Ty>&&>(_Arg);
}
```

Функция `std::move` обертка для `static_cast`, которая «убирает» ссылку у переданного аргумента с помощью `remove_reference_t` и, добавив &&, преобразует тип в `rvalue` ссылку. Атрибут `nodiscard` появился в C++ 17, и указывает компилятору на то, что возвращаемое функцией значение нельзя игнорировать и нужно сохранить в какую-либо переменную. Спецификатор `constexpr` производит операции внутри функции `std::move` во время компиляции, данный спецификатор добавлен в C++ 11.

Правило пяти.

До стандарта C++ 11 в языке фигурировало негласное «правило трех». Правило гласит, что если класс или структура определяет один из следующих методов, то они должны явным образом определить все три метода:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием

Эти три метода являются особыми методами, автоматически создаваемыми компилятором в случае отсутствия их явного объявления программистом. Если один из них должен быть определен программистом, то это означает, что версия, сгенерированная компилятором, не удовлетворяет потребностям класса в одном случае и, вероятно, не удовлетворит в остальных случаях.

С выходом C++ 11 правило расширилось и стало называться «правилом пяти». Теперь при реализации конструктора необходимо реализовать:

- Деструктор
- Конструктор копирования
- Оператор присваивания копированием
- Конструктор перемещения
- Оператор присваивания перемещением

Так же имеет место «правило нуля». Классы с пользовательскими деструкторами, конструкторами копирования / перемещения или операторами назначения копирования / перемещения должны иметь дело исключительно с владением (что следует из принципа единой ответственности SOLID). Другие классы не должны иметь пользовательских деструкторов, конструкторов копирования / перемещения или операторов назначения копирования / перемещения.

Perfect forwarding, std::forward.

Функция `std::forward`, применяется при идеальной передаче (`perfect forwarding`). Идеальная передача позволяет создавать функции-обертки, передающие параметры без каких-либо изменений (`lvalue` передаются как `lvalue`, а `rvalue` – как `rvalue`). В таком случае `std::move` не подходит, так как она безусловно приводит свой результат к `rvalue`. Функция `std::forward`, в случае если ссылка была передана как `rvalue`, вызывает `std::move`, а иначе просто возвращаем то, что было передано.

Реализация `std::forward` выглядит следующим образом:

```
template <class _Ty>
[[nodiscard]] constexpr _Ty&& forward(
    remove_reference_t<_Ty>& _Arg) noexcept {
    return static_cast<_Ty&&>(_Arg);
}
```

```
template <class _Ty>
[[nodiscard]] constexpr _Ty&& forward(
    remove_reference_t<_Ty>&& _Arg) noexcept {
static_assert(!is_lvalue_reference_v<_Ty>, "bad
forward call");
    return static_cast<_Ty&&>(_Arg);
}
```

Данная реализация проверяет, в случае получения `rvalue` ссылки её соответствие с помощью `static_assert` (так как `std::forward` тоже помечен как `constexpr`) и функции `is_lvalue_reference_v.`

По сути задача `std::forward` состоит в том, чтобы передать аргумент не создавая временных копий и не изменяя типа передаваемого аргумента. Например есть функция `foo`, внутри которой вызывается функция `bar`. Есть необходимость, чтобы в эти функции можно было бы передать как `rvalue`, так и `lvalue` ссылки. Для этого можно построить код следующим образом:

```
template<typename T>
void bar(T& x){
    std::cout << "lvalue bar" << std::endl;
}
template<typename T>
void bar(T&& x){
    std::cout << "rvalue bar" << std::endl;
}
template<typename T>
void foo(T& x ){
    std::cout << "lvalue foo" << std::endl;
    bar(x);
}
template<typename T>
void foo(T&& x ){
    std::cout << "rvalue foo" << std::endl;
    bar(std::move(x));
}

…

int i = 100;
foo(i);
std::cout << std::endl;
foo(100);
```

Множество перегрузок не даёт правильно переиспользовать код, поэтому необходимо как то компактно различать `rvalue` от `lvalue` во время компиляции, для этих целей может служить `std::forward`:

```
template<typename T>
void bar( T& x){
    std::cout << "lvalue bar" << std::endl;
}

template<typename T>
void bar( T&& x){
    std::cout << "rvalue bar" << std::endl;
}

template<typename T>
void foo( T&& x ){
    bar(std::forward<T>(x));
}

…

int i = 100;
foo(i);   // lvalue
std::cout << std::endl;
foo(100); // rvalue
```

Resource Acquisition is Initialization

Resource Acquisition is Initialization
(Получение ресурса есть инициализация) или RAII - программная идиома, позволяющая автоматизировать выделение и освобождение ресурсов. Например в конструкторе объект получает доступ к какому либо ресурсу (это может быть файл или соединение по сети к базе данных и т.д.) и сохраняет описатель ресурса в закрытый члена класса, а при вызове деструктура этот ресурс освобождается (закрывается файл или соединение к БД). При объявлении объекта данного класса на стеке происходит и его инициализация с вызовом конструктора, захватывающий ресурс. При выходе из области видимости объект выталкивается из стека, но перед этим вызывается деструктор объекта, который и освобождает захваченный ресурс.

Главные требования к RAII:

1) Захват ресурса в конструкторе;
2) Освобождение ресурсов в деструкторе;
3) Объекты `RAII` создаются на стеке (деструктор будет автоматически вызван при раскручивании стека);

Если создавать объект не на стеке, а например с помощью оператора `new`, то будет необходим явный вызов `delete`, что несколько противоречит идиоме.

Пример реализации RAII (нарушающий правило пяти):

```
class TelephoneLine
{
public:
    void pickUpThePhoneUp()
    {
        cout << "Line locked" << endl;
    }
    void putThePhoneDown()
    {
        cout << "Line unlocked" << endl;
    }
};

class TelephoneCall
{
public:
    TelephoneCall()
    {
        telephoneLine = new TelephoneLine();
        telephoneLine->pickUpThePhoneUp();
    }
    ~TelephoneCall()
    {
        telephoneLine->putThePhoneDown();
        delete telephoneLine;
    }
private:
    TelephoneCall (const TelephoneCall &);
    TelephoneCall& operator=(const TelephoneCall &);
    TelephoneLine * telephoneLine;
};

…

TelephoneCall call;
```

Если ресурс, которым должен управлять класс является уникальным (это может быть файловый дескриптор, сокет, память в куче и др.) RAII идеология должна правильно взаимодействовать с возможным копированием или присваиванием:

```
template<typename T>
class SmartPtr
{
    T *ptr_;
public:

    SmartPtr(): ptr_(nullptr) {}
    explicit SmartPtr( T *ptr): ptr_(ptr) {}
    SmartPtr(const SmartPtr&) = delete;
    SmartPtr& operator=(const SmartPtr&) = delete;

    SmartPtr& operator=(SmartPtr&& other){
        if( &other != this ){
            ptr_ = other.ptr_;
            other.ptr_ = nullptr;
        }
        return *this;
    }

    SmartPtr(SmartPtr&& other): ptr_(other.ptr_){
        other.ptr_ = nullptr;
    }

    ~SmartPtr(){ delete ptr_; }
    T& operator*() const { return *ptr_; }
    T* operator->() const { return ptr_; }
};

int main(){
    SmartPtr<int> sptr0(new int(10));
    SmartPtr<int> sptr1;
    sptr1 = std::move(sptr0);
}
```

Умные указатели

Умные указатели — классический пример идиомы `RAII`. По сути это объекты, инкапсулирующие владение памятью. Стандартная библиотека в C++11 имеет 4 класса умных указателей:

`scoped_ptr` - время жизни объекта ограничено временем жизни умного указателя.
`shared_ptr` - разделяемый объект, реализация с подсчётом ссылок.
`auto_ptr`, `unique_ptr` - эксклюзивное владение объектом с передачей владения при присваивании.
`weak_ptr` - реализация с подсчётом ссылок, слабая ссылка (используется с `shared_ptr`).

Класс `auto_ptr` (`automatic pointer` - автоматический указатель). Данный класс предоставляется стандартной библиотекой `STL` C++ и предназначен для работы с объектами, которые обычно необходимо удалять явно (например, объекты, созданные динамически с помощью оператора new). Для создания объекта класса `auto_ptr` параметром конструктора должен быть указатель на объект, созданный динамически. Дальше с `auto_ptr` можно работать почти как с обычным указателем, который указывает на тот же динамический объект, на который указывал исходный указатель. `auto_ptr` был исключен из стандарта языка C++ 17, вместо него рекомендуется использовать `unique_ptr`.

```
// создание двух автоматических указателей
// под один из них выделяем память для Coord2D
auto_ptr<Coord2D> p1(new Coord2D(6, 5));
auto_ptr<Coord2D> p2;

p2 = p1; // передача права владения
Coord2D *s = p2.get();  // присваивание автоматического указателя обычному
cout << p1->getX() << endl; // вызов функции через автоматический
указатель
```

Класс `unique_ptr` является заменой `auto_ptr` в C++11. `unique_ptr` полностью владеет переданным ему объектом и не делится «владением» еще с другими классами. Умный указатель `unique_ptr` находится в заголовочном файле `memory`. Когда `unique_ptr` выходит из области видимости, он удаляет объект, которым владеет. В отличие от `auto_ptr`, `unique_ptr` корректно реализовывает семантику перемещения:

```
// создание двух автоматических указателей
// под один из них выделяем память для Coord2D
unique_ptr<Coord2D> p1(new Coord2D(6, 5));
unique_ptr<Coord2D> p2;

p2 = std::move(p1); // передача права владения
Coord2D *s = p2.get();  // присваивание unique_ptr указателя обычному
cout << p1->getX() << endl; // ошибка, p1 больше не указывает на объект
```

`shared_ptr` - умный указатель с подсчетом ссылок. Внутри `shared_ptr` существует переменная, которая хранит количество указателей, которые ссылаются на объект. Если эта переменная становится равной нулю, то объект уничтожается. Счетчик инкрементируется при каждом вызове либо оператора копирования либо оператора присваивания. Так же у `shared_ptr` есть оператор приведения к `bool`, что в итоге дает привычный синтаксис указателей.

```
// создание двух shared указателей
// под один из них выделяем память для Coord2D
shared_ptr<Coord2D> p1(new Coord2D(6, 5));
shared_ptr<Coord2D> p2;
p2 = p1;  // инкремент
Coord2D *s = p2.get(); // присваивание shared указателя обычному
if( p2 ){
  cout << p2->getX() << endl;// вызов функции через shared указатель
}
```

Владение динамической памятью с помощью `shared_ptr` может быть осложнено. Например часто встречается проблема циклической зависимости указателей.

```
class Human
{
    string m_name;
    shared_ptr<Human> m_partner;  // внутренний указатель изначально пустой
public:
    Human(const string &name): m_name(name){ cout << m_name << endl;}

    ~Human(){ cout << m_name << " destroyed" << endl;}

    friend bool partnerUp(shared_ptr<Human> &h1,shared_ptr<Human> &h2){
        h1->m_partner = h2;
        h2->m_partner = h1;
        return true;
    }
};

…

shared_ptr<Human> human1 = shared_ptr<Human>(new Human("Human1"));
shared_ptr<Human> human2 = shared_ptr<Human>(new Human("Human2"));
partnerUp(human1, human2); //  human1 указывает на human2, а human2 указывает на human1
```

В примере динамически создаются два объекта `human1` и `human2` класса `Human`. Их управление передаётся умным указателям типа `shared_ptr`. Затем объекты «связываются» с помощью дружественной функции `partnerUp()`. В итоге получаются 4 `shared_ptr`, попарно указывающих на 2 объекта в динамической памяти. В конце функции `main` объект `human2` выходит из области видимости первым. `shared_ptr`, указывающий на `human2` не освободит память, так как указатель на `human2` ещё остался у `human1`. Тоже самое случится с объектом `human1`, он не будет уничтожен по причине того, что указатель на него находится у `human2`. В итоге получается утечка памяти, так как вернуть динамическую память уже невозможно в рамках этой программы.

Умный указатель `weak_ptr` был разработан для решения проблемы «циклической зависимости», описанной выше. `weak_ptr` является наблюдателем — он может наблюдать и получать доступ к тому же объекту, на который указывает `shared_ptr` (или другой `weak_ptr`), но не считаться владельцем этого объекта. Когда `shared_ptr` выходит из области видимости, он проверяет, есть ли другие владельцы `shared_ptr`, однако `weak_ptr` владельцем не считается.
Недостатком умного указателя `weak_ptr` является то, что его нельзя использовать напрямую (нет оператора `->`). Чтобы использовать `weak_ptr`, вы сначала должны конвертировать его в `shared_ptr` с помощью метода `lock()`, а затем уже использовать `shared_ptr`.

Класс Human, переписанный с помощью weak_ptr:

```
class Human
{
    string m_name;
    weak_ptr<Human> m_partner;  // внутренний указатель изначально пустой
public:
    Human(const string &name): m_name(name){ cout << m_name << endl;}

    ~Human(){ cout << m_name << " destroyed" << endl;}

    const std::shared_ptr<Human> getPartner() const { return
m_partner.lock(); }
    friend bool partnerUp(shared_ptr<Human> &h1,shared_ptr<Human> &h2){
        h1->m_partner = h2;
        h2->m_partner = h1;
        return true;
    }
};

…

shared_ptr<Human> human1 = shared_ptr<Human>(new Human("Human1"));
shared_ptr<Human> human2 = shared_ptr<Human>(new Human(«Human2"));
partnerUp(human1, human2); //  human1 указывает на human2, а human2 указывает на
human1
shared_ptr<Human> partner = human2->getPartner();
```