**Перемещающий конструктор. Примеры**

Рассмотрим функцию, выполняющую переворот строки, учитывая, что на вход подано *rvalue (временный объект)*:

1. string reverse(string&& str) {
2. string reverse\_string(str.buffer);
3. str.buffer = nullptr;
4. std::reverse(reverse\_string.begin(), reverse\_string.end());
5. **return** reverse\_string;
6. }

Тут экономия (более высокая эффективность) обеспечивается за счет того, что результирующая строка использует буфер (какие-то сырые данные, возможно **char**\*) из временной строки. Более подробно про то, как работает этот код: [ссылки в С++: rvalue- и lvalue- references](https://pro-prof.com/forums/topic/cplusplus_references). Проблема тут заключается в нарушении инкапсуляции, т.к. сторонний код (функция реверса строки) не должна иметь доступ к сырым данным строки (и даже знать как она устроена внутри).

Решением проблемы являются перемещающие конструкторы, которые вызываются автоматически вместо конструкторов копирования если аргументом является временный объект. Для сравнения рассмотрим конструктор копирования строки и перемещающий конструктор:

1. string(string **const**& str)
2. : m\_buf(**new** **char**[str.size()+1]), m\_size(str.size()) {
3. strcpy(m\_buf, str.m\_buf);
4. }
5. string(string **const**&& str)
6. : m\_buf(str.m\_buf), // char\* m\_buf - заменяется указатель
7. m\_size(str.size()) {
8. str.m\_buf = nullptr;
9. str.m\_size = 0;
10. }

Конструктор копирования, как это обычно происходит, создает новый буфер для хранения данных, вызывая *оператор new*, а перемещающий конструктор — забирает данные у переданного ему временного объекта. Мы можем реализовать такой конструктор и для нашего [примитивного аналога вектора](https://pro-prof.com/forums/topic/cplusplus_custom_array_class) (класса массива, размер которого увеличивается автоматически при вставке элемента):

1. **template** <**class** ElementType>
2. Array<ElementType>::Array(Array<ElementType>&& array)
3. : m\_size(array.m\_size), m\_realSize(array.m\_realSize), m\_array(array.m\_array) {
4. array.m\_size = 0;
5. array.m\_realSize = 0;
6. array.m\_array = nullptr;
7. }

В этих примерах перемещающий конструктор является лишь более оптимальной версией конструктора копирования для случаев, когда на вход подан временный объект, например:

1. **void** foo(string str);
2. **void** bar(**const** string& str);
3. foo(employee.get\_name());
4. bar(employee.get\_office());

**Преимущества перемещающего конструктора**

Было показано, что **перемещающий конструктор является облегченной версией конструктора копирования**, однако иногда реализуют перемещающий конструктор для классов, копирование в которых запрещено. Ярчайший пример — std::unique\_ptr<>, представляющий собой один из умных указателей стандартной библиотеки, реализующий [идиому RAII](https://pro-prof.com/forums/topic/resources-control-objects-raii). Суть этого класса заключается в том, что он владеет единственным указателем на некоторый объект (и автоматически разрушает объект в определенных ситуациях). Указатель должен быть единственным, следовательно конструктор копирования не должен быть доступен, однако перемещение для этого класса вполне логично (мы можем передавать владение указателем от одного объекта к другому — поэтому unique\_ptr, в частности, может быть использован в качестве возвращаемого значения функции. Другими примерами такого поведения из стандартной библиотеки являются классы std::fstream и std::thread — копирование для них лишено смысла, однако передача владения файлом из одной функции в другую может быть логична.  
Таким образом, семантика перемещения является не только средством повышения эффективности программ, но и **позволяет реализовать передачу владения объектом в случаях, когда копирование запрещено** (является очень длительной операцией или вообще лишено смысла).

Стоит отметить, что не для всех классов имеет смысл писать перемещающий конструктор. Так, вы не сможете например повысить эффективность за счет перемещения класса точки или любого другого класса не владеющего внешним ресурсом. Ведь повышение эффективности обеспечивается заменой копирования этого ресурса перемещением:

1. **struct** Point {
2. **double** x, y;
3. Point(**const** Point& point)
4. : x(point.x),
5. y(point.y) {
6. }
7. }

Для этого примера не получится реализовать перемещающий конструктор более эффективно, чем конструктор копирования.

**std::move**

В ряде случаев нужно явно указать компилятору, что объект является временным — сделать это можно с помощью функции std::move, выполняющей приведение типа к *rvalue*(эквивалентной **static\_cast**<T&&>). Если вам нужно переместить именованный (не временный) объект — используйте std::move, т.к. в противном случае будет вызван конструктор копирования:

1. **void** foo(Type&& obj) {
2. Type copy\_obj(obj); // вызов конструктора копирования
3. Type move\_obj(std::move(obj)); // вызов перемещающего конструктора
4. }
5. foo(Type()); // успешный вызов функции foo
6. Type obj;
7. foo(obj); // ошибка, функция принимает ссылку на r-значение, но мы передаем l-значение

Этот пример напоминает, что функции, принимающие указатели на *rvalue* будут вызваны только для временных объектов. Однако, внутри функции эти объекты уже не являются временными, ведь у них есть имя — поэтому при создании объекта copy\_obj будет вызван конструктор копирования. Если нам нужно перемещение — нужно явно указать, что объект является временным с помощью std::move.

В рассмотренном выше примере данные строки (класс string) хранились в виде массива символов, поэтому вызов m\_buf(str.m\_buf) работал эффективно — просто заменялся указатель. Однако, если данных хранились бы в векторе, то такой код привел бы к вызову конструктора копирования:

1. string(string **const**&& str)
2. : m\_buf(str.m\_buf) // vector<char> m\_buf, вызывается конструктор копирования
3. {
4. }

Решить проблему можно с помощью std::move:

1. string(string **const**&& str)
2. : m\_buf(std::move(str.m\_buf)) // vector<char> m\_buf, вызывается перемещающий конструктор
3. {
4. }

Важно помнить, что std::move не перемещает объект, а лишь выполняет приведение типа, которое позволяет вызвать перемещающий конструктор.

**std::swap**

Функция std::swap предназначена для обмена значений двух объектов. До принятия *стандарта С++11* обмен происходил с использованием вспомогательной переменной, что требовало выполнения конструктора копирования и двух операций присваивания:

1. **template**<**class** T>
2. **void** swap(T& a, T& b) {
3. T c(a); a=b; b=c;
4. }

Сейчас для классов, поддерживающих семантику перемещения эта функция вызывает перемещающий конструктор и два перемещающих оператора присваивания:

1. **template** <**class** T> **void** swap(T& a, T& b) {
2. T c(std::move(a)); a=std::move(b); b=std::move(c);
3. }

Это работает очень эффективно и может применяться для более простой реализации перемещающего конструктора:

1. string(string&& str)
2. : string() {
3. swap(\***this**, str);
4. }

Перемещающий конструктор вызывает конструктор по умолчанию, в результате чего создается строка нулевой длины (память не выделяется, код работает очень быстро). Затем текущий объект обменивается при помощи перемещающей функции swap с временным, переданным в качестве аргумента.  
Не во всех случаях такой подход будет одинаково эффективным, там для класса Array, рассмотренного выше, конструктор по умолчанию динамически выделяет блок памяти некоторого начального размера.

**Перемещающий оператор присваивания и std::swap**

Оператор присваивания является одной из наиболее часто используемых функций, даже std::swap использует его, при этом семантика перемещения может повысить эффективность:

1. Array<**int**> get\_values();
2. Array<**int**> values;
3. values = get\_values();

В данном примере вызов функции создает временный объект, который присваивается объекту values. Если для класса Array не реализована перемещающая версия оператора, то произойдет копирование данных массива.

1. **template**<**typename** T>
2. Array<T>& **operator**=(Array<T>&& source) {
3. **if** (**this** != &source) {
4. **delete** m\_array;
5. m\_array = source.m\_array;
6. m\_size = source.m\_size;
7. m\_realSize = source.m\_realSize;
8. source.m\_array = nullptr;
9. }
10. **return** \***this**;
11. }

Оператор выполняет проверку самоприсваивания чтобы не уничтожить единственную копию данных, затем освобождает память из под данных, уже имеющихся в объекте, помещает в него данные из объекта правой части оператор и обнуляет указатель временного объекта.

С помощью функции std::swap этот код может быть упрощен:

1. **template**<**typename** T>
2. Array<T>& **operator**=(Array<T> source) {
3. swap(\***this**, source);
4. **return** \***this**;
5. }

Обратите внимание, что объект передается по значению -это очень забавный трюк, который позволяет реализовать одновременно и копирующее и перемещающее присваивания. Дело в том, что если в качестве аргумента будет передан временный объект, то при создании переменной source сработает перемещающий конструктор, но если на вход подано lvalue, то в любом случае потребовалось бы создавать вспомогательный объект (для обмена значений).

**Вспомогательная литература:**

1. [Ссылки в С++: rvalue- и lvalue- references](https://pro-prof.com/forums/topic/cplusplus_references) — описываются ссылки на r-значения, появившиеся в С++11 и лежащие в основе семантики перемещения;
2. [Реализация класса расширяющегося одномерного массива](https://pro-prof.com/forums/topic/cplusplus_custom_array_class) — в примерах статьи используется перемещающий конструктор для этого класса;
3. [Идиома RAII. Объекты управления ресурсами в C++](https://pro-prof.com/forums/topic/resources-control-objects-raii) — описывается идиома, лежащая в основе класса unique\_ptr из стандартной библиотеки, рассматриваемого в качестве примера класса, поддерживающего семантику перемещения;
4. [Умный указатель unique\_ptr](https://pro-prof.com/forums/topic/cplusplus-unique_ptr-smart_pointers) — более подробное описание возможностей и особенностей класса unique\_ptr.