Программирование на C++. Часть 2
Лекция 4

ПМИ Семестр 2

Демяненко Я.М.

2026

Семантика перемещения

Семантика перемещения появилась в стандарте C++11.

Она нацелена на **уменьшение количества создаваемых копий** объектов при выполнении конструктора копии и операции присваивания, которые вызываются для **rvalue** выражений.

Пример. Реализовать move-семантику на примере упрощённого класса для динамического массива

Для этого достаточно:

- определить конструктор с параметрами по умолчанию
- определить конструктор копии
- определить деструктор
- перегрузить операцию сложения двух массивов одинакового размера
- перегрузить операцию присваивания

```
class myvector {
private:
  static int f;
  string name;
  int size;
  int * vect;
public:
  myvector(int s = 1, string nm ="noname"): size(s), name(nm){
    f++;
    cout << "constructor > " << name<<" "<<f<< endl;
    vect = new int[s];
  }
```

```
myvector(const myvector & v): size(v.size){
    f++;
    name = "Copy ( "+v.name+" )";
    cout << "copy > " << name << " " <<  f << endl;
    vect = new int[v.size];
    for (int i = 0; i<v.size; ++i)
      vect[i] = v.vect[i];
  }


~myvector(){
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
```

```
myvector& operator= (const myvector &v){

    cout << name <<" operator= > " << v.name <<" "<< f << endl;

    if (&v != this) {
      delete[] vect;
      vect = new int[v.size];
      for (int i = 0; i<v.size; ++i)
        vect[i] = v.vect[i];
      size = v.size;
    }
    return *this;
  }
```

```
 myvector operator+(const myvector& v){
    if (size == v.size) {
      myvector v1(size, name + " + "+v.name);
      for (int i = 0; i < size; ++i)
        v1.vect[i] = vect[i] + v.vect[i];
      return v1;
    }
    return *this;//лучше использовать исключение
  }
};
```

```
#include "vect.h"
int myvector::f = 0;
int main() {

  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```

Командная строка

```
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy ( A ) 3
constructor > A + B 4
copy > Copy ( A + B ) 5
destructor > A + B 4
A operator= > B 4
constructor > A + Copy ( A ) 5
copy > Copy ( A + Copy ( A ) ) 6
destructor > A + Copy ( A ) 5
B operator= > Copy ( A + Copy ( A ) ) 5
destructor > Copy ( A + Copy ( A ) ) 4
destructor > Copy ( A + B ) 3
destructor > Copy ( A ) 2
destructor > B 1
destructor > A 0
```

Именно для них будут создаваться дополнительные временные объекты, которые после использования в конструкторе копии и операции присваивания тут же будут удалены.

Выражениями **rvalue** являются

a + b внутри вызова конструктора копии для объекта cc

и **a + D** в операторе присваивания.

Решение проблемы

Во многих современных компиляторах встроен механизм Return Value Optimization (RVO), решающий, в том числе, и эту проблему.

Однако автоматическая оптимизация не всегда бывает эффективной, поэтому в стандарте C++11 Бьярн Страуструп предложил вынести решение на уровень языка.

Для этого были введены **move-конструктор** и **move-operator=**

Идея move-семантики

Не удалять временный объект и **не выделять** память для полей в новом объекте, а **инициализировать** поля в создаваемом объекте **ссылками** на поля временного объекта.

Конструктор копирования выделяет новую область памяти для хранения данных, вызывая оператор **new**, а **перемещающий конструктор** — **забирает** данные **у** переданного ему **временного объекта**.

При использовании **move-семантики деструктор не** должен **освобождать** память временного объекта, поскольку ссылкой на неё инициализируется поле в другом объекте.

Для этого в **move-конструкторе** и **move-операции присваивания** поле указатель временного объекта меняет значение на **nullptr**,
а **в деструкторе** выделенная **память освобождается** только, **если** поле указатель не равен nullptr

move-конструктор

```
//move-конструктор
myvector(myvector&& v){

  name = "Move_Copy ( " + v.name+" )";
  cout << "move > " << name << " " << f << endl;

  size = v.size;
  vect = v.vect;

  v.vect = nullptr; // Не позволит сразу удалить временный объект
}
```

move-операция присваивания

```
//move-операция присваивания
myvector& operator=(myvector&& v){

  cout << name <<" operator-move= > "<< v.name <<" "<< f <<endl;

  if (vect != nullptr)
    delete[] vect;

  size = v.size;
  vect = v.vect;
  v.vect = nullptr;

  return *this;
}
```

деструктор

```
//изменённый деструктор
~myvector(){
  if (vect != nullptr) {
    f--;
    cout << "destructor > " << name << " " << f << endl;
    delete[] vect;
  }
}
```

myvector && v

Чтобы отличать функции с перемещающей семантикой в стандарте C++11 введены **rvalue ссылки — myvector && v**.

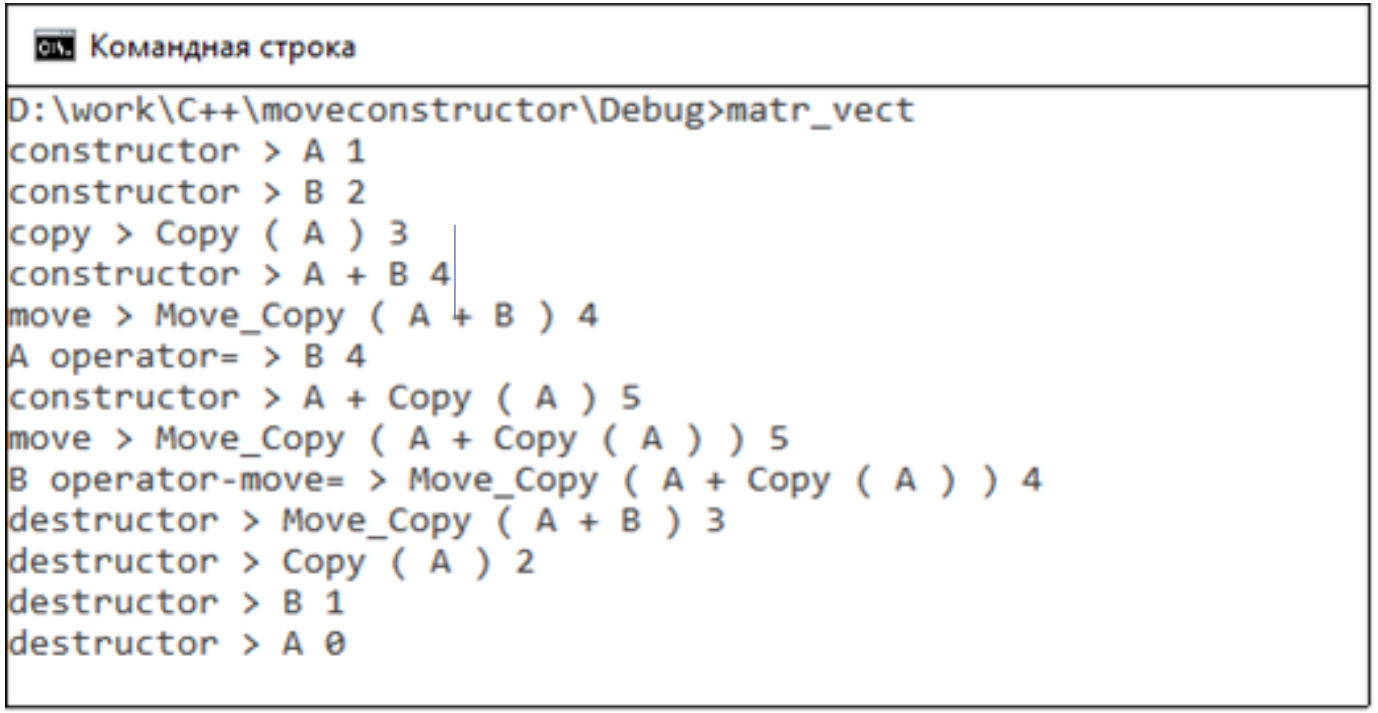
При этом компилятор будет **использовать** функции с **перемещающей семантикой** только в случае, если параметром является **rvalue выражение** (временный объект).

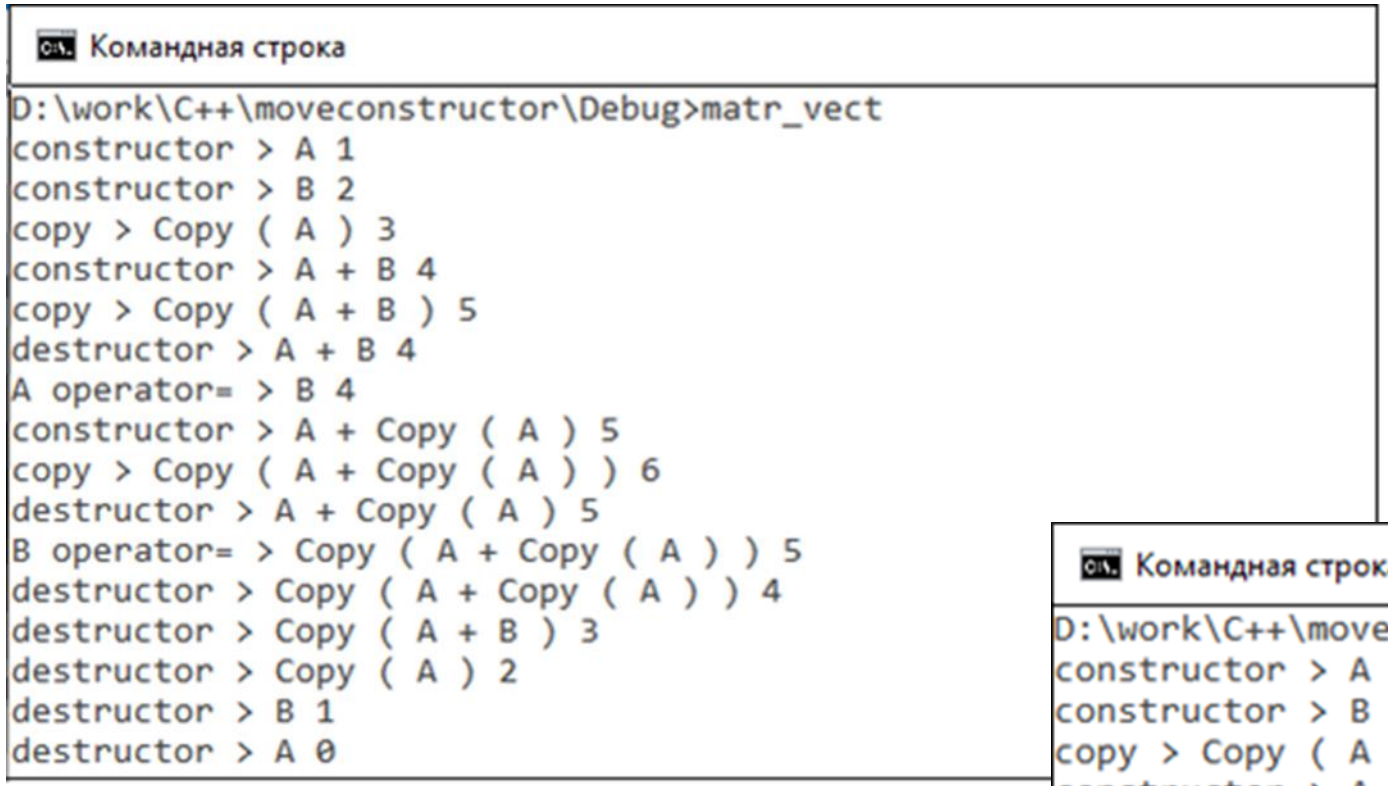
Теперь, при выполнении той же самой программы, для строки `myvector cc(a+b);` компилятор выберет move-конструктор вместо конструктора копии.

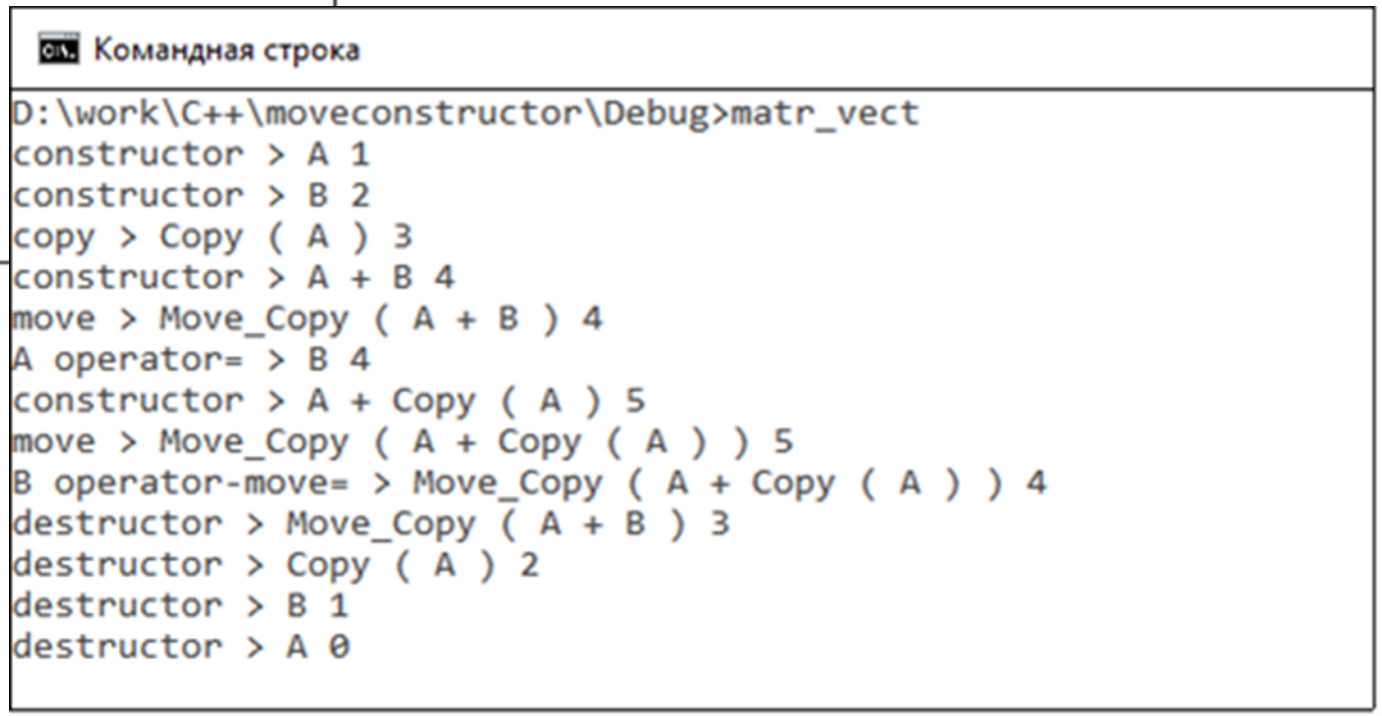
А для строки `b = a + D;` компилятор выберет move-операцию присваивания.

```
#include "vect.h"
int myvector::f = 0;
int main() {

  myvector a(3,"A"), b(3,"B"),D(a);
  myvector cc (a + b);
  a = b;
  b = a + D;
  return 0;
}
```




Командная строка
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy (A) 3
constructor > A + B 4
copy > Copy (A + B) 5
destructor > A + B 4
A operator= > B 4
constructor > A + Copy (A) 5
copy > Copy (A + Copy (A)) 6
destructor > A + Copy (A) 5
B operator= > Copy (A + Copy (A)) 5
destructor > Copy (A + Copy (A)) 4
destructor > Copy (A + B) 3
destructor > Copy (A) 2
destructor > B 1
destructor > A 0


Командная строка
D:\work\C++\moveconstructor\Debug>matr_vect
constructor > A 1
constructor > B 2
copy > Copy (A) 3
constructor > A + B 4
move > Move_Copy (A + B) 4
A operator= > B 4
constructor > A + Copy (A) 5
move > Move_Copy (A + Copy (A)) 5
B operator-move= > Move_Copy (A + Copy (A)) 4
destructor > Move_Copy (A + B) 3
destructor > Copy (A) 2
destructor > B 1
destructor > A 0

Итог

Ввиду наличия большого количества стандартных классов использовать move-конструкторы и move-операции присваивания приходится редко, однако знание такого механизма необходимо.

Правило пяти

Правило пяти — правило в C++, которое гласит, что если явно определяется один из пяти специальных методов класса, скорее всего, нужно явно определить и остальные четыре.

Это эволюция предшествующего правила трёх, адаптированная для учёта нововведений C++11, таких как семантика перемещения.

Правило пяти применяется, когда класс управляет ресурсами: динамической памятью, файловыми дескрипторами, сетевыми подключениями и т. д.

Цель правила — корректная обработка владения ресурсами и предотвращение таких проблем, как двойное освобождение или утечки памяти.

Копирование ресурсов может быть ресурсоёмким с точки зрения производительности, а перемещение позволяет избежать ненужных операций копирования, передавая владение ресурсами непосредственно новому объекту.

В правило пяти включаются:

- Деструктор — автоматически вызывается при уничтожении объекта, гарантирует, что все выделенные ресурсы будут корректно освобождены.
- Конструктор копирования — позволяет создавать новые объекты как копии существующих.
- Оператор присваивания копированием — позволяет одному уже существующему объекту принять состояние другого существующего объекта.
- Конструктор перемещения — передаёт владение ресурсами от одного объекта к другому.
- Оператор присваивания перемещением — используется, когда существующему объекту присваивается значение rvalue.