Семантика перемещения

В чем проблема

Фундаментальным принципом организации работы функций в языках C/C++ является соглашение о передаче параметров по значению. Это означает, что формальный параметр функции является самостоятельным объектом, и при вызове функции выполняется копирование значения из фактического параметра в этот формальный параметр. Далее при выполнении операции возврата return значение, вычисляемое в этом операторе, также копируется в некоторый временный объект, который далее участвует в вычислении тех выражений, откуда эта функция была вызвана.

Пока речь идет о передаче простых значений (например, базовых типов), это копирование не вызывает особых проблем и эффективно защищает внешний по отношению к функции контекст от возможного случайного изменения в результате вызова функции. Однако для более "тяжеловесных" объектов — классов языка C++ подобное копирование может оказаться весьма накладным в плане расходования вычислительных мощностей процессора и соответственно вылиться в заметное увеличение времени работы. Для обеспечения корректности передачи параметров и возврата значений в случае классов рекомендуется всегда следовать **правилу трех**, т.е. в явном виде предоставлять для класса конструктор копирования, оператор присваивания с копированием и деструктор.

Одной из естественных попыток устранить эту проблему стало использование указателей и ссылок при передаче параметров. В этом случае в функцию копируется не сам класс, а только ссылка/указатель, которые уже весьма малы по занимаемой памяти и не повлекут заметных расходов на их копирование. Однако в некоторых случаях нам принципиально не удается использовать ссылки, особенно если речь мдет о передаче возвращаемого значения из функции.

Рассмотрим следующий простой пример — класс, представляющий собой простейший массив. Предположим, что для таких массивов определена операция сложения как почленная сумма элементов массивов. Для простоты представленный код не будет содержать никаких проверок корректности, поскольку не в них сейчас проблема. То есть мы обращаем внимание только на суть операций копирования при передаче параметров.

```
class Arr {
   size_t size;   // размер
   double *x;     // сам массив
 public:
   Arr(size_t _size=0) : x(nullptr), size(_size) {
      if(size) x = new double[size];
   }

   // правило трех:
   Arr(const Arr &b) : x(nullptr), size(b.size) {
      if(size) x = new double[size];
      for (size_t i = 0; i < size; ++i) { x[i] = b.x[i]; }
   }
   const Arr & operator=(const Arr &b) {
      delete [] x;   x = nullptr;
      size = b.size;
      if(size) x = new double[size];
      for (size_t i = 0; i < size; ++i) { x[i] = b.x[i]; }
      return *this;
   }
   ~Arr() { delete [] x;}
```

```
    // сумма
    Arr operator+(const Arr &b) {   // не обращаем внимания на
       Arr sum(size);               // корректность этой операции
       for (size_t i = 0; i < size && i < b.size; ++i) {
          sum.x[i] = x[i] + b.x[i];
       }
       return sum;
    }
    // остальные методы класса нам пока не интересны
};
```

Как мы видим, оператор присваивания и конструктор копирования содержат циклы прохода по всем значениям. И в операторе сложения мы не можем уйти от возвращения вычисленного внутри результата sum по значению. Следовательно, здесь будет задействован конструктор копирования с дополнительным циклом прохода по всем элементам. Здесь у нас параметрами методов является ссылка `const Arr &b` и поэтому проблем при передаче этого параметра нет. Но вполне могла бы быть ситуация, когда для некоторого вызова допустима сигнатура только с параметром по зачению (`Arr b`).

Рассмотрим вот такой пример.

```
size_t n = 1000000;
Arr a(n), b(n), c(n), d(n);
... заполнение этих массивов .....
d = a + b + c;
```

В последней операции два сложения и, следовательно два раза возникнет копирование класса Arr. Кроме этого, результатом сложения будет временный объект, который нужен только для того, чтобы стать правым аргументом в операции присваивания. После присваивания этот временный объект уже никому не нужен. Возникает естественное желание попытаться воспользоваться этим временным статусом и съэкономить на копированиях в тех местах, где участвуют временные объекты, которые будут почти сразу уничтожены.

Как ее решать

Это желание привело к введению в язык техники перемещения и понятия rvalue ссылки. Если не вдаваться в тонкости, то rvalue — это ссылка на объект, который повляется в правой части некоторой операции и уже не нужен (может быть уничтожен) после выполнения этой операции. Классическими примерами таких операций являются как раз присваивание и конструктор копирования в случае, когда их аргументы получаются как временные объекты, созданные по результатам некоторых других вычислений. В этом случае мы можем упростить конструктор и присваивание, просто обменяв области памяти левой и правой части присваивания (и аналогично для конструктора). При этом обмен больших областей часто можно выполнить просто обменом указателей, что намного проще, чем прямой обмен данными. Эта техника называется перемещением и выливается во введение конструктора перемещения и присваивания с перемещением. В нашем случае это выглядит так

```
Arr(Arr &&b) : size(b.size), x(b.x) {
   b.x = nullptr; b.size = 0;
}
const Arr & operator=(Arr &&b) {
   size_t n = size; size = b.size; b.size = n;
```

```
        double *p = x; x = b.x; b.x = p;
        return *this;
    }
```

Здесь `Arr &&b` — формальный синтаксис объявления ссылки rvalue для типа Arr. Эта ссылке не константная так как параметр b в результате меняет свое состояние.

В этих функциях исчезли циклы копирования, а суть сводится к обмену значениями размера массива и указателями на его начало. Обратите внимание, в присваивании нет освобождения текущего массива `delete []x;`, этот укаатель просто обменивается с `b.x` и будет освобожден, когда мы выйдем из области видимости объекта `b`, и он будет уничтожен.

При такой технике в вышеупомянутом примере присваивания с суммой трех объектов в правой части уже будут срабатывать конструкторы копирования и присваивания с перемещением.

Заметим, что современные компиляторы иногда самостоятельно могут оптимизировать вычисления, используя технику RVO (Return Value Optimization). В этом случае они могут применять технику перемещения даже если программист не определил явно конструктор и оператор присваивания с перемещением. Конечно, это будет происходить только в относительно простых случаях. Для компилятора GCC существует параметр `-fno-elide-constructors`, который отключает RVO, и в этом случае все операции происходят с необходимым копированием. Вот наглядный пример

```
// файл rvo.cpp
#include <iostream>

struct A {
   int x;
   A(int xx) : x(xx) { std::cout << "A(int)\n"; }
   A(const A &b) : x(b.x) { std::cout << "A(const A&)\n"; }
   const A & operator=(const A &b) {
      x = b.x;
      std::cout << "operator=\n";
      return *this;
   }
   A operator+(const A &b) {
      A a(x + b.x);
      std::cout << "operator+\n";
      return a;
   }
   ~A() { std::cout << "~A()\n"; }
};

int main()
{
    A a(10), b(20), c(0);
    std::cout << "------------\n";
    c = a + b;
    return 0;
}
```

Теперь посмотрим как проявляется RVO при разных режимах компиляции.

```
g++  rvo.cpp                    g++ -fno-elide-constructors rvo.cpp
вывод программы:                вывод программы:

A(int)                          A(int)
```

```
A(int)                          A(int)
A(int)                          A(int)
------------                    ------------
A(int)                          A(int)
operator+                       operator+
operator=                       A(const A&)
~A()                            ~A()
~A()                            operator=
~A()                            ~A()
~A()                            ~A()
                                ~A()
                                ~A()
```

Видно, что во втором случае происходит создание и уничтожение дополнительного объекта при отработке оператора return, а в первом случае фактически происходит перемещение в локальный объект `A a` в операторе сложения.

В некоторых случаях программист может сам решить, что некоторый объект в его программе уже стал не нужен и может быть использован как приемник для перемещения данных какой-либо операции. Для этого используется функция `std::move`, которая превращает свой аргумент в rvalue ссылку.

В качестве примера рассмотрим следующую программу, которая иллюстрирует семантику перемещения, правило пяти и использование функции move для простейшего класса-массива. Запустите эту программу и проследите вызовы конструкторов, деструкторов и других операторов класса при выполнении простейших действийб в каких случаях используется копирование, а в каких перемещение. Для этого во все операторы вставлены строчки для печати соответствующих сообщений.

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

class A
{
    size_t n;
    double *x;
public:
    double & operator[](int i) { return x[i]; }
    const double & operator[](int i) const { return x[i]; }
    size_t Size() const { return n; }

    A(size_t nn) {
        n = nn;
        x = new double[nn];
        cout << "base A()\n";
    }
    A(const A & b) {
        x = new double[b.n];
        n = b.n;
        for (size_t i = 0; i < n; i++) x[i] = b.x[i];
        cout << "copy A()\n";
    }
    A(A &&b) {
        x = b.x;
        n = b.n;
        b.x = nullptr;
        b.n = 0;
```

```
        cout << "move A()\n";
    }
    const A & operator=(const A& b) {
        delete [] x;
        x = new double[b.n];
        n = b.n;
        for (size_t i = 0; i < n; i++) x[i] = b.x[i];
        cout << "copy =\n";
        return *this;
    }
    const A & operator=(A&& b) {
        int m = n; n = b.n; b.n = m;
        double *p = x; x = b.x; b.x = p;
        cout << "move =\n"; return *this;
    }
//    A operator+(const A &b) const {
//        A s(n);
//        for (size_t i = 0; i < n && i < b.n; i++) s[i] = x[i] + b.x[i];
//        return s;
//    }

friend  A operator+(const A &a, const A &b);
friend  ostream & operator<<(ostream &os, const A &a);
};

A operator+(const A &a, const A &b)
{
        A s(a.n);
        for (size_t i = 0; i < a.n && i < b.n; i++) s[i] = a[i] + b[i];
        return s;
}
ostream & operator<<(ostream &os, const A &a)
{
    os << "A: ";
    for (size_t i = 0; i < a.n; i++) os << " " << a[i];
    os << endl;
    return os;
}

int main()
{
    A a(3);
    a[0] = a[1] = a[2] = 1;
    cout << "a " << a;

    A b(3);
    b[0] = b[1] = b[2] = 2;
    cout << "b " << b;

    A c(a);
    cout << "c " << c;

    cout << "A d(move(a)) ------------\n";
    A d(move(a));
```

```
    cout << "d " << d;
    cout << "a " << a;

    cout << "c = b   ------------\n";
    c = b;
    cout << "b " << b;
    cout << "c " << c;

    cout << "b = c + d ------------\n";
    b = c + d;
    cout << "b " <<  b;
    cout << "c " <<  c;
    cout << "d " <<  d;

    cout << "c = move(b) ------------\n";
    c = move(b);
    cout << "b " <<  b;
    cout << "c " <<  c;

    cout << "A f(b + c); ------------\n";
    A f(b + c);
    cout << "f " <<  f;
    cout << "b " <<  b;
    cout << "c " <<  c;

    return 0;
}
```