```
#include <iostream>

// https://en.cppreference.com/w/cpp/types/remove_reference
template <typename T> struct remove_reference      { using type = T; };
template <typename T> struct remove_reference<T&>  { using type = T; };
template <typename T> struct remove_reference<T&&> { using type = T; };

template <typename T>
using remove_reference_t = typename remove_reference<T>::type;

// https://en.cppreference.com/w/cpp/utility/move
template <typename T>
remove_reference_t<T>&& move(T&& t)
{
        return static_cast<
                remove_reference_t<T>&&
        >(t);
}

// https://en.cppreference.com/w/cpp/utility/forward
template <typename T>
T&& forward(remove_reference_t<T>& t)
{
        return static_cast<T&&>(t);
}

// https://en.cppreference.com/w/cpp/algorithm/swap
template <typename T>
void swap(T& a, T&b)
{
        T t{move(a)};
        a = move(b);
        b = move(t);
}

// A simplified skeleton of
// https://en.cppreference.com/w/cpp/container/vector
template <typename T>
class vector
{
        size_t  _count;
        T*              _data;
public:
        vector() :
                _count{0},
                _data{new T[_count]}
        {
                std::cout << "vector::vector()" << std::endl;
        }

        ~vector()
        {
                delete[] _data;
                std::cout << "vector::~vector()" << std::endl;
        }

        // Copy semantics
        vector(const vector& rhs) :
                _count{rhs._count},
                _data{new T[_count]}
        {
                std::copy(rhs._data, rhs._data + rhs._count, _data);
                std::cout << "vector::vector(copy)" << std::endl;
        }

        vector& operator=(const vector& rhs)
        {
                std::cout << "vector::op=(copy)" << std::endl;
                if (this != &rhs)
```

```
                {
                        T* temp = new T[rhs._count];
                        std::copy(rhs._data, rhs._data + rhs._count, temp);
                        delete[] _data;
                        _data = temp;
                        _count = rhs._count;
                }
                return *this;
        }

        // Move semantics
        vector(vector&& rhs) :
                _count{rhs._count},
                _data{rhs._data}
        {
                rhs._count = 0;
                rhs._data = new T[rhs._count];
                std::cout << "vector::vector(move)" << std::endl;
        }

        vector& operator=(vector&& rhs)
        {
                std::cout << "vector::op=(move)" << std::endl;
                swap(_count, rhs._count);
                swap(_data, rhs._data);
                return *this;
        }
};

// Examples of perfect forwarding

template <typename T1, typename T2>
void assign(T1& a, T2&& b)
{
        a = forward<T2>(b);
}

template <typename T>
class UniquePtr
{
        T* _data;

        UniquePtr(T* data) : _data{data}
        {
        }
public:
        template <typename Arg>
        static UniquePtr create(Arg&& arg)
        {
                return UniquePtr{
                        new T{
                                forward<Arg>(arg)
                        }
                };
        }

        UniquePtr(const UniquePtr&) = delete;
        UniquePtr& operator=(const UniquePtr&) = delete;

        ~UniquePtr()
        {
                delete _data;
        }
};

// Use cases

int main()
{
```

```
        vector<int> v1;
        vector<int> v2{v1};
        vector<int> v3{std::move(v1)};

        v2 = v1;
        v3 = std::move(v2);
        assign(v2, v1);
        assign(v3, std::move(v2));

        using UP = UniquePtr< vector<int> >;
        UP u1 = UP::create(v1);
        UP u2 = UP::create(std::move(v1));

        (void)v3;
        (void)u1;
        (void)u2;
}
```