C++11 Optimizations

Piotr Padlewski

SFBay Association of C/C++ Users 09.09.2015

Plan

- std::move() i rvalue references
- Universal references
- Noexcept
- Data structures optimizations

What is rvalue?

- Temporary objects - unnamed
- “No longer needed objects”
- Everything that doesn’t have an address

rvalue references

```
std::string foo(std::string&& s)
{
    return s;
}
int main()
{
    foo(std::string("SFBay Association of"));
    foo("C/C++");
    foo(foo("Users"));
}
```

rvalue references

```
std::string foo(std::string&& s)
{
    return s;
}

int main()
{
    std::string a(" :( ");
    foo(a);  //error: cannot bind std::string lvalue to std::string&&
}
```

std::move()

```
std::string foo(std::string&& s)
{
    return s;
}

int main()
{
    std::string a(" :) ");
    foo(std::move(a));  // fine
}
```

std::move()

```
struct Foo {
    Foo(std::string&& temp) : a_(temp)   // Copies temp to a_!
    {
    }
    std::string a_;
};
int main() {
    std::string c("42");
    Foo foo(std::move(c));
    Foo foo2(“42”);
}
```

std::move()

```
struct Foo {
    Foo(std::string&& temp) : a_(std::move(temp))  // fine, everything is moved
    {
    }
    std::string a_;
};
int main() {
    std::string c("42");
    Foo foo(std::move(c));
    Foo foo2(“42”);
}
```

std::move()

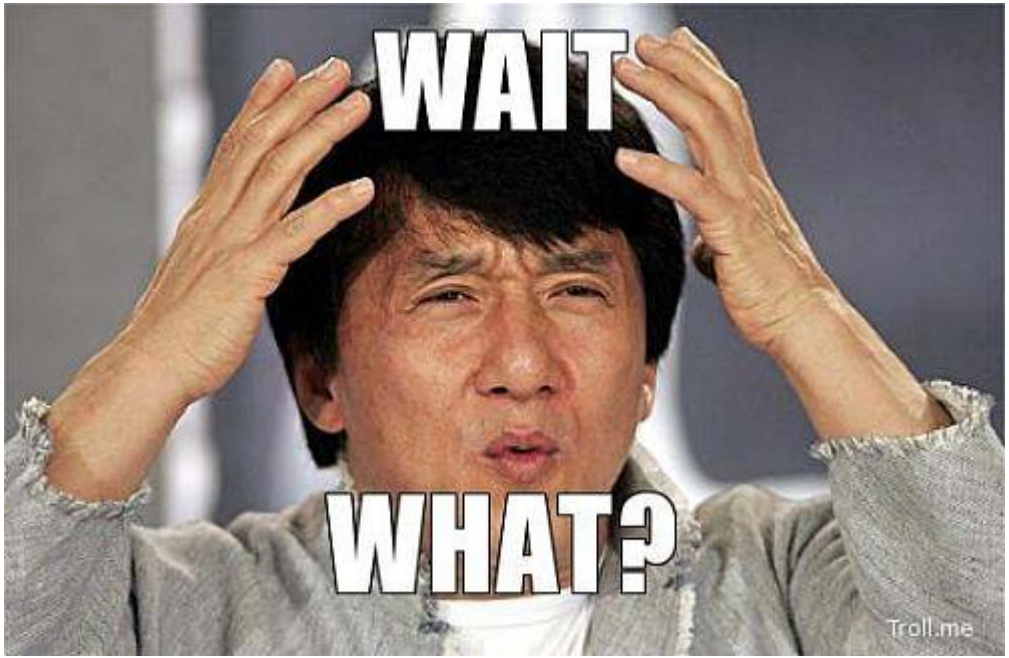
```
template< class T >
typename std::remove_reference<T>::type&& move( T&& t )
{
    return static_cast<remove_reference<decltype(arg)>::type&&>(arg);
}
```

std::move() **doesn’t move** any objects!

It only change expression to rvalue.

Universal references

```
template <typename T>
void foo(T&& t)
{
}
int main()
{
    foo("123");

    std::string a("abc");
    foo(a);
}
```



Universal references

```
template <typename T>
void foo(T&& t)
{
}
int main()
{
    foo(std::string("123")); // calls foo(std::string&&)

    std::string a("abc");
    foo(a);                  //cals foo(std::string&)
}
```

- ~~A& & becomes A&~~
- ~~A& && becomes A&~~
- ~~A&& & becomes A&~~
- ~~A&& && becomes A&&~~

- A & becomes A&
- A& && becomes A&
- A && becomes A&&

perfect forwarding

```
struct Foo {
    template <typename T>
    Foo(T&& t) : s_(std::forward<T> (t))
    {
    }
    std::string s_;
};
int main() {
    Foo f("fdafds");

    std::string b("fdas");
    Foo f2(b);
}
```

```
template<class S>
S&& forward(typename remove_reference<S>::type& a) noexcept
{
        return static_cast<S&&>(a);
}
```

std::forward “restore/preserves” expression type

string&& -> string&&

string& -> string&

Universal references

```
struct ConstExtraParamArgs
{
        typedef std::string                                 ExtraParamValueType;
        typedef std::vector<ExtraParamValueType>    ExtraParamValuesContainer;

    ConstExtraParamArgs(ExtraParamValueType key = "",
                        std::string separator = "",
                        std::string recursiveOtherName = "",
                        size_t limit = 0,
                        bool recursive = false,
                        ExtraParamValuesContainer predefinedValues=ExtraParamValuesContainer());
      … // The same members as in ctor
};
```

Universal references

```
class ExtraParamArgs {
    typedef const ConstExtraParamArgs                    PointerElementType;
public:
    typedef std::shared_ptr<PointerElementType>        ConstExtraParamArgsPtr;
    ExtraParamArgs() : index(0),
            constExtraParamArgsPtr_(std::make_shared<PointerElementType>())
    { }
    template <typename... Args>
    ExtraParamArgs(size_t index, Args&&... args)
        : index(index),
          constExtraParamArgsPtr_(std::make_shared<PointerElementType>(std::forward<Args>(args)...))
    { }
    size_t index;
private:
    ConstExtraParamArgsPtr constExtraParamArgsPtr_;
};
```

Universal reference

Universal references doesn't always work.

```
foo({0, 1, 2})
```

```
5.cc:11:18: error: no matching function for call to 'foo(<brace-enclosed initializer list>)'
    foo({0, 1, 2});
                ^
5.cc:6:6: note:   template argument deduction/substitution failed:
5.cc:11:18: note:   couldn't deduce template parameter 'T'
   foo({0, 1, 2})
```

But this will compile:

```
auto v = {0, 1, 2};  // type v to std::initializer_list<int>
foo(std::move(v));
```

```
template <typename T>
void foo(T&& t) {
    std::vector<int> v(t);
}
```

Defence programming

```
std::string str("Find the bug”);
tokenizer<> tok(str.substr(0, 5));
for(tokenizer<>::iterator beg=tok.begin(); beg!=tok.end();++beg){
    cout << *beg << "\n";
}
```

Because boost::tokenizer parse expression lazily, it won’t work with temporary objects.

Defence programming

```
struct tokenizer {
  template <typename Container>
  tokenizer(const Container&& con) : member(con) {
    static_assert( !std::is_rvalue_reference<Container&&>::value,
                   "It's invalid to construct tokenizer from rvalue");
  }
  const std::string &member;
};
```

Defence programming

```
jajko-wielkanocne.cc:8:5: error: static_assert failed "It's invalid to
construct tokenizer from rvalue"
    static_assert( !std::is_rvalue_reference<Container&&>::value,
    ^              ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

jajko-wielkanocne.cc:15:13: note: in instantiation of function template
specialization 'tokenizer::tokenizer<std::basic_string<char> >' requested here
  tokenizer t(a.substr(0, 5));
            ^
```

Reference vs value

```
struct Foo {
    Foo(std::string s) : s_(std::move(s))
    {
    }

    std::string s_;
};
int main() {
    Foo f("fdafds"); // 0 copies
    std::string b("fdas");
    Foo f2(b); // 1 copy
    Foo x(std::move(b)); //0 copies
}
```

If the type of the passed object

- has move ctor and
- copy is inevitable

Then you should pass arguments by value.

In the other case, you probably want to pass it by normal reference.

std::move()

```
std::vector<std::string> v;
void make_something(const std::string s)
{
      //stuff
    v.push_back(std::move(s));
}


int main() {
    make_something("123");
}
```

Will compile and cause extra copy.

Choosing from:
push_back(const string&)
push_back(string&&)
Will choose the first one and because

string doesn't have ctor like this:
string(const string&&)
It will led to normal copy.

to move or not to move?

```
std::vector<std::string> foo(std::string s)
{
  std::vector <std::string> v;
  v.push_back(std::move(s));
  return v;
}
int main()
{
  auto v = foo("hmm");
}
```

URVO i NRVO

(Named/Unnamed) **Return Value Optimization** is widely used optimization, aimed to awoid copy of return value.

It consist of creating object in the place of the object on callee site.

In general, the C++ standard allows a compiler to perform any optimization, provided the resulting executable exhibits the same observable behaviour as if all the requirements of the standard have been fulfilled.

RVO is one of exception of the “**as-if”** rule.

URVO i NRVO - how it's made

```
struct Foo {
    Foo(int a, int b);
    void some_method();
};
void do_something_with(Foo&);
Foo rbv() {
    Foo y = Foo(42, 73);
    y.some_method();
    do_something_with(y);
    return y;
}
void caller() {
   Foo x = rbv();
}
```

```
// Pseudo-code
void Foo_ctor(Foo* this, int a, int b) {
  // ...
}
void caller() {
    struct Foo x;
    // Note: x is not initialized here!
    rbv(&x);
}
```

URVO i NRVO - how it's made

```
// Pseudo-code
void Foo_ctor(Foo* this, int a, int b) {
  // ...
}
void caller() {
    struct Foo x;
    // Note: x is not initialized here!
    rbv(&x);
}
```

```
// Pseudo-code
void rbv(void* put_result_here) {
    Foo_ctor((Foo*)put_result_here, 42, 73);
    Foo_some_method(*(Foo*)put_result_here);
    do_something_with((Foo*)put_result_here);

    return;
}
```

URVO i NRVO ex.

```
struct Foo {
        Foo() {
            std::cout << "Foo()" << std::endl;
        }
        Foo(const Foo&) {
            std::cout << "Foo(const Foo&)" << std::endl;
        }
        Foo(Foo&&) {
            std::cout << "Foo(Foo&&)" << std::endl;
        }
        ~Foo() {
            std::cout << "~Foo()" << std::endl;
        }
        void someMethod() {
            std::cout << "some method" << std::endl;
        }
    };
```

```
Foo bar(bool p) {
    return Foo(); //URVO
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

Foo()
end
~Foo()

to move or not to move?

```
Foo bar(bool p) {
    return Foo(); //URVO
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

```
Foo()
end
~Foo()
```

```
Foo bar(bool p) {
    return std::move(Foo()); //no URVO!
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

```
Foo()
Foo(Foo&&)
~Foo()
end
~Foo()
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    Foo a;
    a.someMethod();
    return a; //NRVO
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

```
Foo()
some method
end
~Foo()
```

```
Foo bar(bool p) {
    Foo a;
    a.someMethod();
    return std::move(a);
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

```
Foo()
some method
Foo(Foo&&)
~Foo()
end
~Foo()
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    Foo a;
    if (p)
        return a;
    else {
        a.someMethod();
        return a;
    }
}
```

```
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out: ./prog

```
Foo()
some method
end
~Foo()
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    if (p)
        return Foo();
    else {
        Foo a;
        a.someMethod();
        return a;
    }
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

runing: ./**prog**

gcc-4.8.2 out:

Foo()
some method
Foo(Foo&&)
~Foo()
end
~Foo()

clang-3.5 out:

Foo()
some method
end
~Foo()

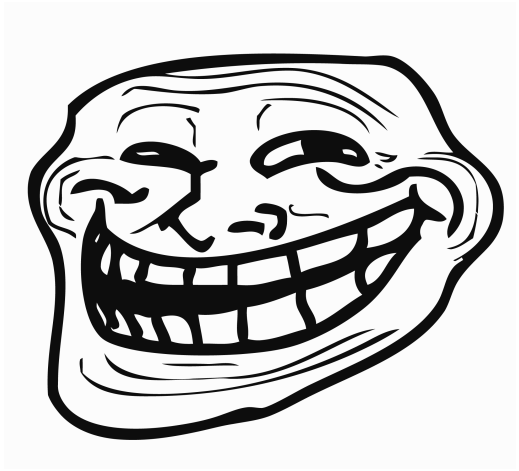
URVO i NRVO ex.

When the criteria for elision of a copy operation are met or would be met save for the fact that the source object is a function parameter, and the object to be copied is designated by an lvalue, overload resolution to select the constructor for the copy is first performed as if the object were designated by an rvalue.

URVO i NRVO ex.

```
    // We add this 2 ctors
    Foo(const std::initializer_list<int>&) {
        cout << "Foo(initializer_list &)" << endl;
    }
    Foo(std::initializer_list<int>&&) {
        cout << "Foo(initializer_list &&)" << endl;
    }

Foo bar(bool p) {
    return {};
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:
Foo()
end
~Foo()



URVO i NRVO ex.

```
Foo bar(bool p) {
    return {1, 2, 3};
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    cout << "end" << endl;
}
```

out:

```
Foo(initializer_list &&)
end
~Foo()
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    auto v = {1, 2, 3};
    return v;
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    cout << "end" << endl;
}
```

out:

```
Foo(initializer_list &)
end
~Foo()
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    auto v = {1, 2, 3};
    return std::move(v); // :(
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    cout << "end" << endl;
}
```

out:

```
Foo(initializer_list &&)
end
~Foo()
```

```
// cleaner
Foo bar(bool p) {
    auto v = {1, 2, 3};
    Foo f(std::move(v));
    return f;
}
```

URVO i NRVO ex.

```
Foo bar(bool p) {
    if (p)
        return {1, 2, 3};
    else {
        Foo a;
        return a;
    }
}
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

clang out: ./prog
Foo()
end
~Foo()

clang out: ./prog 123
Foo(initializer_list &&)
end
~Foo()

gcc out: ./prog 123
Foo(initializer_list &&)
end
~Foo()

gcc out: ./prog
Foo()
Foo(Foo&&)
~Foo()
end
~Foo()

URVO i NRVO ex.

```
Foo bar(bool p) {
    Foo a;
    a.someMethod();
    Foo b;
    if (p)
        return b;
    else
        return a;
}
```

```
int main(int argc, char* argv[]) {
    Foo f = bar(argc > 1);
    std::cout << "end" << std::endl;
}
```

out:

```
Foo()
some method
Foo()
Foo(Foo&&)
~Foo()
~Foo()
end
~Foo()
```

URVO i NRVO summary

- Never use “return std::move(...)” - even if compiler will not be able to perform RVO, it will move it without your help, **unless:**
  - you return object of the different type than type function return type
- Try to return the same object in all returns.
- Make sure that returned type has move ctor - RVO like programmers is not perfect…

noexcept

```
void maybe();
void foo() throw();
void bar() noexcept;
```

The difference between unwinding the call stack and possibly unwinding it has a surprisingly large impact on code generation. In a noexcept function, optimizers need not keep the runtime stack in an unwindable state if an exception would propagate out of the function, nor must they ensure that objects in a noexcept function are destroyed in the inverse order of construction should an exception leave the function. The result is more opportunities for optimization, not only within the body of a noexcept function, but also at sites where the function is called. Such flexibility is present only for noexcept functions. Functions with “throw()” exception specifications lack it, as do functions with no exception specification at all.

> An implementation shall not reject an expression merely because when executed it throws or might throw an exception that the containing function does not allow.

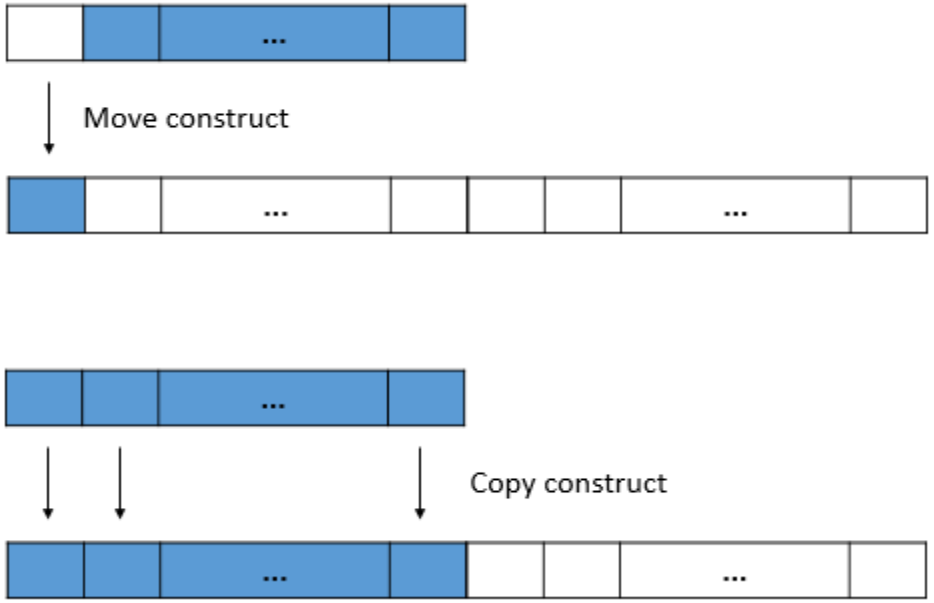
noexcept

```
struct Foo {
    Foo() {
        cout << "Foo()" << endl;
    }
    Foo(const Foo&) {
        cout << "Foo(const Foo&)" << endl;
    }
    Foo(Foo&&) {
        cout << "Foo(Foo&&)" << endl;
    }
    ~Foo() {
        cout << "~Foo()" << endl;
    }
};
```

```
int main()
{
    std::vector<Foo> v;
    for (int i = 0 ; i < 3; i++)   {
        cout << "inserting" << endl;
        v.emplace_back();
    }
    cout << "end" << endl;
}
```

```
inserting
Foo()                #1
inserting
Foo()                #2
Foo(const Foo&)      #1'
~Foo()               #1
inserting
Foo()                #3
Foo(const Foo&)      #1''
Foo(const Foo&)      #2''
~Foo()               #1'
~Foo()               #2'
koniec
~Foo()               #1''
~Foo()               #2''
~Foo()               #3''
```

noexcept



move_if_noexcept()

```
template <class T>
  typename conditional < is_nothrow_move_constructible<T>::value ||
                         !is_copy_constructible<T>::value,
                         T&&, const T& >::type
 move_if_noexcept(T& arg) noexcept;
```

Casts to rvalue if one of condition is true

- move constructor is noexcept
- there is no copy constructor

noexcept

```
struct Foo {
    Foo() {
        cout << "Foo()" << endl;
    }
    Foo(const Foo&) {
        cout << "Foo(const Foo&)" << endl;
    }
    Foo(Foo&&) noexcept {
        cout << "Foo(Foo&&)" << endl;
    }
    ~Foo() {
        cout << "~Foo()" << endl;
    }
};
```

```
int main()
{
    std::vector<Foo> v;
    for (int i = 0 ; i < 3; i++)   {
        cout << "inserting " << endl;
        v.emplace_back();
    }
    cout << "end" << endl;
}
```

```
inserting
Foo()                #1
inserting
Foo()                #2
Foo(Foo&&)           #1'
~Foo()               #1
inserting
Foo()                #3
Foo(Foo&&)           #1''
Foo(Foo&&)           #2''
~Foo()               #1'
~Foo()               #2'
koniec
~Foo()               #1''
~Foo()               #2''
~Foo()               #3''
```

noexcept

```
struct Foo {
    Foo() {
        cout << "Foo()" << endl;
    }
    Foo(const Foo&) {
        cout << "Foo(const Foo&)" <<
endl;
    }
    Foo(Foo&&) = default;
    ~Foo() {
        cout << "~Foo()" << std::endl;
    }
};
```

An inheriting constructor (12.9) and an implicitly declared special member function (Clause 12) have an *exception-specification*. If `f` is an inheriting constructor or an implicitly declared default constructor, copy constructor, **move constructor**, destructor, copy assignment operator, or move assignment operator, its implicit *exception-specification* specifies the type-id `T` if and only if `T` is allowed by the *exception-specification* of a function directly invoked by `f`'s implicit definition; `f` allows all exceptions if any function it directly invokes allows all exceptions, and `f` has the *exception-specification* `noexcept(true)` if every function it directly invokes allows no exceptions.

noexcept(expresion)

```
template <class T1, class T2>
struct pair {
    void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                                noexcept(swap(second, p.second)));
};
```

```
std::cout << std::boolalpha << noexcept(Foo(std::move(f))) << std::endl;
```

noexcept summary

- Use noexcept for documentation,
- Generate default ctors with “ = default”,
- If you define own constructor, always mark it with noexcept (unless they throw).
- Always use noexcept instead of throw()

Never use noexcept when the function may throw (f.e. with mem allocation), or if it’s unknow that function will never throw in the future.

Small digression

What will happen?

```
std::vector<int> w(42);
w = std::move(w);
```

Undefined Behaviour!

moving containers

```
std::vector<std::string> data;
std::vector<std::string> cache;
    // some inserting to both
std::copy(cache.begin(), cache.end(), std::back_inserter(data));
data.insert(data.end(), cache.begin(), cache.end());

std::move(cache.begin(), cache.end(), std::back_inserter(data));

data.insert(data.end(),
                std::make_move_iterator(cache.begin()),
                std::make_move_iterator(cache.end()));
```

Optimizations without sense

- inline on your own - **because call is so expensive**
- ++i instead i++ - **copy elimination**
- bit shifting instead of multiplication/division/modulo
- Extracting end() to value (maybe)
- Using **register** and **restrict**
- Using weird type_traits to change function signature from T& to T for POD types

Clang IMBA

```
int64_t getValue(int n) {
    int64_t result = 0;
    for (int i = 1 ; i <= n ; i++)
            result += i;
    return result;
}
```

```
int main(int argc, char* argv[]) {
    assert(argc == 3);
    int n = atoi(argv[1]);
    int64_t value = strtoll(argv[2], NULL, 10);
    for (int i = 1; i <= n ; i++) {
        if (getValue(i) == value)
            std::cout << i << std::endl;
    }
}
```

./eq 1000000000 5000000005000000000
clang ~ 2s
g++ ~ 42 years

set vs unordered_set

set vs unordered_set

sorted data and less memory vs speed

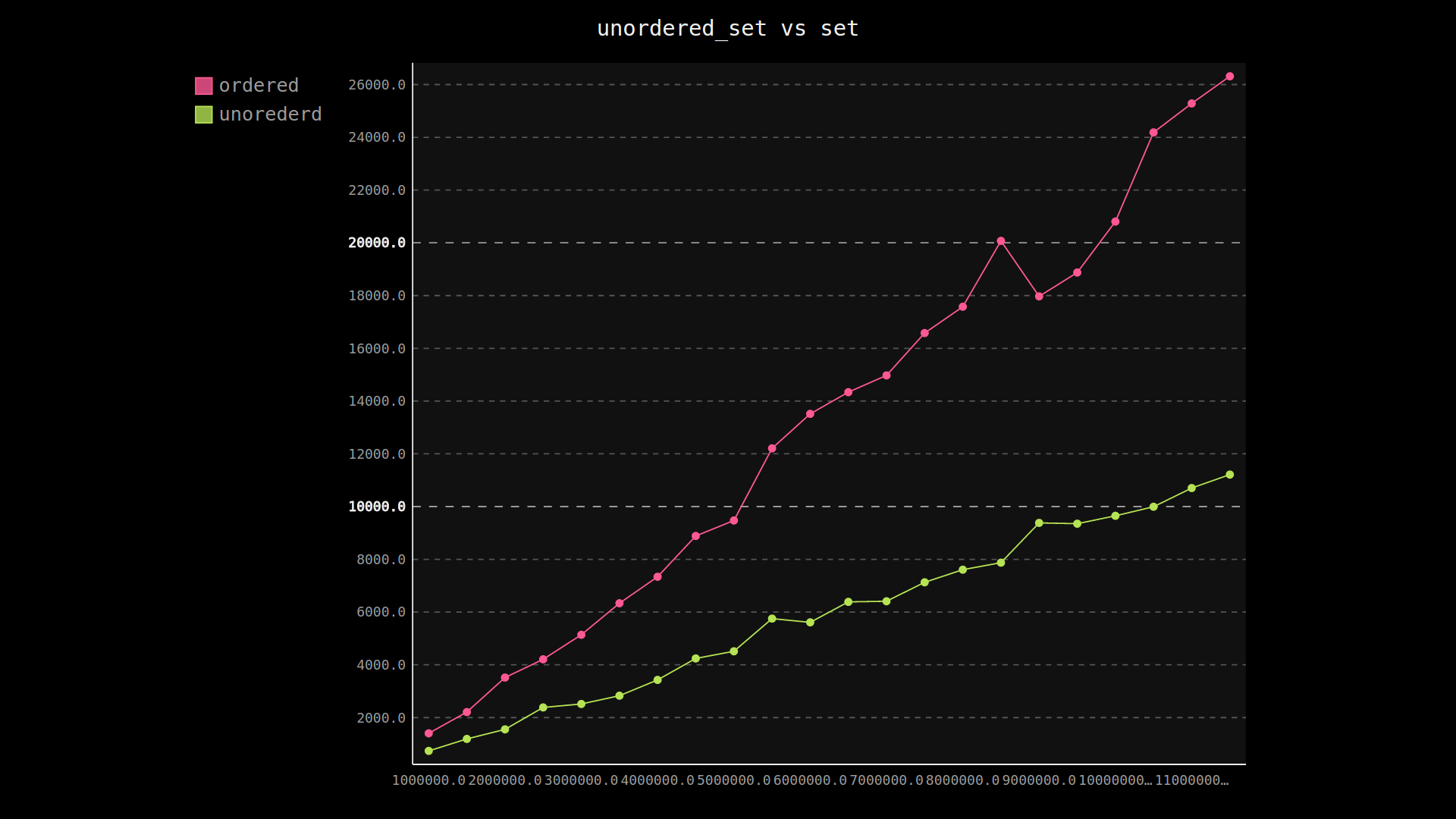
set vs unordered_set

```
int64_t benchSet(int size)
{
    std::set<int64_t> secior;
    for (int i = 0 ; i < size ; i++)
        secior.insert(mt());

    int64_t result = 0;
    for (auto& entry : secior)
        result += entry;
    return result;
}
```

```
random_device rd;
mt19937 mt(rd());
```

set vs unordered_set

```
int64_t benchUnorderedSet(int size) {
    std::unordered_set<int64_t> secior;
    for (int i = 0 ; i < size ; i++)
        secior.insert(mt());

    std::vector <int64_t> v(secior.begin(), secior.end());
    std::sort(v.begin(), v.end());
    int64_t result = 0;
    for (auto& entry : v)
        result += entry;
    return result;
}
```

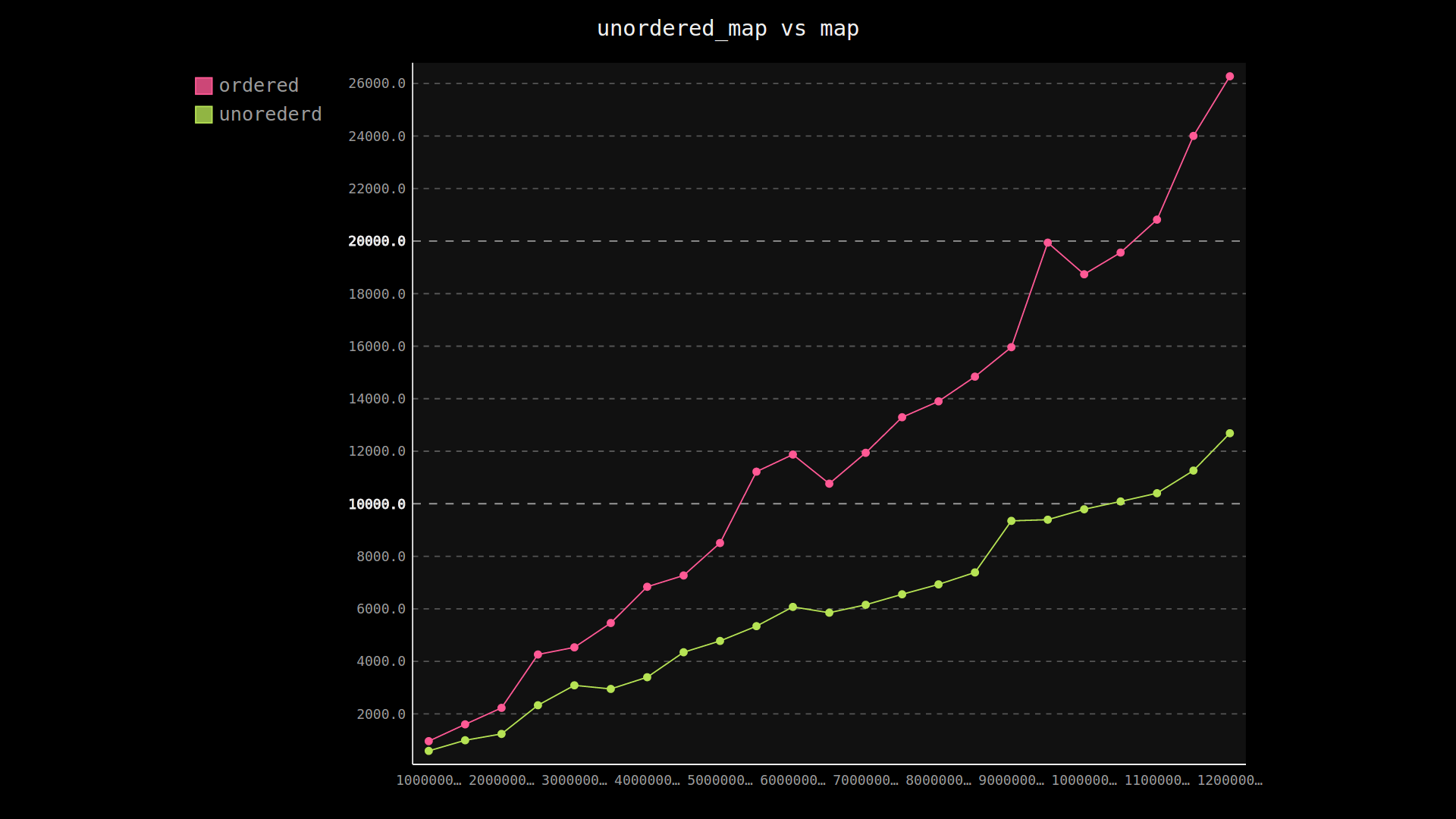

unordered_set vs set
ordered
unorederd
26000.0
24000.0
22000.0
20000.0
18000.0
16000.0
14000.0
12000.0
10000.0
8000.0
6000.0
4000.0
2000.0
1000000.0
2000000.0
3000000.0
4000000.0
5000000.0
6000000.0
7000000.0
8000000.0
9000000.0
10000000…
11000000…

map vs unordered_map

```
int64_t benchMap(int size) {
    std::map<int64_t, int64_t> mapcior;
    for (int i = 0 ; i < size ; i++)
        mapcior[mt()] = i;

    int64_t result = 0;
    for (auto& entry : mapcior)
        result += entry.second;
    return result;
}
```

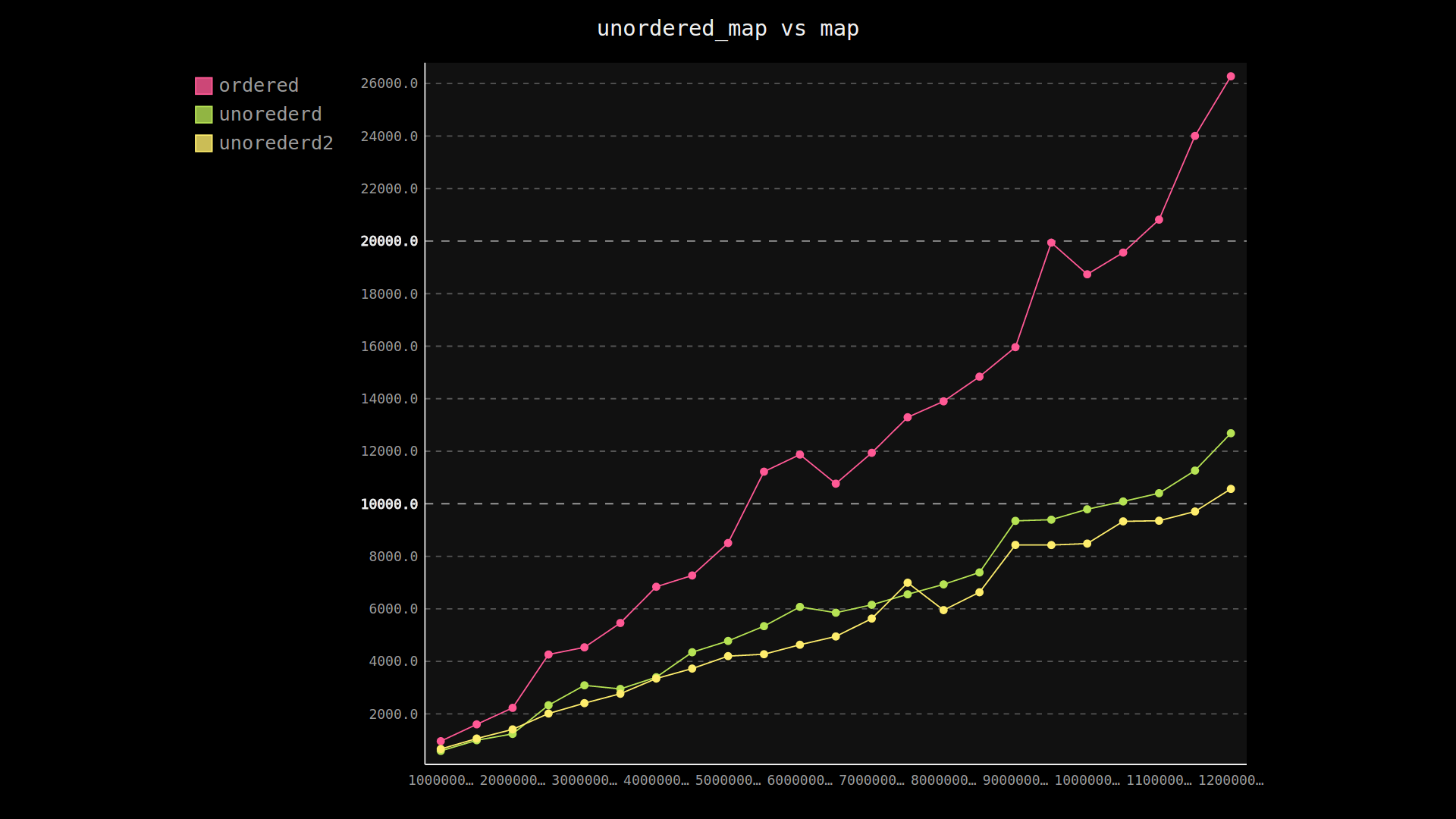
map vs unordered_map

```
int64_t benchUnorderdMap(int size) {
    std::unordered_map<int64_t, int64_t> mapcior;
    for (int i = 0 ; i < size ; i++)
        mapcior[mt()] = i;

    std::vector <std::pair<int64_t, int64_t> > v(mapcior.begin(), mapcior.end());
    std::sort(v.begin(), v.end());
    int64_t result = 0;
    for (auto& entry : v)
        result += entry.second;
    return result;
}
```


unordered_map vs map
ordered
unorederd
26000.0
24000.0
22000.0
20000.0
18000.0
16000.0
14000.0
12000.0
10000.0
8000.0
6000.0
4000.0
2000.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…
1200000…

map vs unordered_map

```
int64_t benchUnorderdMap(int size) {
    std::unordered_map<int64_t, int64_t> mapcior;
    for (int i = 0 ; i < size ; i++)
        mapcior[mt()] = i;

    std::vector <int64_t> v;
    v.reserve(mapcior.size());
    for (auto& entry : mapcior)
        v.push_back(entry.second);

    std::sort(v.begin(), v.end());
    int64_t result = 0;
    for (int64_t value : v)
        result += value;
    return result;
}
```


unordered_map vs map
ordered
unorederd
unorederd2
26000.0
24000.0
22000.0
20000.0
18000.0
16000.0
14000.0
12000.0
10000.0
8000.0
6000.0
4000.0
2000.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…
1200000…

Back to the copies

```
template <typename SetType>
struct ListJoinSets
{
    ListJoinSets(const vector<set<SetType> >& sets)
        : sets_(sets) { ... }

private:
    ...
    vector<set<SetType> >sets_;
};
```

```
template <typename SetElementType>
struct ListJoinSets
{
    ListJoinSets(vector<set<SetType> >& sets)
        : sets_(sets)
     ...
private:
    ...
    vector<set<SetElementType> >   &sets_;
};
```

Back to the copies

Potyczki algorytmiczne 2014

Fiolki

Rezultat

10/10 pkt **vs** **8/10 pkt**

References are so slow!

Copy vs Ref vs Move

```
template <typename Container>
int benchHelper(const Container& container, int64_t reads) { // (Container container, ..) in ver 2
    random_device rd;
    mt19937 mt(rd());
    int value = 0;
    while (reads > 0) {
        for (const auto& val: container) {
            if (get(val) <= mt()) value++;
            reads--;
            if (reads == 0) return value;
        }
    }
    return value;
}
```

```
int64_t get(int64_t val) { return val; }

template <typename T>
int64_t get(const T& t) { return t.second; }
```

Copy vs Ref vs Move

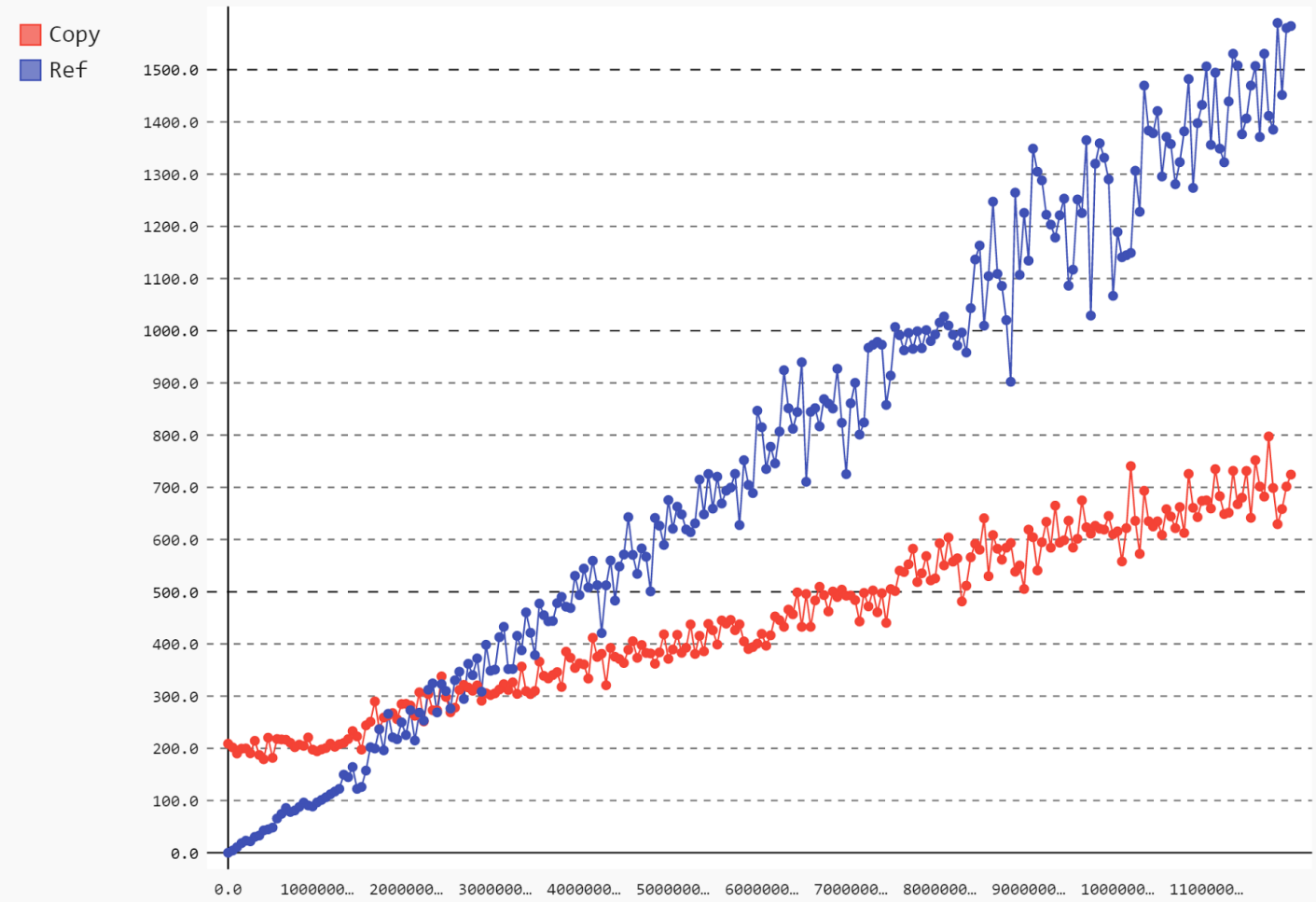
```
int bench(const set<int64_t> &secior, int64_t reads) {
    return benchHelper(secior, reads);
}
int bench(const unordered_set<int64_t> &secior, int64_t reads) {
    return benchHelper(secior, reads);
}
```

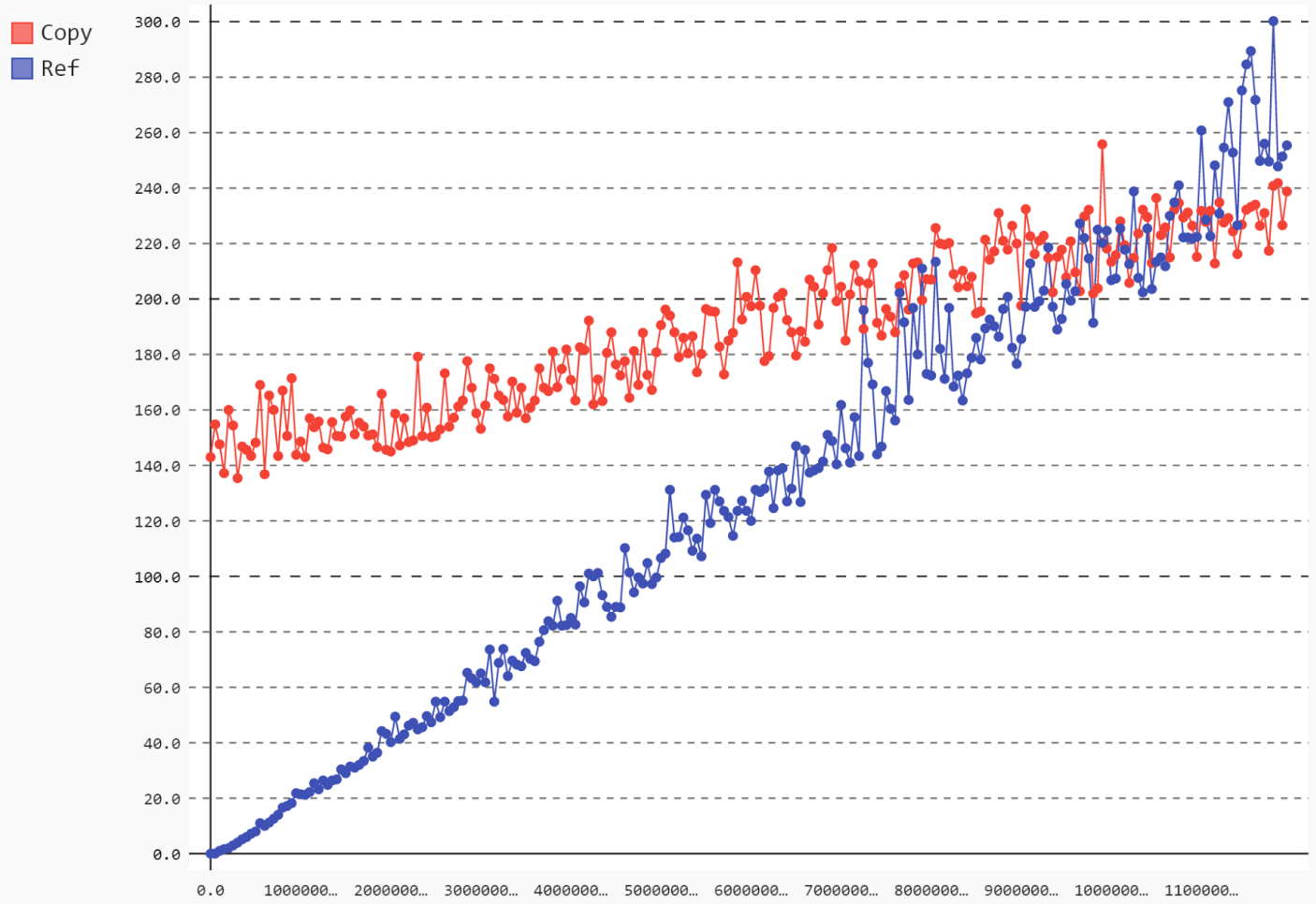
vs

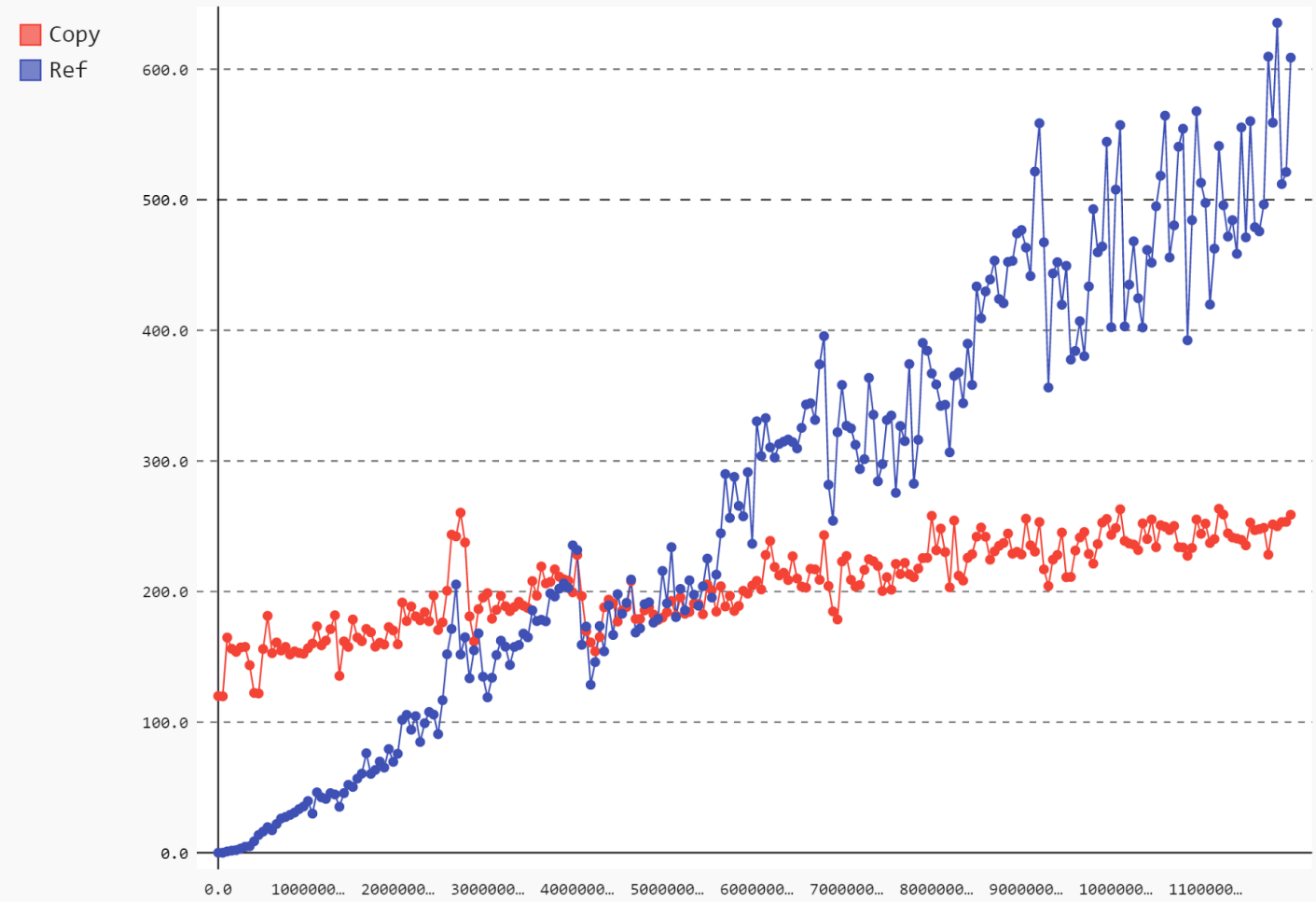
```
int bench(std::set<int64_t> secior, int64_t reads) {
    return benchHelper(move(secior), reads);
}
int bench(std::unordered_set<int64_t> secior, int64_t reads) {
    return benchHelper(move(secior), reads);
}
```

Copy vs Ref vs Move

```
template <typename Container>
void benchSet(int size, int readsCount) {
    Container secior;
    for (int i = 0 ; i < size ; i++)
        secior.insert(mt());

    auto now = system_clock::now();
    bench(secior, readsCount);
    auto duration = chrono::duration_cast<chrono::milliseconds>(
                                            system_clock::now() - now).count();
    cout << duration << endl;
}
```

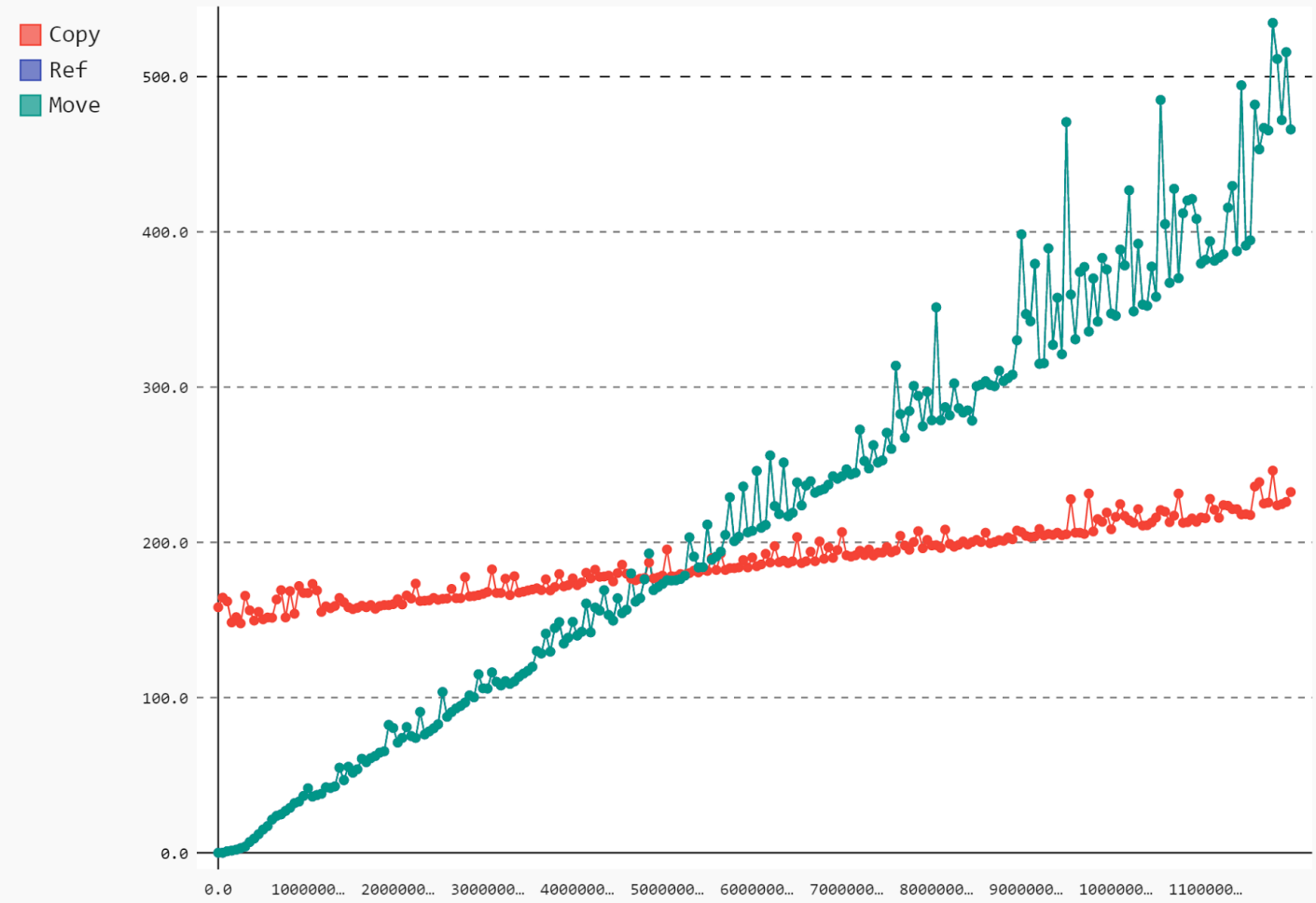

map Copy vs Ref: 1000000 elements
Copy
Ref
1500.0
1400.0
1300.0
1200.0
1100.0
1000.0
900.0
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

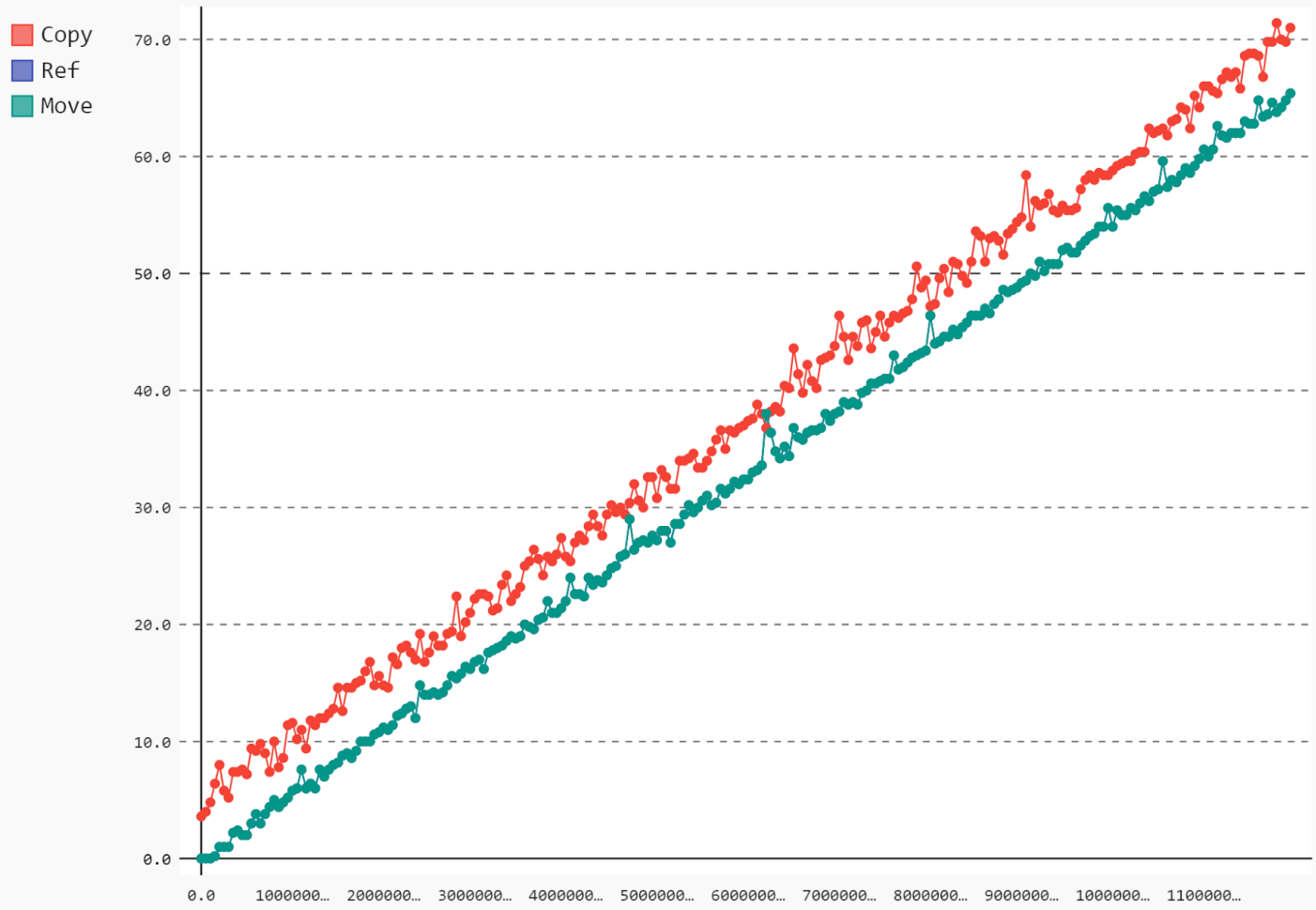

unordered_set Copy vs Ref: 1000000 elements
Copy
Ref
300.0
280.0
260.0
240.0
220.0
200.0
180.0
160.0
140.0
120.0
100.0
80.0
60.0
40.0
20.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…


unordered_map Copy vs Ref: 1000000 elements
Copy
Ref
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

Copy vs Ref vs Move

What will happen when we will move?


unordered_map Copy vs Ref: 1000000 elements
Copy
Ref
Move
500.0
400.0
300.0
200.0
100.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

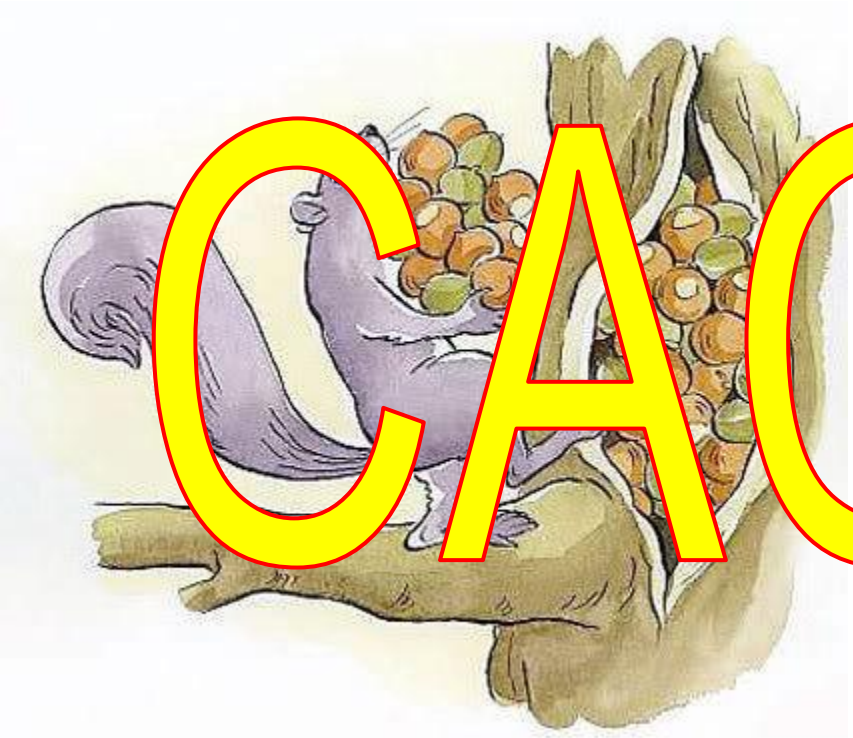

vector Copy vs Ref: 1000000 elements
Copy
Ref
Move
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

Copy vs Ref vs Move

References are so slow!

Conclusion: there is some magic during copy

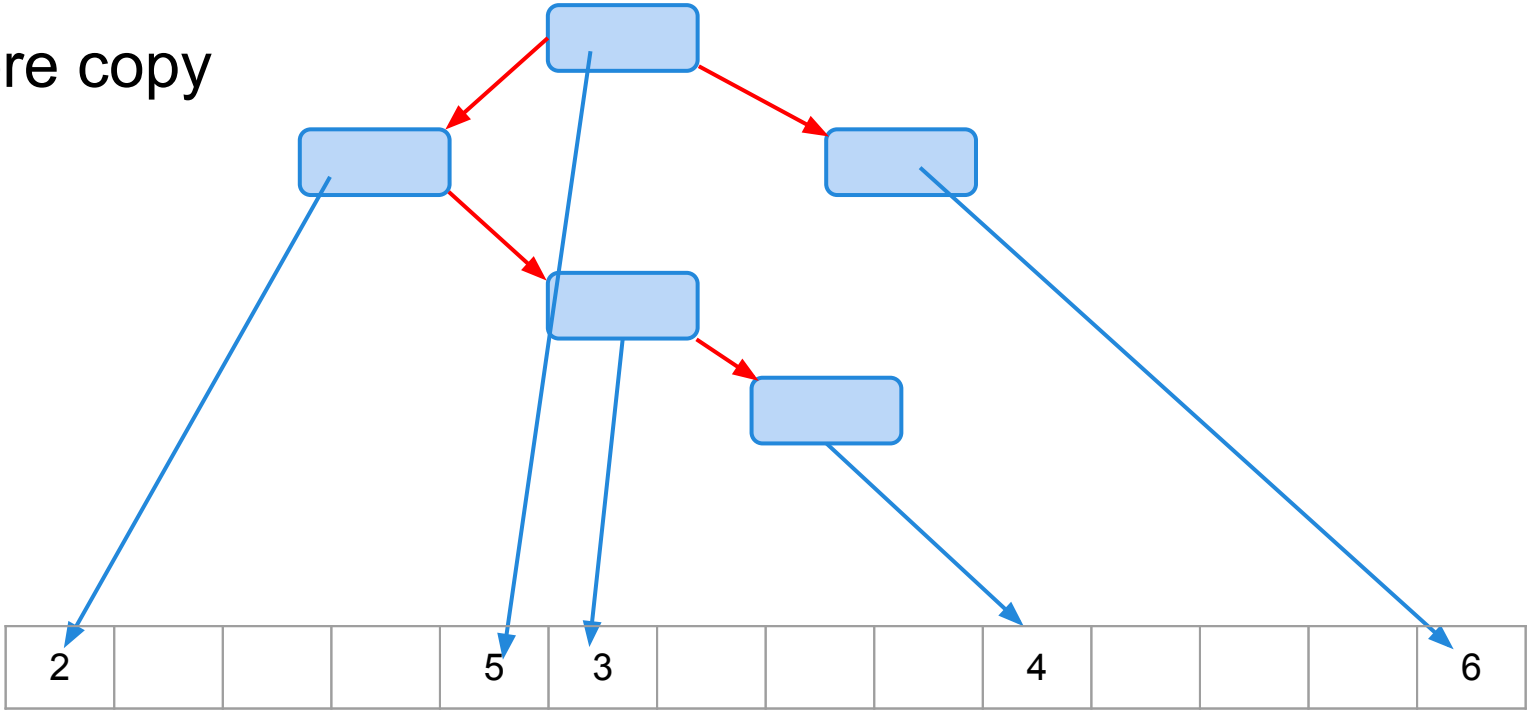
hint:



CACHE

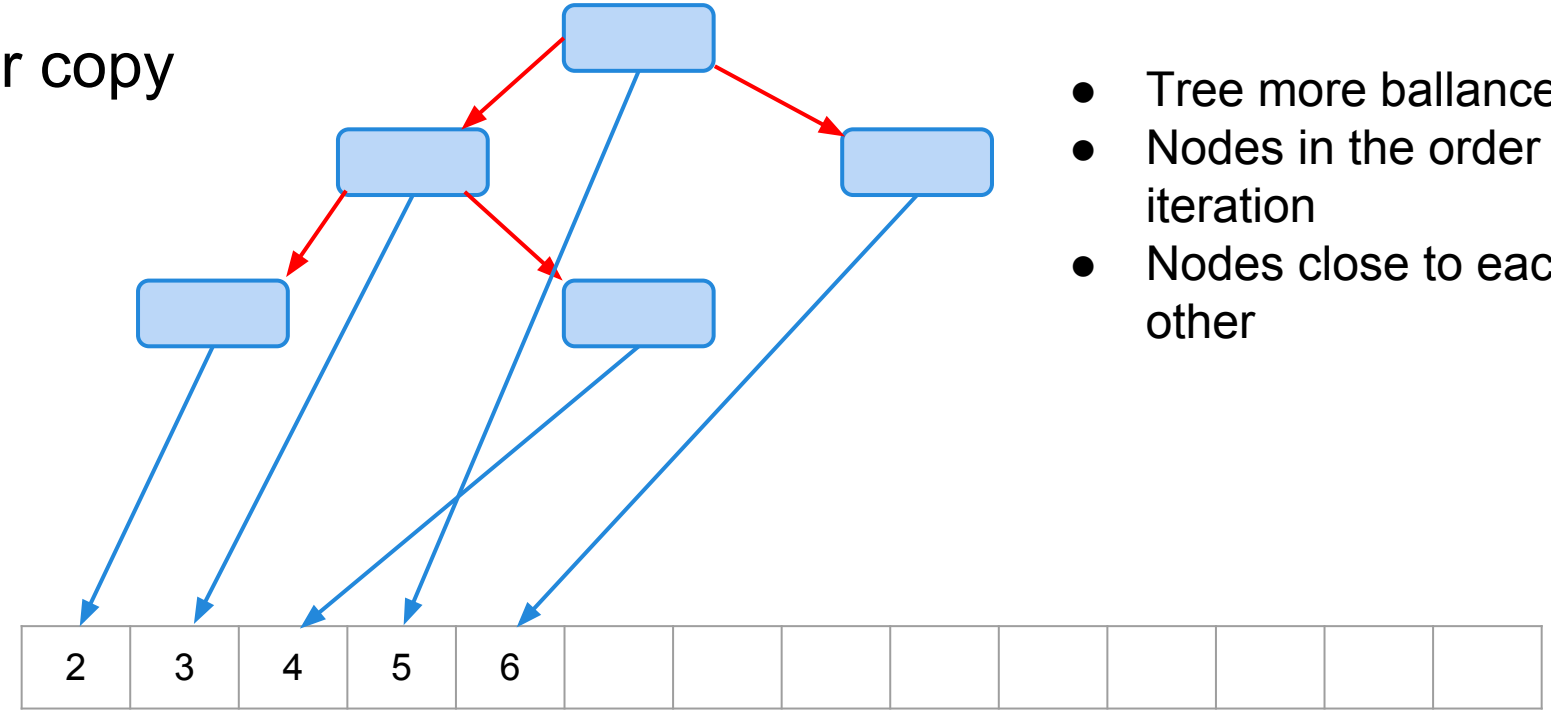
What is going on?

Before copy



What is going on?

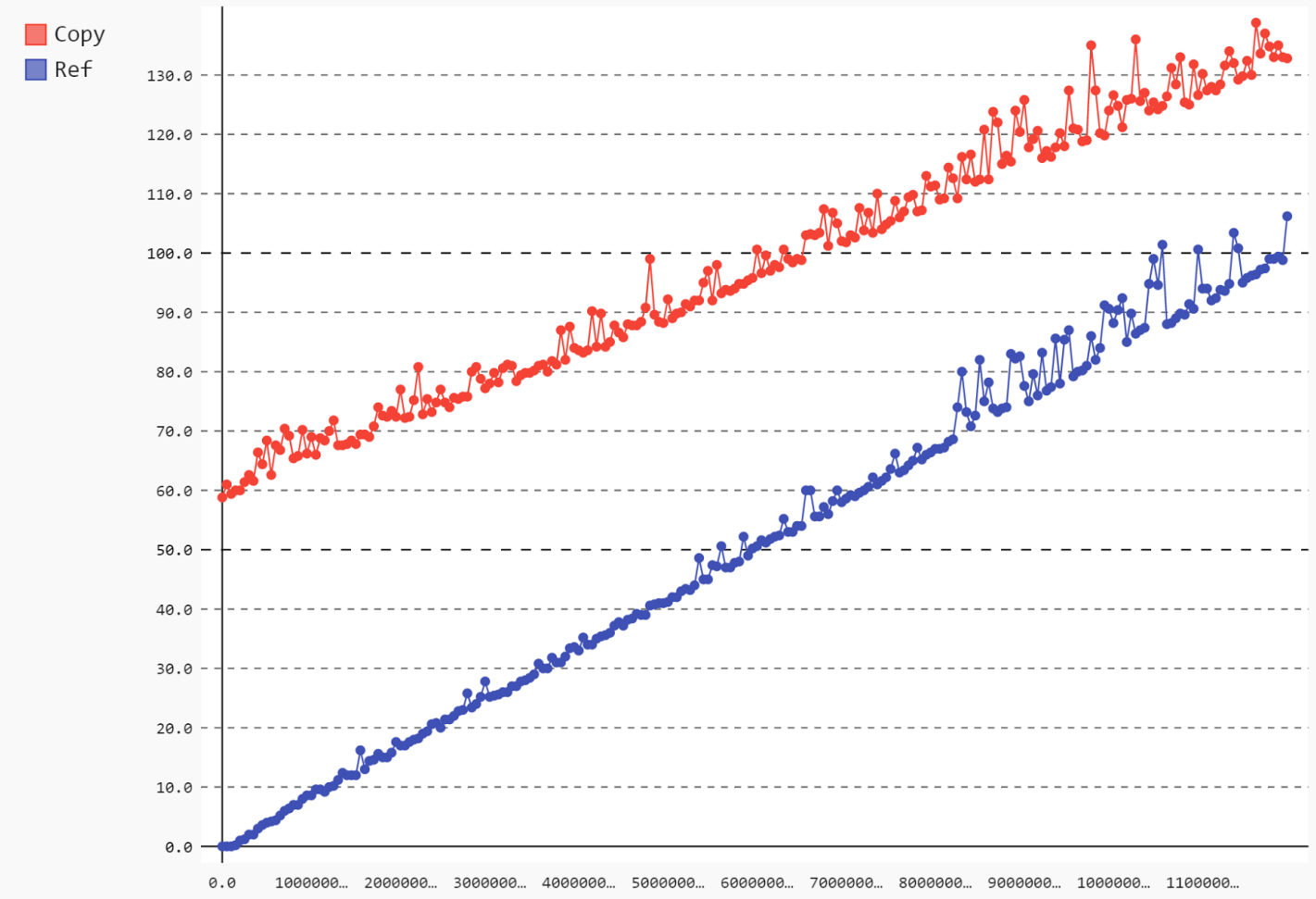
After copy



- Tree more ballanced
- Nodes in the order of iteration
- Nodes close to each other

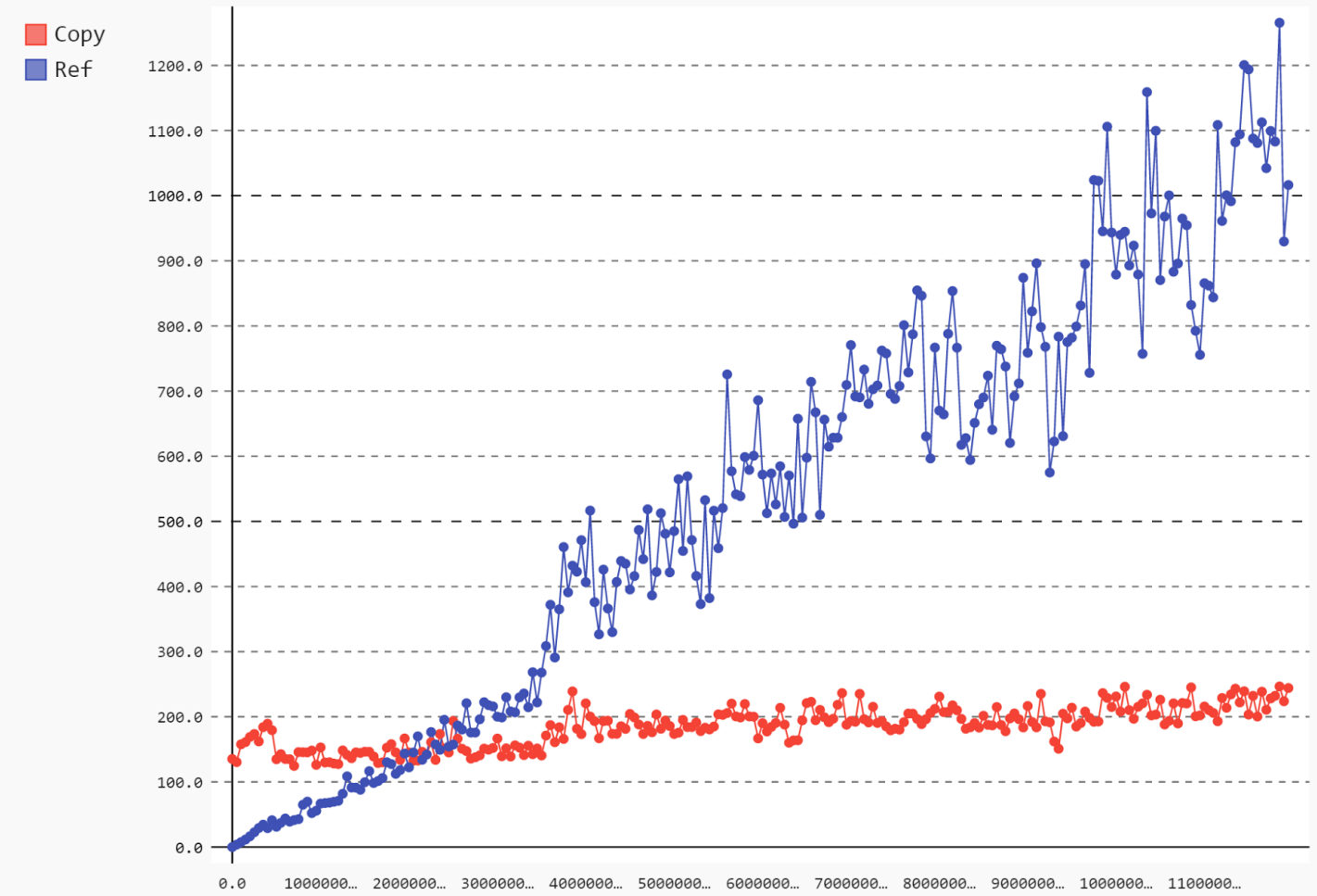
Copy vs Ref vs Move

```
void benchList(int size, int64_t readsCount) {
    list<int64_t> liscior;
    for (int i = 0 ; i < size ; i++)
        if (i % 2)
            liscior.push_back(mt());
        else
            liscior.push_front(mt());

    auto now = system_clock::now();
    bench(liscior, readsCount);
    auto duration = chrono::duration_cast<chrono::milliseconds>(
                                system_clock::now() - now).count();
    cout << duration << endl;
}
```


list Copy vs Ref: 1000000 elements
Copy
Ref
130.0
120.0
110.0
100.0
90.0
80.0
70.0
60.0
50.0
40.0
30.0
20.0
10.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

Copy vs Ref vs Move

```
void benchList(int size, int64_t readsCount) {
    list<int64_t> liscior;
    for (int i = 0 ; i < size ; i++)
        if (i % 2)
            liscior.push_back(mt());
        else
            liscior.push_front(mt());
    liscior.sort();
    auto now = system_clock::now();
    bench(liscior, readsCount);
    auto duration = chrono::duration_cast<chrono::milliseconds>(
                                system_clock::now() - now).count();
    cout << duration << endl;
}
```


list Copy vs Ref: 1000000 elements
Copy
Ref
1200.0
1100.0
1000.0
900.0
800.0
700.0
600.0
500.0
400.0
300.0
200.0
100.0
0.0
0.0
1000000…
2000000…
3000000…
4000000…
5000000…
6000000…
7000000…
8000000…
9000000…
1000000…
1100000…

Copy vs Ref vs Move

```
void benchList(int size, int64_t readsCount) {
    list<int64_t> liscior;
    for (int i = 0 ; i < size ; i++)
        if (i % 2)
            liscior.push_back(mt());
        else
            liscior.push_front(mt());
    liscior.sort(); // random_shuffle of memory
    auto now = system_clock::now();
    bench(liscior, readsCount);
    auto duration = chrono::duration_cast<chrono::milliseconds>(
                                system_clock::now() - now).count();
    cout << duration << endl;
}
```

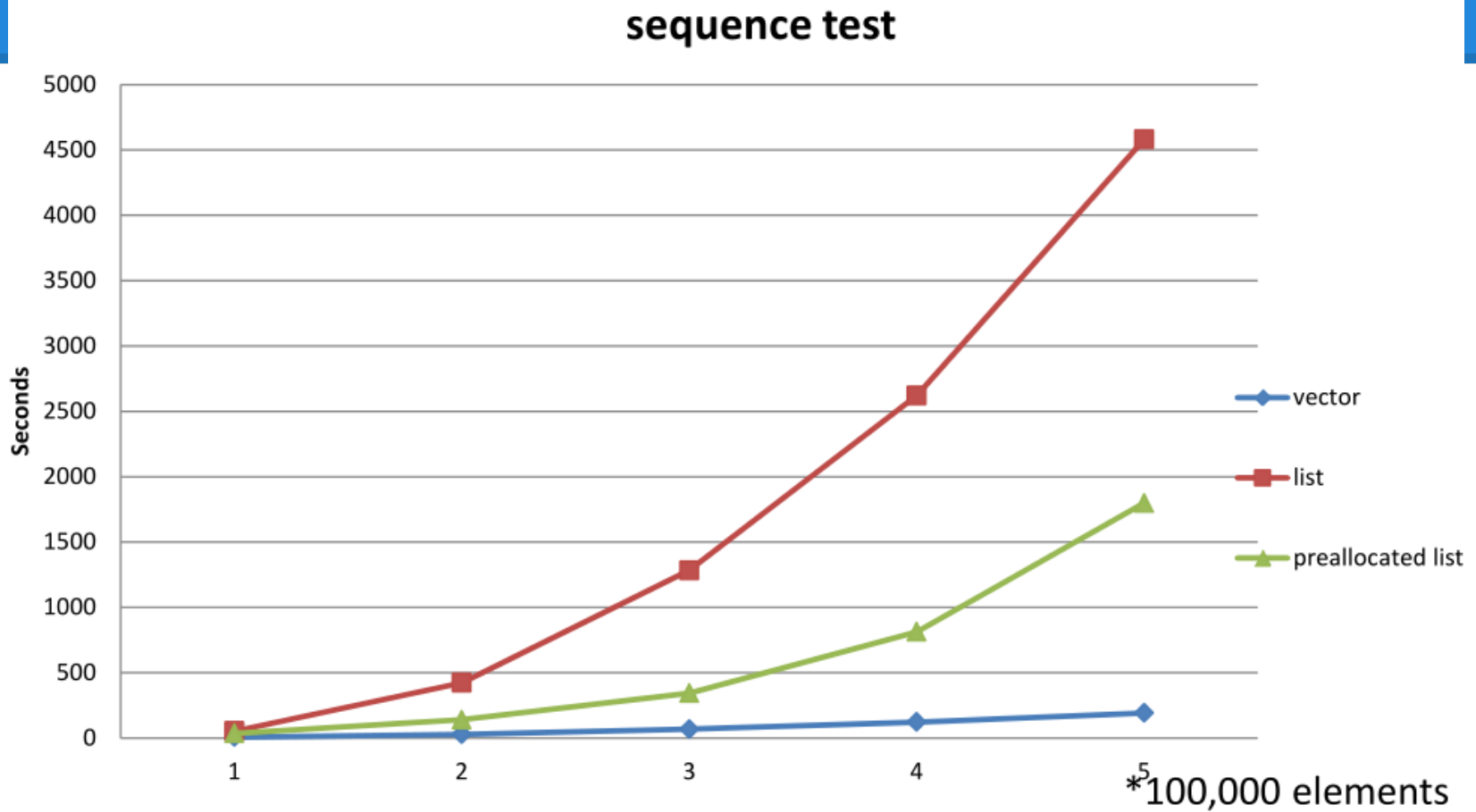
Copy vs Ref vs Move

Copy defragments memory

Objects that are pointed by near each other pointers are near each other in memory

fever cache misses, performance boost

Vector vs. List



Vector as default

Use vector when you need:

- Sequential unordered data
- “static set/map” - write, sort, find
- Small set/map with all operations
- Integer indexed array from range [0, 10^7]

```cpp
template <typename T>
class AutoStretchingVector : public std::vector<T> {
   typedef std::vector<T>  Self;
public:
   using typename Self::value_type;
   using typename Self::reference;
   using typename Self::const_reference;
   using typename Self::size_type;
   using typename Self::iterator;
   using typename Self::const_iterator;

   using Self::Self;

   reference get(size_type index)
   {
      if (Self::size() <= index)
         Self::resize(index + 1);
      return Self::operator[](index);
      assertyouDidntBreakIt_(); //have to call it to be instantiated
   }
private:
   static void assertyouDidntBreakIt_();
};
```

```cpp
template <typename T>
inline void AutoStretchingVector<T>::
assertyouDidntBreakIt_()
{
   static_assert(sizeof(std::vector<T>) ==
                 sizeof(AutoStretchingVector<T>),
                 "Don't add any data to this class!");
}
```

```cpp
AutoStretchingVector<
        AutoStretchingVector<
            AutoStretchingVector <int>
        > > matrix;

matrix.get(i).get(j).get(k) = 42;

get(get(get(matrix, i), j), k) = 42; // function instead of method
```

sources

http://thbecker.net/articles/rvalue_references/section_08.html

http://isocpp.org/wiki/faq/ctors#return-by-value-optimization

http://aristeia.com/EC++11-14/noexcept%202014-03-31.pdf

http://stackoverflow.com/questions/20517259/why-vector-access-operators-are-not-specified-as-noexcept

http://channel9.msdn.com/Events/GoingNative/GoingNative-2012/Keynote-Bjarne-Stroustrup-Cpp11-Style

Effective Modern c++ - Scott Meyers

https://youtu.be/hrXXM1eRURg - the same talk in polish

Copy vs Ref benchmark:

https://drive.google.com/file/d/0B72TmzNsY6Z8WnRYUFRhcDgtVmc/view?usp=sharing

Dziękuję za uwagę!