Relocation in C++

C++ standard proposal

Sébastien Bini

February 1, 2022

Contents

1 Revisions

Revision	Data	Changes
1	11/21/2021	First revision

2 Motivation

C++11 introduced rvalue-references, and along with it, move constructors and assignment operators. Although this comes with many advantages, it also comes with the burden of handling a new "moved-from" state in classes that can be moved.

2.1 Case 1: Classes that own system (or otherwise expensive) resources

Take the following class for instance:

```
class TempDir
{
public:
    // Creates a temporary directory
    // Throws if directory creation failed
    TempDir();
    // Removes the directory and recursively all contained files
    ~TempDir() noexcept;
    // Returns directory path
    std::string_view path() const;
};
```

This class holds a strong class invariant: *any instance owns a temporary directory, which will only be removed once the instance is destructed*.

Now what happens if we need to move one instance of `TempDir` from one location to another? For instance:

```
Task createTask()
{
    TempDir dir;
    fillDirectory(dir.path());
    // imagine a Task object that requires a
    // temporary directory already filled with content
    return Task{std::move(dir)}; // how do we forward dir? std::move?
}
```

We have two common solutions:

2.1.1 Make a move constructor

The first solution is to write a move constructor to `TempDir`. This can be done in two ways:

1. **The move constructor allocates new resources for the moved-from instance.** In our case it means that the moved-from instance: (a) transfers the ownership of its directory to the newly constructed instance, and (b) creates a new temporary directory for itself. This leads to wasted resources in our example, as `dir` will be destructed just after the move call, and hence the directory will be created needlessly.
2. **The move constructor modifies the moved-from instance to indicate that it has lost ownership of the resources.** In our example the `TempDir` class then needs a way to remember it has lost the ownership of the directory and at the very least not to destroy it in its destructor. This makes for no wasted resources but breaks the class invariant by introducing the moved-from state. Indeed the class invariant then becomes: ***unless it has been moved-from,*** *any instance owns a temporary directory, which will only be removed once the instance is destructed*.

2.1.2 Use a wrapper

Using a wrapper (like `std::unique_ptr<TempDir>` or `std::optional<TempDir>`) is another alternative. It works without breaking the class invariant, as the wrapper handles the moved-from state for us. But now we need to work with a wrapper (pointer semantics in this case) which is more cumbersome and sometimes error-prone. Besides if we consider the type invariant of `std::unique_ptr<TempDir>` or `std::optional<TempDir>`, then we are no better than with `TempDir` equipped of the move constructor. Indeed the invariant of the wrapped type is: ***unless it is the nullptr/nullopt,*** *any instance owns a temporary directory, which will only be removed once the instance is destructed*.

2.2 Case 2: The p-impl design pattern

P-impl is a famous design pattern in C++. It consists of hiding all data-members of a class behind a unique data member of opaque type. A typical implementation looks like this:

```
// In header file:
class MyClass
{
public:
    MyClass();
    // other ctors are omitted for the moment
    ~MyClass() noexcept;
private:
    struct Impl;
    std::unique_ptr<Impl> _d;
};
```

```cpp
// In implementation (.cpp) file:
struct MyClass::Impl
{
    // [...] all data members go here
};

MyClass::MyClass() : _d{std::make_unique<Impl>()} {}
MyClass::~MyClass() noexcept {}
```

What if we need to put instances of `MyClass` into a vector?

```cpp
MyClass Make();

void foo()
{
    std::vector<MyClass> vec;
    for (int i = 0; i  != 10; ++i)
        vec.push_back(Make());
}
```

We will need to make a move constructor. We have two choices here:

```cpp
// First implementation:
MyClass::MyClass(MyClass&& other) : _d{std::make_unique<Impl>(std::move(*other._d))} {}
// Second implementation:
MyClass::MyClass(MyClass&& other) : _d{std::move(other._d)} {}
```

1. The first implementation makes a new allocation of `Impl` and delegates to `Impl` move constructor. It also guarantees that `other` is left in a correct state (i.e. `other._d` is not null). However we could argue that it is sub-optimal as the move constructor needs to perform a memory allocation.
2. The second implementation directly takes the `Impl` from `other`. It does not perform any memory allocation but it is leaving `other` is a somewhat invalid state (`other._d` is the nullptr). This new state (`_d == nullptr`) now needs to be properly handled by every public function of `MyClass`.

In our case the moved-from instance will be destructed right after being moved. Hence in this scenario (which is common), none of these two implementations are satisfying (one makes an unnecessary memory allocation, the other adds unnecessary checks in the public API).

2.3 Drawbacks of the move constructor and rvalue-references

We showed through those examples that move constructors and rvalue-references have some drawbacks, notably because of the introduction of the "moved-from" state. The legitimate fact that a moved-from object must remain in a valid state may introduce either some extra house-keeping, or extra allocated resources that will likely be wasted (destructed right away). We argue that in a fair amount of cases, this moved-from state is unnecessary as the moved-from objects will be disposed of right away.

In addition, the move constructor and rvalue-references have extra drawbacks:

- C++ programmers find the notion of moved-from state confusing. Moved-from state in classes are often not properly implemented. See also https://herbsutter.com/2020/02/17/move-simply/
- We cannot efficiently move `const` variables. In our `createTask` function, `dir` is created and never modified, except when being moved to the `Task` object right before its destructor is called. We could have been tempted to mark `dir` as `const` , but the move prevents us.

2.4 Solution discussed in this paper

This paper introduces the notion of relocation as an attempt to fix this. Class instances can be relocated without the added burden of the moved-from state. This proposed relocation mechanism is similar to the move semantics introduced in C++11, but with the extra warranty that "relocated-from" objects are disposed of right-away, and hence alleviate the need to implement support for the "relocated-from" state in class designs.

Note: This paper does not propose a replacement to move semantics. Relocation can live along with move semantics, and move semantics cover use cases that relocation does not.

This paper introduces:

1. a new `reloc` operator that can be used to relocate local complete named objects.
2. a new constructor: the relocation constructor. This constructor also acts as a destructor for the relocated object.
3. a new kind of reference: the relocation reference. A relocation reference on a type `T` is denoted by: `T~`

The solution to our problem just becomes:

```
class TempDir
{
public:
    /* [...] other functions are still here,
     * we remove the move constructor and
     * we just add the new relocation constructor: */

    /* \brief relocation constructor
     * \param[in] other TempDir instance to build *this from
     *
     * other can be left in a dirty state as it will be disposed of right after
     * (this constructor serves as its destructor)
     *
     * TempDir~ is a relocation reference on a TempDir object.
     */
    TempDir(TempDir~ other) noexcept = default;
};

class Task
{
    TempDir _d;
public:
    explicit Task(TempDir d) : _d{reloc d} {}
};

Task createTask()
{
    const TempDir dir;
    fillDirectory(dir.path());
    return Task{reloc dir}; /* reloc is a new operator that marks
        the end of life of a named object and
        relocates it at a new address. */
}
```

And:

```
MyClass::MyClass(MyClass~ other) noexcept : _d{other._d} {}
//                                                      ^
//                  calls std::unique_ptr(unique_ptr~)
//
// or MyClass::MyClass(MyClass~ other) noexcept = default; does the same thing
```

```
void foo()
{
    std::vector<MyClass> vec;
    for (int i = 0; i  != 10; ++i)
        vec.push_back(std::relocate, Make()); /*
            suggested new std::vector API to insert items by value so that
            they are relocated directly into the vector memory. */
}
```

3 reloc operator

This paper suggests to introduce a new operator, named `reloc`. `reloc` is a unary operator that can be applied to named local complete objects (understand, variables that will be destructed in the same function `reloc` is used from, and whose destructor call may be discarded). This operator aims to clearly indicate that a variable will be relocated and must not be reused within its scope.

`reloc obj` constructs a new instance from `obj` and marks the "early" end of scope of `obj`. Unless otherwise specified, `reloc obj` returns the freshly built instance as a temporary. We rely on mechanisms similar to copy elision to elude this temporary object in most cases.

In addition `reloc obj` relocates the object in a safe way; it guarantees that no object slicing will occur, proper destruction of the relocated object, and prevents programming errors where the relocated variable is reused.

3.1 unrelocatable objects

An object is said to be *unrelocatable* (with regards to a function `f`) if any of the following is true:

- the object is not a *complete object* ;
- the object does not have local storage with regards to `f` ;
- the object is ref-qualified or is a pointer dereference ;
- the object is anonymous ;
- the object is a structured binding ;
- the object has no relocation constructor, move constructor or copy constructor.

3.1.1 reloc on unrelocatables is an error

It is a compile-time error to:

- call `reloc` within a function `f` on *unrelocatable* objects ;
- call `reloc` outside a function body.

For instance:

```
void foo(std::string str);
std::string get_string();
std::pair<std::string, std::string> get_strings();

std::string gStr = "static string";

void bar(void)
{
    std::string str = "test string";
    foo(reloc str); // OK
    foo(reloc gStr); // ERROR, gStr does not have automatic storage duration

    std::pair p{std::string{}, std::string{}};
    foo(reloc p.first); // ERROR: p.first is not a complete object
```

```
    foo(reloc get_string()); // ERROR: reloc on anonymous object
    foo(reloc get_strings().first); // ERROR: not a complete object
        // and is not anonymous
}

void foobar(const std::string& str)
{
    foo(reloc str); // ERROR: str does not have local storage
}
void foobar(std::string* str)
{
    foo(reloc *str); // ERROR: *str does not have local storage
}
void foobar2(std::string* str)
{
    foobar(reloc str); // OK, the pointer itself is relocated (not the pointed value)
}

class A
{
    std::string _str;
public:
    void bar()
    {
        foo(reloc _str); // ERROR: _str is not a complete object
    }
};
```

3.2 reloc stages

`reloc obj` (with `obj` of type `T`) acts in three stages:

1. Constructs a new object from `obj` using the relocation constructor (if available), the move constructor (if available) or the copy constructor.
2. Ensures the destruction of `obj`.
3. Mark the end of scope of the name `obj`. Any further instruction using `obj` up to today's-end-of-scope of `obj` is an error. Any pointer or reference on the relocated object becomes dangling once the instruction containing `reloc` is completed.

3.2.1 Stage 1: construct a new instance

`reloc obj` will use the following rules to construct the new instance:

1. If `T` defines a relocation constructor, then does: `T{const_cast<T~>(obj)}`.
2. Otherwise if `T` defines a move constructor, then does: `T{static_cast<T&&>(obj)}`.
3. Otherwise does: `T{obj}`.

If `T` defines a relocation constructor then `obj` will be considered destructed after the relocation constructor. In which case, in the expression evaluation, `obj` is marked as destructed right after the relocation constructor call.

3.2.2 Stage 2: post-evaluation

At the end of the evaluation of an instruction containing `reloc`, the relocated objects must be destructed. As such, if `obj` isn't marked as destructed then its destructor is called. `obj` is then marked as as destructed.

The destructor calls for undestructed relocated objects happen at the end of the expression evaluation, similar to when temporary objects of an expression are destructed.

3.2.3 Stage 3: early end of scope

`reloc obj` simulates an early end-of-scope of `obj`. To do so, it forbids any further mention of the name `obj`.

Any mention of the name `obj` which resolves to the object that was relocated, **in all code paths** from after the instruction that contained `reloc obj` up to its end of scope, will then yield a compilation error.

This further implies that any extra mention of `obj` in an instruction containing `reloc obj` is a compile-time error. For instance `foo(x, func(reloc x));` and `bar(reloc y, reloc y);` both yield an error.

Consider the following examples:

```
void relocate_case_01()
{
    const T var = getT();
    bar(reloc var);
    if (sometest(var)) // ERROR
        do_smth(var); // ERROR
}
```

`var` cannot be reused after the `reloc` call, as it is now out of scope.

```
void relocate_case_02()
{
    const T var;
    {
        const T var;
        bar(reloc var);
        do_smth(var); // ERROR
        {
            const T var;
            do_smth(var); // OK
        }
        do_smth(var); // ERROR
    }
    do_smth(var); // OK
}
```

The second and forth calls to `do_smth(var)` are allowed because the name `var` does not resolve to the relocated object.

```
void relocate_case_03()
{
    const T var = getT();
    if (sometest(var))
        bar(reloc var);
    else
        do_smth(var); // OK
}
```

`do_smth(var)` is allowed because the `else` branch is not affected by the `reloc` call of the `if` branch.

```
void relocate_case_04()
{
```

```
    const T var = getT();
    if (sometest(var))
        bar(reloc var);
    else
        do_smth(var); // OK
    // [...]
    do_smth_else(var); // ERROR
}
```

`do_smth_else(var)` is an error because `var` is mentioned after the `reloc` call.

```
void relocate_case_05()
{
    const T var = getT();
    bool relocated = false;
    if (sometest(var))
    {
        bar(reloc var);
        relocated = true;
    }
    else
        do_smth(var); // OK
    // [...]
    if (!relocated)
        do_smth_else(var); // ERROR
}
```

Same here. It does not matter that the programmer attempted to do the safe thing with the `relocated` variable. The code-path analysis associated with this stage of `reloc` is a compile-time analysis. Run-time values (like `relocated`) are disregarded.

```
void relocate_case_06()
{
    constexpr bool relocated = my_can_relocate<T>{}();
    const T var = getT();
    if constexpr(relocated)
    {
        bar(reloc var);
    }
    else
        do_smth(var); // OK
    // [...]
    if constexpr(!relocated)
        do_smth_else(var); // OK
}
```

The above example is safe because (a) now `relocated` is a `constexpr` and of (b) the use of `if constexpr` .

```
void relocate_case_07()
{
    const T var = getT();
    if (sometest(var))
    {
        bar(reloc var);
        return;
    }
    do_smth(var); // OK
```

```
}
```

This example is also safe thanks to the `return` statement right after the `reloc` operator, which prevents from running `do_smth(var);` .

```
void relocate_case_08()
{
    const T var = getT();
    for (int i = 0; i != 10; ++i)
        do_smth(reloc var); // ERROR
}
```

This will not work as each iteration reuses `var` which is declared out of scope in the loop. Even if `i` were compared against `1` or even `0` (i.e. `for (int i = 0; i != 1; ++i)` (one iteration) or `for (int i = 0; i != 0; ++i)` (no iteration)) this code will still be invalid. Run-time values (like `i`) are disregarded in the code-path analysis that comes with `reloc` . The analysis will report that there is an optional code jump, after the `do_smth` call (and `reloc var`), which jumps to before the `reloc var` call and after the initialization of `var` . Although the jump is optional (depends on `i` , whose value is disregarded) it may still happen and as such such code will produce an error.

```
void relocate_case_09()
{
    const T var = getT();
    for (int i = 0; i != 10; ++i)
    {
        if (i == 9)
            do_smth(reloc var); // ERROR
        else
            do_smth(var); // ERROR
    }
}
```

This will not work for the same reason as above. The code-path analysis will report that any iteration of the for-loop may take any branch of the if statement and potentially reuse a relocated variable.

```
void relocate_case_10()
{
    const T var = getT();
    for (int i = 0; i != 10; ++i)
    {
        if (i == 9) {
            do_smth(reloc var); // OK
            break;
        }
        else
            do_smth(var); // OK
    }
}
```

Adding the break statement right after the `reloc` call makes the code work. Indeed the `break` statement forces the exit of the loop, which implies that the conditional jump at the end of loop (that may start the next iteration) is no longer part of the code path that follows the `reloc` operator call.

```
void relocate_case_11()
{
    for (int i = 0; i != 10; ++i)
    {
        const T var = getT();
```

```
        do_smth(reloc var); // OK
    }
}
```

`var` is local to the for-loop body, so `reloc` is safe here.

```
void relocate_case_12()
{
    const T var = getT();
from:
    if (sometest(var)) // ERROR
    {
        do_smth(var); // ERROR
    }
    else
    {
        do_smth(reloc var);
    }
    goto from;
}
```

Because of the `goto` instruction, `var` may be reused after `reloc var`.

```
void relocate_case_13()
{
    const T var = getT();
from:
    if (sometest(var)) // OK
    {
        do_smth(var); // OK
        goto from;
    }
    else
    {
        do_smth(reloc var);
    }
}
```

In this scenario `goto` is placed in a way that does not trigger the reuse of relocated `var`.

3.2.3.1 Tracking object destruction

In certain situations some extra hidden booleans need to be introduced to track the destruction state of relocated objects.

Consider the following instruction: `foo(reloc p1, bar(), reloc p2);`. `bar()` may throw and interrupt the normal code path. In such cases the program must obviously still call the destructors of `p1` and `p2` if they are not marked as destructed. In this case the extra flags may only pertain to the instruction.

Another example is the following code:

```
bool foo()
{
    const T var;
    if (sometest(var))
        bar(reloc var);
    else
        do_smth_else(var);
```

```
    return true;
}
```

Here var is either destructed by the reloc operator, or when the control flow reaches the return statement. Some flag may be introduced so that, at scope exit, the destructor of var is not called if it was passed to the reloc operator.

In a general case, the use of these flags may be needed when two code paths (one where a variable is passed to reloc, another when not) join. Then these flags can be used at scope exit to know whether to call the destructor.

This is an implementation detail. We suggest leaving this out of the language specification and for compiler vendors' discretion only.

3.3 Reloc initialization

If a `reloc` statement is used to initialize an object (like in `auto y = reloc x;`), that the object to initialize and the object to relocate have the same type (cv-qualifiers ignored), and that the object to initilize is not ref-qualified, then `reloc` constructs the new instance into the object to initialize directly instead of returning a temporary.

Consider the following:

```
void foobar(T a);
void foo(T obj)
{
    T a = reloc obj;
    // ...
}
void bar(T obj)
{
    T a{reloc obj};
    // ...
    foobar(reloc a);
}
void foo2(T obj)
{
    auto a = T{reloc obj};
}
void foo3(T obj)
{
    auto a = reloc obj;
}
class FOO {
    T a;
public:
    FOO(T obj) : a{reloc obj} {}
};
```

In all those cases `reloc` constructs the new instance into `a` directly (even when `a` is the parameter of `foobar`).

3.4 Initialization from a temporary

If an object is initialized from a temporary object of the same type (cv-qualifiers ignored), and that that type is relocation constructible, then the object is initialized from its relocation constructor and the destructor of the temporary is not called (because destructed by the relocation constructor).

Note: this only applies if copy elision could not happen.

For instance:

```
class T {
public:
    T(T~) noexcept = default;
};

T foo();

void bar() {
    const T a = foo(); // the relocation constructor is called,
        // unless the copy can be entirely optimized out
}

T const& foo2();

void bar2() {
    const T a = foo2(); // the copy constructor is called,
        // unless the copy can be entirely optimized out
}
```

3.5 Placement-reloc operator

Just like the `new` operator, the `reloc` operator has a “placement” variant: `reloc (addr) obj`. This behaves like `reloc obj` except that the new instance is constructed at the provided address `addr`. In fact `reloc (addr) obj` is equivalent to `new (addr) T{reloc obj}` but is clearer.

For instance:

```
template<class T, std::size_t N>
class static_vector
{

    std::aligned_storage_t<sizeof(T), alignof(T)> _data[N];
    std::size_t _size = 0;

public:
    void push_back(std::relocate_t, T d)
    {
        if (_size >= N)
            throw std::bad_alloc{};

        reloc (_data+_size) d;
        // d is now out of scope
        ++_size;
    }
};
```

3.6 Relocation of C-array

C-arrays can be relocated as long as they are not *unrelocatable* and the size of the array is known. `reloc` called on a C-array returns a new C-array of the same size, where each element has been relocated.

See also:

```
void foo(int (&x)[])
{
    int y[] = reloc x; // ERROR: x does not have local storage
```

```
}
void foo2(int (&x)[N])
{
    int y[N] = reloc x; // ERROR: x does not have local storage
}
void foo3()
{
    int x[5] = {0};
    auto y = reloc x; // OK, y has type int[5]
}
```

While relocation of a C array might not seem beneficial, it enables relocation of `std::array` or any class using `std::array`.

4 Relocation constructor

The new relocation constructor is written as follows:

```
class T
{
public:
    T(T~ other) noexcept;
};
```

The relocation constructor acts as a destructor for the relocated object. The relocation constructor is in fact a second destructor. **The regular destructor of an object must not be called if said object was passed to a relocation constructor.**

The relocation constructor signature must be `T::T(T~)` (`noexcept` is optional). `noexcept` is recommended as it is for regular destructors. Throwing from a relocation constructor is an UB.

Relocation constructors can also be defaulted:

```
class T
{
public:
    T(T~ other) noexcept = default;
};
```

Note: The relocation constructor can be called manually by programmers, but at their own risk. If the relocation constructor is called manually without using `reloc` :

- object slicing may occur (e.g. if we are relocating a derived class to a base class) ;
- the language won't discard the call to the destructor of the relocated object ;
- the language won't offer any protection against reuse of the relocated object.

However this remains permitted as useful on some cases, like manual memory management (e.g. `std::vector` implementation).

4.1 Mixed-type relocation

Relocation from an object of another type is not permitted:

```
class T
{
public:
    T(U~ other) noexcept; // with U different from T;
        // ERROR: T::T(U~) is not a valid relocation constructor
};
```

Since the destructor of a class can only be defined in that class, the same goes for the relocation constructor.

4.2 Implicit declaration

4.2.1 Implicitly-declared relocation constructor

If no user-defined relocation constructors are provided for a class type (struct, class, or union), and all of the following is true:

- there is no user-declared copy constructor
- there is no user-declared move constructor
- there is no user-declared destructor

then the compiler will declare a relocation constructor as an inline public member of its class with the signature `T::T(T~) noexcept` .

Rules for implicit declaration of copy constructor and move constructor are also changed to take the relocation constructor into account, but this change will be detailed in a future revision.

4.2.2 Deleted implicitly-declared relocation constructor

The implicitly-declared or defaulted relocation constructor for class `T` is defined as *deleted* if any of the following is true:

- `T` has non-static data members that cannot be relocated (have deleted, inaccessible, or ambiguous relocation constructors)
- `T` has direct or virtual base class that cannot be relocated (has deleted, inaccessible, or ambiguous relocation constructors)
- `T` has direct or virtual base class with a deleted or inaccessible destructor
- `T` is a union-like class and has a variant member with non-trivial relocation constructor.

A defaulted relocation constructor that is deleted is ignored by overload resolution.

Rules for deleted implicitly-declarated copy constructor and move constructor are also changed to take the relocation constructor into account, but this change will be detailed in a future revision.

4.3 Trivial relocation constructor

The relocation constructor for class `T` is *trivial* if all of the following is true:

- it is not user-provided (meaning, it is implicitly-defined or defaulted);
- `T` has no virtual member functions;
- `T` has no virtual base classes;
- `T` has no non-static data members of volatile-qualified type;
- the relocation constructor selected for every direct base of `T` is *trivial*;
- the relocation constructor selected for every non-static class type (or array of class type) member of `T` is *trivial*.

A trivial relocation constructor is a constructor that performs the same action as the trivial copy constructor, that is, makes a copy of the object representation as if by `std::memmove` . All data types compatible with the C language (POD types) are trivially relocation constructible.

4.3.1 Optimization with trivially relocation constructible objects

Data structures are encouraged to check whether the objects they handle have a trivial relocation constructor. If that is the case, then the call to the relocation constructor can be completely shortcut by a mere memmove.

`std::vector` is a good candidate for such optimizations. Each time it needs to move its data to a new memory chunk, it can make a simple `std::memmove` call instead of calling some constructor and destructor on all its items. See also `std::uninitialized_reloc` and `std::relocate_at` .

4.4 Implicitly-defined relocation constructor

If the implicitly-declared relocation constructor is neither deleted nor trivial, it is defined (that is, a function body is generated and compiled) by the compiler if odr-used or needed for constant evaluation. For union types, the implicitly-defined relocation constructor copies the object representation (as by `std::memmove`). For non-union class types (class and struct), the relocation constructor performs full member-wise relocation of the object's bases and non-static members, in their initialization order, using direct initialization with a relocation reference argument. If this satisfies the requirements of a constexpr constructor, the generated relocation constructor is `constexpr` .

5 Relocation reference

The relocation constructor takes a relocation reference as parameter. A relocation reference on an object of type `T` is denoted by `T~` . Relocation references are similar to lvalue references, but have a different type because of the different use cases. Semantically a relocation reference is a reference on an object which is being relocated (it was passed as parameter to the relocation constructor).

5.1 Interoperability with other references

5.1.1 References on relocation references

Any reference on a relocation reference is a relocation reference. As such: `T~ &` is the same as `T~` , and `T~ &&` is the same as `T~` . This is motivated by :

- Taking a reference on a reference does not add a level of indirection (`T& &` is the same of `T&` and is not internally a double pointer). Hence a reference on a relocation reference must still be a reference.
- `T~ &` is still a reference on an object being relocated, which is the definition of a relocation reference.

5.1.2 Relocation references on references

Any relocation reference on a reference stays a reference on the same type. As such: `T& ~` is the same as `T&` , `T&& ~` is the same as `T&&` , and `T~ ~` is the same as `T~` .

5.1.3 Relocation reference casts

A relocation reference can be implicitly cast to an lvalue reference. For instance:

```
void foo(const T&);
void bar(T&&);

void T::foobar();

T::T(T~ t) noexcept
{
    foo(t); // OK, implicit cast
    bar(t); // ERROR, cannot convert from T~ to T&&
    bar(std::move(t)); // OK, using new std::move overload
    foo(static_cast<const T&>(t)); // OK
    t.foobar(); // OK
}
```

An object cannot be implicitly cast to a relocation reference. One must use a `const_cast` .

```
void foo(T~ t);
void bar()
{
    T t;
    T a(const_cast<T~>(t)); // OK, but at the programmer's risk.
```

```
    // destructor of t will still be called, which will likely cause a leak
}
```

5.1.4 Relocation pointer

Taking the address of a relocation reference of type `T~` gives a relocation pointer, designated by `T~*`.

```
T::T(T~ t) noexcept
{
    static_assert(std::is_same_v<decltype(&t), T~*>);
    T~* ptr = &t;
    static_assert(std::is_same_v<decltype(&ptr), T~**>);
}
```

A relocation pointer acts like a regular pointer, excepts that its `operator*` and `operator[]` yield a relocation reference instead of an lvalue reference.

Implicit casts from a relocation pointer to a regular pointer are authorized:

```
T::T(T~ t) noexcept
{
    T* ptr = &t; // OK
}
```

5.2 cv-qualifiers

The relocation reference discards any `const` and `volatile` qualifiers (i.e. `T~` is the same type as `const T~`, `volatile T~`, and `const volatile T~`). The incentive is that a relocation reference denotes an object that is being destructed (consequence of the relocation). Destruction will happen the same way regardless of the cv-qualifiers it had before. C++ destructors work this way: the same destructor is called regardless of the cv-qualifiers of the object.

5.3 Propagation on data members

Suppose we have a *relocation reference* `t` : `T~ t`. Any non-static data member of `t` is also a relocation reference. To illustrate:

```
template <class A, class B>
pair<A,B>::pair(pair<A,B>~ t) noexcept
{
    // decltype(t.first) is A~
    // decltype(t.second) is B~
}
```

The rationale is that since `t` is being relocated, then so is any of its non-static data member. Hence all non-static data members of a relocation reference are relocation references as well.

5.4 Relocation reference on *this

Relocation references allow another kind of function overloads:

```
class T
{
public:
// [...]
    U& getU();
    U const& getU() const;
// New possible overload, selected only if *this is a relocation reference
```

```
    U~ getU() ~;
};
```

5.5 Structured relocation

Structured binding is a language feature that enables to split an object into parts and to initialize name aliases that refer to each part: `auto&& [x,y] = foo();` . Here `x` and `y` are the new identifiers that reference each one part of the object returned by `foo()` (the initializer).

Structured relocation is a suggested variant to structured binding. Structured relocation enables to split a complete object (the initializer) into parts, and each part is relocated into new complete objects. The initializer is fully relocated at the end of the expression as we expect each of its part to form a partition of the object. Unlike in structured bindings, the newly introduced names are not aliases but complete objects on their own.

A structure binding declaration is upgraded to a structured relocation declaration if all the following conditions are met:

- the declared identifiers are not ref-qualified. Each declared identifier must be a complete object and not a reference to another subobject. (`auto [x, y] = foo();` is okay, `auto&& [x, y] = foo();` is not) ;
- the structured binding must be initialized from either:
  - a reloc statement: `auto [x, y] = reloc obj;`
  - a temporary object: `auto [x, y] = foo();`
- the type of the initializer must support structured relocation, as described below.

If the structure binding declaration could not be upgraded to a structured relocation then the rules for the structure binding declaration are applied.

In what follows, we denote by `t` of type `T` the complete object that is to be relocated.

5.5.1 Enabling structured relocation

As of C++17, structured bindings may be performed in three ways:

- **Array case**: if `T` is an array type.
- **Tuple-like case**: if `T` is a non-union class type and `std::tuple_size<T>` is a complete type with a member named `value` ;
- **Data members case**: if `T` is a non-union class type and `std::tuple_size<T>` is not a complete type.

5.5.1.1 Array case

`T` is an array type whose array element type is denoted by `U` and size by `N` (i.e. `T` is `U[N]` for some integer `N`).

Structured relocation is enabled if and only if `U` is relocation constructible. In which case each object of the structured relocation is initialized by its relocation constructor, whose parameter is a relocation reference on the corresponding array element.

5.5.1.2 Tuple-like case

The initializer for the I-th variable is:

- `const_cast<T~>(t).get<I>()` , if lookup for the identifier `get` in the scope of `T` by class member access lookup finds at least one declaration that is a function template whose first template parameter is a non-type parameter and is specialized for a relocation reference on *this ;
- Otherwise, `get<I>(const_cast<T~>(t))` , where `get` is looked up by argument-dependent lookup only, ignoring non-ADL lookup, and its only parameter is of type `T~` .

Structured relocation is enabled if and only if such a `get<I>` function (`I` of type `std::size_t`) is found for all introduced objects.

The I-th object is then constructed from the result of the selected `get<I>` function.

5.5.1.3 Data-member case

The constraints specified in C++ standard that apply on `T` for this structured binding declaration case apply.

In addition, structured relocation is enabled if and only if `T` is *triavially relocation constructible* and only have public data-members.

The I-th object is then constructed by its relocation constructor, whose parameter is a relocation reference on the corresponding I-th non-static data-member, using the same data-member selection rules as in structured binding.

6 Changes to the standard library

6.1 Type traits

```
namespace std
{
template <class T>
struct is_relocation_reference : public std::false_type {};
template <class T>
struct is_relocation_reference<T~> : public std::true_type {};

template<class T>
inline constexpr bool is_relocation_reference_v = is_relocation_reference<T>::value;

template<class T>
struct add_relocation_reference; /*
    If `T` is an object or function that isn't ref-qualified and isn't a pointer,
    then provides a member typedef `type` which is `T~`.
    If `T` is a relocation reference to some type `U`, then `type` is `U~`.
    Otherwise, `type` is `T`. */

template<class T>
using add_relocation_reference_t = typename add_relocation_reference<T>::type;

template<class T>
struct is_relocation_constructible;
template<class T>
struct is_trivially_relocation_constructible;

/* Possible implementation:
template<class T>
struct is_relocation_constructible; :
    std::is_constructible<T, typename std::add_relocation_reference<T>::type> {};
template<class T>
struct is_trivially_relocation_constructible :
    std::is_trivially_constructible<T, typename std::add_relocation_reference<T>::type> {};
*/

template<class T>
inline constexpr bool is_relocation_constructible_v =
```

```
    is_relocation_constructible<T>::value;
template<class T>
inline constexpr bool is_trivially_relocation_constructible_v =
    is_trivially_relocation_constructible<T>::value;
}
```

6.2 Algorithm

6.2.1 relocate_at

```
namespace std
{
template <class T>
void relocate_at(T* src, void* dst);
}
```

Relocates `src` into `dst`. `src` will be destructed at the end of the call. `src` and `dst` must not be the nullptr, or it otherwise results in UB.

If `T` is trivially relocation constructible, it behaves as if by:

```
template <class T>
void relocate_at(T* src, void* dst)
{
    memmove(dst, src, sizeof(T));
}
```

Otherwise if `T` is relocation constructible, it behaves as if by:

```
template <class T>
void relocate_at(T* src, void* dst)
{
    new (dst) T{const_cast<T~>(*src)};
}
```

Note that this version does not check for exceptions as it an UB to throw from a relocation constructor.

Otherwise if `T` is move constructible, it behaves as if by:

```
template <class T>
void relocate_at(T* src, void* dst)
{
    try {
        new (dst) T{static_cast<T&&>(*src)};
        src->~T();
    } catch (...) {
        src->~T();
        throw;
    }
}
```

Otherwise it behaves as if by:

```
template <class T>
void relocate_at(T* src, void* dst)
{
    try {
        new (dst) T{*src};
        src->~T();
    } catch (...) {
```

```
        src->~T();
        throw;
    }
}
```

6.2.2 uninitialized_reloc

```
namespace std
{
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first);

template<class ExecutionPolicy, class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(ExecutionPolicy&& policy, InputIt first, InputIt last,
    ForwardIt d_first) ;

template<class InputIt, class Size, class ForwardIt>
std::pair<InputIt, ForwardIt> uninitialized_reloc_n(InputIt first, Size count,
    ForwardIt d_first) ;

template<class ExecutionPolicy, class InputIt, class Size, class ForwardIt>
std::pair<InputIt, ForwardIt> uninitialized_reloc_n(
    ExecutionPolicy&& policy, InputIt first, Size count, ForwardIt d_first);
}
```

Relocates elements from the range `[first, last)` to an uninitialized memory area beginning at `d_first` . Elements in `[first, last)` will be destructed at the end of the function (even if an exception is thrown).

Returns:

- `uninitialized_reloc` : an iterator to the element past the last element relocated;
- `uninitialized_reloc_n` : a pair whose first element is an iterator to the element past the last element relocated in the source range, and whose second element is an iterator to the element past the last element relocated in the destination range.

If the type to relocate is trivially relocation constructible and both iterator types are contiguous, it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first)
{
    using value_type = typename std::iterator_traits<ForwardIt>::value_type;

    std::memmove(static_cast<void*>(std::addressof(*d_first)),
        static_cast<void*>(std::addressof(*first)),
        std::distance(first, last)*sizeof(value_type));

    return d_first + std::distance(first, last);
}
```

If the type to relocate is trivially relocation constructible and one of the iterator type is not contiguous, it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first)
{
    using value_type = typename std::iterator_traits<ForwardIt>::value_type;

```

```
    for (; first != last; ++d_first, (void) ++first)
        std::memmove(static_cast<void*>(std::addressof(*d_first)),
            static_cast<void*>(std::addressof(*first)),
            sizeof(value_type));

    return d_first;
}
```

If the type to relocate is relocation constructible (not trivially), it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first)
{
    using value_type = typename std::iterator_traits<ForwardIt>::value_type;

    for (; first != last; ++d_first, (void) ++first)
        ::new (static_cast<void*>(std::addressof(*d_first)))
            value_type{const_cast<value_type~>(*first)};

    return d_first;
}
```

Note that this version does not check for exceptions as it an UB to throw from a relocation constructor.

If the type to relocate is move constructible, it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first)
{
    using value_type = typename std::iterator_traits<ForwardIt>::value_type;

    try {
        for (; first != last; ++d_first, (void) ++first) {
            ::new (static_cast<void*>(std::addressof(*d_first)))
                value_type{static_cast<value_type&&>(*first)};
            first->~value_type();
        }
    } catch (...) {
        for (; first != last; ++first)
            first->~value_type();
        throw;
    }

    return d_first;
}
```

Last, if the type to relocate is only copy constructible, it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_reloc(InputIt first, InputIt last, ForwardIt d_first)
{
    using value_type = typename std::iterator_traits<ForwardIt>::value_type;

    try {
        for (; first != last; ++d_first, (void) ++first) {
            ::new (static_cast<void*>(std::addressof(*d_first))) value_type{*first};
            first->~value_type();
        }
    } catch (...) {
```

```
        for (; first != last; ++first)
            first->~value_type();
        throw;
    }

    return d_first;
}
```

6.3 Utility library

6.3.1 std::move

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

6.3.2 reloc helpers

```
namespace std
{
// placeholder to indicate that the next parameter
// will be passed by value and be relocated
struct relocate_t {};
inline constexpr relocate_t relocate = {};

// wrapper around a value to be relocated that can be passed by reference
// may use an std::optional<T> in its implementation.
template <class T>
struct reloc_wrapper
{
public:
    /**
     * Construct a wrapper from a value. The value will be relocated into the
     * contained value of reloc_wrapper.
     */
    explicit reloc_wrapper(T value) noexcept;

    /**
     * Returns whether the reloc_wrapper has a value
     */
    bool has_value() const noexcept;

    /**
     * returns the value, relocated from the contained value.
     * throws a new exception (bad_reloc_access, derived from std::logic_error)
     * if extract was already called.
     */
    T extract();
};
}
```

6.3.3 std::pair and std::tuple

```
template <class U1, class U2>
constexpr pair(U1&& x, U2&& y);

template <class U1, class U2>
constexpr std::pair<V1,V2> make_pair(U1&& x, U2&& y);

template< class... UTypes >
constexpr tuple( UTypes&&... args );

template< class... Types >
constexpr tuple<VTypes...> make_tuple( Types&&... args );
```

If `std::decay_t<U1>` (resp. `std::decay_t<U2>`) results in `std::reloc_wrapper<X>` for some `X` then the pair data member is initialized with `x.extract()` (resp. `y.extract()`).

Today's rule for `V1` and `V2` is: The deduced types `V1` and `V2` are `std::decay<T1>::type` and `std::decay<T2>::type` (the usual type transformations applied to arguments of functions passed by value) unless application of `std::decay` results in `std::reference_wrapper<X>` for some type `X` , in which case the deduced type is `X&` . *We suggest in addition:* if `std::decay` results in `std::reloc_wrapper<X>` for some type `X` , in which case the deduced type is `X` .

The same changes are suggested to `std::tuple` constructor and `std::make_tuple` .

6.3.4 std::get (pair, tuple, array)

We suggest adding the following overloads to enable structured relocation with pair, tuple and arrays:

```
template< std::size_t I, class T1, class T2 >
constexpr std::tuple_element_t<I, pair<T1, T2> >~
    get( pair<T1, T2>~ t );

template< class T, class T1, class T2 >
constexpr T~ get( pair<T1, T2>~ t );

template< std::size_t I, class... Types >
constexpr std::tuple_element_t<I, tuple<Types...> >~
    get( tuple<Types...>~ t );

template< class T, class... Types >
constexpr T~ get( tuple<Types...>~ t );

template< size_t I, class T, size_t N >
constexpr T~ get( array<T,N>~ a );
```

6.3.5 std::optional

Add new class methods:

```
/**
 * \brief construct the optional by relocating val into the contained value
 * (as if by std::relocate_at).
 */
template <class T>
optional<T>::optional(std::relocate_t, T val);
```

```
/**
 * \brief Extracts the contained value from the optional
 *
 * The returned value is relocated from the contained value.
 *
 * After this call the optional no longer contains any value.
 *
 * \throws std::bad_optional_access if the optional did not contain any value.
 */
template <class T>
T optional<T>::extract();
```

6.3.6 std::variant

Add new class methods:

```
/**
 * \brief construct the variant by relocating val into the contained value
 * (as if by std::relocate_at).
 * If an exception is thrown, *this may become valueless_by_exception.
 */
template <class T>
constexpr variant(std::relocate_t, T val);

/**
 * \brief does the same as calling *this = t
 */
template <class T>
void assign(T&& t);

/**
 * \brief destroys the contained value if any, and constructs a new one by
 * relocating t (as if by std::relocate_at).
 * If an exception is thrown, *this may become valueless_by_exception.
 */
template <class T>
void assign(std::relocate_t, T t);
```

6.3.7 std::any

```
template<class T>
std::any make_any(std::relocate_t, T value);

template<class T>
std::any::any(std::relocate_t, T value);
```

Those aim to initialize an `std::any` from a relocated value.

6.4 Container library

All containers should provide a way to insert and remove data by relocation.

Unfortunately existing APIs cannot fulfill this need. They mostly take references of some kind as parameter, while relocation requires to pass items by value.

As such we suggest adding overloads to all insertion functions. These shall take an `std::relocate_t` as parameter and the item to add as next parameter (taken by value).

`std::relocate_t` is here to help distinguish from otherwise ambiguous overloads. Indeed `vec.push_back(reloc a);` will call `vector::push_back(T&&)` and the item `a` won't be relocated inside the vector. If we add `void vector::push_back(T val)` as overload then the previous call will become ambiguous.

We want to avoid the use of `std::reloc_wrapper` because of the extra relocation it incurs (the value needs to be relocated into the wrapper first).

In addition we add various "extract" function to remove values from the container.

6.4.1 std::vector

```
// pushes a value by relocation
template <class T, class Alloc>
constexpr void vector<T, Alloc>::push_back(std::relocate_t, T value);

// inserts a value by relocation
template <class T, class Alloc>
iterator vector<T, Alloc>::insert(const_iterator pos, std::relocate_t, T value);

// removes the last item from the vector and returns it
template <class T, class Alloc>
T vector<T, Alloc>::extract_back();

// removes the item from the vector and returns it with the next valid iterator
template <class T, class Alloc>
std::pair<T, const_iterator> vector<T, Alloc>::extract(const_iterator pos);

// relocates items in [from, to[ into out.
// items within range are removed from *this.
template <class T, class Alloc>
template <class OutputIterator>
OutputIterator vector<T, Alloc>::relocate(
    iterator from, iterator to, OutputIterator out);
```

6.4.2 std::deque

```
// pushes a value by relocation
template <class T, class Alloc>
constexpr void deque<T, Alloc>::push_front(std::relocate_t, T value);
template <class T, class Alloc>
constexpr void deque<T, Alloc>::push_back(std::relocate_t, T value);

// inserts a value by relocation
template <class T, class Alloc>
iterator deque<T, Alloc>::insert(const_iterator pos, std::relocate_t, T value);

// removes the last item from the queue and returns it
template <class T, class Alloc>
T deque<T, Alloc>::extract_back();
// removes the first item from the queue and returns it
template <class T, class Alloc>
T deque<T, Alloc>::extract_front();
// removes the item from the queue and returns it with the next valid iterator
template <class T, class Alloc>
std::pair<T, const_iterator> deque<T, Alloc>::extract(const_iterator pos);
```

```
// relocates items in [from, to[ into out.
// items within range are removed from *this.
template <class T, class Alloc>
template <class OutputIterator>
OutputIterator deque<T, Alloc>::relocate(
    iterator from, iterator to, OutputIterator out);
```

6.4.3 std::list

```
// pushes a value by relocation
template <class T, class Alloc>
void list<T, Alloc>::push_front(std::relocate_t, T value);
template <class T, class Alloc>
void list<T, Alloc>::push_back(std::relocate_t, T value);

// inserts a value by relocation
template <class T, class Alloc>
iterator list<T, Alloc>::insert(const_iterator pos, std::relocate_t, T value);

// removes the last item from the list and returns it
template <class T, class Alloc>
T list<T, Alloc>::extract_back();
// removes the first item from the list and returns it
template <class T, class Alloc>
T list<T, Alloc>::extract_front();
// removes the item from the list and returns it with the next valid iterator
template <class T, class Alloc>
std::pair<T, const_iterator> list<T, Alloc>::extract(const_iterator pos);

// relocates items in [from, to[ into out.
// items within range are removed from *this.
template <class T, class Alloc>
template <class OutputIterator>
OutputIterator list<T, Alloc>::relocate(
    iterator from, iterator to, OutputIterator out);
```

6.4.4 std::forward_list

```
// inserts a value by relocation
template <class T, class Alloc>
iterator forward_list<T, Alloc>::insert_after(const_iterator pos,
    std::relocate_t, T value);
template <class T, class Alloc>
void forward_list<T, Alloc>::push_front(std::relocate_t, T value);

// removes the first item from the list and returns it
template <class T, class Alloc>
T forward_list<T, Alloc>::extract_front();
// removes the item after pos from the list and returns it with the iterator following pos
template <class T, class Alloc>
std::pair<T, const_iterator> forward_list<T, Alloc>::extract_after(const_iterator pos);

// relocates items in ]from, to[ into out.
// items within range are removed from *this.
template <class T, class Alloc>
```

```
template <class OutputIterator>
OutputIterator forward_list<T, Alloc>::relocate_after(
    iterator from, iterator to, OutputIterator out);
```

6.4.5 set and map containers

```
// std::set, std::multiset, std::map, std::multimap,
// std::unordered_set, std::unordered_multiset, std::unordered_map
// and std::unordered_multimap, all aliased as 'map':
std::pair<iterator, bool> map::insert(std::relocate_t, value_type value);
iterator map::insert(const_iterator hint, std::relocate_t, value_type value);

// extract the stored value from the container
std::pair<value_type, const_iterator> map::extract_value(const_iterator position);
```

6.4.6 queues

```
// for std::stack, std::queue, std::priority_queue, aliased queue below:
void queue::push(std::relocate_t, T value);

// removes the next element from the queue
T queue::extract();
```

6.5 Iterator library

6.5.1 back_reloc_iterator

`std::back_reloc_iterator` is an OutputIterator that appends to a container for which it was constructed. The container's push_back(std::relocate overload) member function is called whenever the iterator (whether dereferenced or not) is assigned to. Incrementing the std::back_reloc_iterator is a no-op.

```
template< class Container >
std::back_reloc_iterator<Container> back_relocator( Container& c );
```

`back_relocator` is a convenience function template that constructs a `std::back_reloc_iterator` for the container c with the type deduced from the type of the argument.

6.5.2 front_reloc_iterator

`std::front_reloc_iterator` is an OutputIterator that appends to a container for which it was constructed. The container's push_front(std::relocate overload) member function is called whenever the iterator (whether dereferenced or not) is assigned to. Incrementing the std::front_reloc_iterator is a no-op.

```
template< class Container >
std::front_reloc_iterator<Container> front_relocator( Container& c );
```

`front_relocator` is a convenience function template that constructs a `std::front_reloc_iterator` for the container c with the type deduced from the type of the argument.

6.5.3 insert_reloc_iterator

`std::insert_reloc_iterator` is an OutputIterator that appends to a container for which it was constructed. The container's insert(std::relocate overload) member function is called whenever the iterator (whether dereferenced or not) is assigned to. Incrementing the std::insert_reloc_iterator is a no-op.

```
template< class Container >
std::insert_reloc_iterator<Container> insert_relocator( Container& c, typename Container::iterator it );
```

`insert_relocator` is a convenience function template that constructs a `std::insert_reloc_iterator` for the container c and its iterator i with the type deduced from the type of the argument.

6.6 Concept

6.6.1 TriviallyCopyable

The TriviallyCopyable has a new requirement: Every relocation constructor is trivial or deleted.

6.7 New constructors

Relocation constructors (with signature `T::T(T~) noexcept`) should be added *as defaulted* to the following classes of the standard library:

Library	Classes
Containers	All containers (implicitly declared for `std::array`)
String	`std::basic_string`
Utility	`std::pair`, `std::tuple`, `std::optional`, `std::any`, `std::variant`, `std::function`, `std::reference_wrapper`, `std::shared_ptr`, `std::weak_ptr`, `std::unique_ptr`
Regular expression	`std::basic_regex`, `std::match_results`
Thread support	`std::thread`, `std::lock_guard`, `std::unique_lock`, `std::scoped_lock`, `std::shared_lock`, `std::promise`, `std::future`, `std::shared_future`, `std::packaged_task`
Filesystem	`std::filesystem::path`

All classes that have at least one virtual function are not good candidates for relocation, and are such not listed here. Indeed, such objects are polymorphic by design, and should not be copied around by value as object slicing may occur.

7 Discussions

7.1 Intended usage

The aim of this paper is to enable relocation. We tried several different approaches (library solution, destructor dedicated to relocated objects, new STL reference wrapper instead of relocation references) and this one was the most promising.

It does come with a bunch of new concepts and rules (new kind of reference, new operator). We argue that most of these rules will never be used by most developers outside of their intended purpose (relocation references should not appear outside the relocation constructor).

We don't intend developers:

- to write other types of functions that consume relocation references, i.e. `void foo(T~) noexcept;` ;
- to cast an lvalue reference to a relocation reference without knowing what it does;
- to take the address of relocation reference and play with relocation pointers.

7.2 Why not a library solution?

To ensure proper relocation, we need to make sure that the destructor of the relocated value is not called, or rather, to consider the relocation constructor to be a destructor on its own. Otherwise things would not be much different from the move constructor. This requires changes in the language.

7.3 Why doesn't reloc return a relocation reference instead?

It may seem counter-intuitive that reloc returns the constructed object. It could have returned a relocation reference, which would trigger a relocation constructor call when used to initialize a new object.

But this wouldn't be safe. The relocation reference needs to reach a relocation constructor to ensure that the relocated object is destructed. Anything can happen from the `reloc` call and the moment the relocation constructor is reached (exceptions can be thrown, the reference forwarded to functions that simply discard it, etc…). Such a code would then be open to leaks.

Instead, the first version of this paper worked in a way similar to this suggestion. The `reloc` operator would not return a new instance, but some wrapper object. If the wrapper object made its way to a relocation constructor, then the wrapper would mark the object as destructed and the call site would not destruct the object.

We got everything together but we felt that the proposal was way more complicated than it ought to be, because of the new rules we had to add:

- have a convenient wrapper type that was handy to manipulate;
- subobjects (e.g. non-static data members) would have to be wrapped as well;
- make the relocation constructor have a special mechanic so it marks the destruction of the object…
- … however other functions that would simply forward the wrapper would not mark the object as destructed.

In the end, stating that `reloc` returns a newly built instance allowed for many simplifications.

7.4 Why a new operator?

When a variable was relocated it can no longer be used. The reloc operator emphasizes that point. `reloc` stands out in the code (we can easily see it), and it guarantees the early end of scope of the relocated variable.

7.5 Why a new type of reference?

We made several attempts to get the relocation constructor "right". The alternatives we considered are:

- Using a wrapper instead: `T::T(std::some_wrapper<T> val) noexcept`. This has several drawbacks:
  - The constructor signature is quite different from the copy and move constructors;
  - The value must be passed inside a wrapper;
  - The value must be rewrapped each time it is passed to base and data members relocation constructors;
  - There is no dot operator so accessing the contained value is less convenient.
- Using a placeholder type: `T::T(std::relocate_t, T&& value) noexcept`. Slightly better, but:
  - Requires to pass a dummy value each time, even to base and data members relocation constructors;
  - This constructor does not look "special". Programmers unfamiliar with the concept may be tempted to call the constructor directly: `auto a = T{std::relocate, std::move(obj)}` which will have disastrous consequences.

For all those reasons we found that using a dedicated type of reference was a better solution.

We also decided to denote it by `T~` to better emphasize its relationship with object destruction.

7.6 Why use const_cast to cast into a relocation reference?

We have no strong opinion on which type of cast to use. We hesitated between `static_cast` and `const_cast` .

On one hand, `static_cast` seems to be semantically the more correct choice, although it does not convey this sense of caution a `const_cast` does. On the other hand, casting to a relocation reference does not deal with cv-qualifiers (even though a relocation reference discards them), so `const_cast` feels a bit out of place.

We favored the sense of caution raised by `const_cast` , that's why we picked it.

7.7 Can reloc operator be overloaded?

No, we found no good use of this. We may add this in a future extension if a convincing use-case appears.

7.8 Will we see reloc used only to trigger early end of scope of variables?

For instance, will we see code like this, should this proposal be approved?

```
void foo()
{
    T a;
    T b;
    // do stuff with a and b;
    reloc b; // end of scope of b;
    // we no longer use b
    // do stuff with a only
}
```

This code could be used instead of introducing an extra scope in `foo` . We personally feel that adding extra scopes is more readable than using reloc in such a way. This could be avoided by adding `[[no_discard]]` to `reloc` , but we have no strong opinion about this.

7.9 Why reuse extract in STL containers and extract_value in set and map containers?

`std::set` and `std::map` already have their `extract` function, which don't do exactly what we want, so that's why we introduced `extract_value` instead.

We thought of names that could be identical across all STL containers. Another one we considered is `relocate` (and `relocate_back` , `relocate_front`). However this doesn't say if we intend to relocate a value inside or outside the container. Then this would become `relocate_out` , `relocate_back_out` and `relocate_front_out` , which is too verbose in our opinion. We could also use `extract_value` in every container. This point is left for further discussion.

7.10 Will it make C++ easier?

Even though it does come with new rules, we argue that it mostly removes the moved-from state understanding problem.

On one hand, if a introducing a "moved-from" state in a class feels out of place, then it's best to remove the move constructor and only use the relocation constructor.

On the other hand, if there is a use case where it makes sense to still use the value after moving it (like for an `std::vector` or `std::unique_ptr`) then the moved-from state makes sense and its implementation should be intuitive (empty vector or null pointer).