Relocation in C++

C++ standard proposal

Sébastien Bini

April 29, 2022

Contents

1 Revisions

Revision	Data	Changes
1	11/21/2021	First revision
2	03/16/2022	The relocation constructor is no longer a destructor

1.1 Changes from the first revision

The relocation constructor still acts as a destructor, but no longer forceably prevents the call to the destructor. This is motivated by the fact that it would lead to object destruction tracking (to ensure object is not destructed twice) across functions which breaks the ABI.

For the same reason, `reloc` no longer guarantees the object destruction but still prevents reuse of the name of the relocated object.

We further added comparison with existing works on the subject.

1.2 Comparison with existing proposals

This proposal introduces the `reloc` operator which, as far as we know, has not been proposed yet. This operator allows users to explicitly and safely relocate local variables in their code base.

This proposal is also one of the few (with P0308R0), to our knowledge, to tackle the case of classes that can only be relocated, but not copied nor moved. The `reloc` operator thus becomes necessary to safely pass around such objects.

Also, all these proposals (but P0308R0) aim to optimize the move and destruct operations into a single memcpy. But there are places where this optimization could not happen, and we are left with an suboptimized move and destruct. This is not tackled by those proposals.

Consider the following:

```
void sink(std::list<int> lst_c);

void fwd_to_sink(std::list<int> lst_b)
{
    sink(std::move(lst_b));
}

void bar()
{
    std::list<int> lst_a;
    std::fill_n(std::back_insertor(lst_a), 10, 0); // fill it
    sink(std::move(lst_a));
}
```

In some implementations, `std::list` move constructor makes a memory allocation to give the moved-from instance a new sentinel node. In such cases:

- `lst_a` is emptied and a new sentinel is allocated for it. `lst_a` is then destructed, deleting the previously allocated sentinel. Should `std::list` be trivially relocatable, this move and destruct could be optimized into memcpy, although not all proposals suggest this.
- Again, `lst_b` is moved to construct `lst_c` . However `fwd_to_sink` cannot make the memcpy optimization, as it cannot prevent the call to the destructor of `lst_b` , which is triggered by `bar` , which itself cannot know whether a memcpy optimization happened (imagine the functions to be in separate translation units).

This point is tackled by this proposal by the relocation constructor and the `reloc` operator.

1.2.1 P1144R5: Object relocation in terms of move plus destroy by Arthur O'Dwyer

The current proposal is fully compatible with P1144R5. We reuse the `[[trivially_relocatable]]` attribute, the `relocate_at` and `uninitialized_relocate` algorithms. The rules to deduct the trivially_relocatable trait are reused, although slightly modified to account for the new constructor.

However unlike P1144R5, the current proposal does allow users to define their own relocation function, which is done by the relocation constructor.

As we have seen the move constructor performs extra operations to ensure that the moved-from object remains at a valid state. Those operations are wasted if the object is destructed right after. Hence we believe we should allow users to provide their own relocation function (the relocation constructor) so that relocation and destruction can be better optimized than “move” and destruction.

1.2.2 P0023R0: Relocator: Efficiently moving objects by Denis Bider

The relocator introduced into P0023R0 is somewhat similar to the proposed relocation constructor.

P0023R0 allows optimizations in containers with trivial relocation, which the current proposal also does.

The key difference is that objects passed to the relocator must not be destructed. In fact the relocator can be viewed as a custom memmove operation.

In container implementations, the current proposal does not allow to customize the memmove operation like a relocator does. Instead the relocation constructor allows for a custom relocation function but the destructor of the relocated object must still be called. However the current proposal also enables the trivial relocation optimization, in which the call to the move (or relocation) constructor and destructor can be optimized into a single memcpy operation.

In contrast to P0023R0, the current proposal allows to pass variables around by relocation. This is not possible in P0023R0 in the general case (P0023R0 suggests the compiler may use it as its own discretion when a code flow analysis shows it's possible).

1.2.3 N4158: Destructive Move by Pablo Halpern

N4158 proposes a customizable function `std::uninitialized_destructive_move`.

- relocation can only happen if this function is called. Typically this function would be called in container implementation. We cannot relocate local variables with this.
- users can write their own `uninitialized_destructive_move` overload, but it looks more obscure than writing a relocation constructor.
- classes that have non-static data-member or inherit from classes that have a specialized overload of `uninitialized_destructive_move` do not get it for free. It is likely that writers of such classes will simply forget to write the overload.

For instance, although `std::list` may have an `uninitialized_destructive_move` overload, `struct S { std::list<int> _mylist; };` does not.

1.2.4 P1029R3: move = bitcopies by Niall Douglas

P1029R3 provides a special bitcopies move constructor.

The current proposal allows the same set of optimizations with trivial relocation.

Besides we feel that P1029R3 forces too strong constraints on the default and move constructors to enable this optimization. Indeed, classes that have a bitcopies move constructor are no longer allowed to have a non-constexpr default constructor or a move constructor with side effects, which may be useful on cases where the bitcopies optimization cannot happen.

P1029R3 could be made compatible with the current proposal with a bitcopies relocation constructor, but we feel that will miss some optimization cases.

Consider the `std::list` case:

- In some implementations, `std::list` move constructor makes a memory allocation to give the moved-from instance a new sentinel node. In those implementations, `std::list` could be trivially relocatable as the first and last node store no pointer back to the list itself (only to the sentinel node, which does not reside in `std::list` memory as it is heap-allocated) ;
- However `std::list` constructor is not constexpr so it cannot benefit from P1029R3 ;
- Even so if the constructor were to be marked constexpr, and if the move constructor could be defaulted to bitcopies, and in places where the second memcpy (`memcpy(&src, &T{}, sizeof(T))`) and destructor call cannot be alleviated, then the move constructor would still perform a memory allocation to put back a sentinel node.

The current proposal allows to reuse P1144R5 `[[trivially_relocatable]]` attribute. Hence library vendors can mark `std::list` with `[[trivially_relocatable]]` if it applies. If the `std::list` cannot be `[[trivially_relocatable]]` , **or in places where the trivial relocation optimization cannot apply**, then the proposal allows to write a new relocation constructor so that relocation can happen in a more optimized manner than with the move constructor.

1.2.5 P0308R0: Valueless Variants Considered Harmful by Peter Dimov

In P0308R0, we only look at the “pilfering” proposition. P0308R0's pilfering shares some similarities with the current proposal. P0308R0 is a pure library solution to relocation while this proposal provides a language solution.

We believe a language solution is better suited here:

- the `reloc` operator makes sure the relocated object is not reused, while `std::pilfer` does not ;

- the pilfering constructor is inconvenient to write as we need to unwrap from `std::pilfered` and rewrap to propagate to base classes and data-members ;
- the pilfering constructor cannot be defaulted ;
- trivial relocation is not possible with pilfering.

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 broken by move constructor

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 several ways, none of which are satisfying:

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*.

3. **The move constructor modifies the moved-from instance to lazily allocate new resources.** The moved `TempDir` lost its resources but does not break the class invariant. Instead, the first time the resource is queried, the instance will perform the allocation again. This is not a light change however, especially if the resources are accessed from a const method. In that case the resources would need to have the mutable flag and they would need to be allocated in thread-safe way, which is a heavy machinery for a silly case.

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*.

This case also highlights that some classes have no room for the moved-from state, it would just break their class design. And those classes still have a legitimate need to be moved around.

2.2 Case 2: Relocation in a single operation is faster than move and destruct

Many proposals about this topic have already pointed this out. As the move operation leaves the object in a valid state, the moved-from object must still be destructed. This comes at a cost, especially if you look at any `std::vector` implementation.

When the vector needs to grow, it will move (if possible) all its elements into a new larger vector. This requires a first iteration over all items. The previous vector is then destructed, triggering the destruction over all moved-from elements, which cause a second iteration over all items. While this could be optimized into a single large `memcpy` in many cases.

This proposal enables several optimizations:

- If the elements are "trivially relocatable" then they can be simply "memcpy-ed" into the new vector, and the old vector can simply drop its items without triggering their destruction.
- If the elements are relocatable, but the operation is not trivial, then we leave the chance to the user to implement its own relocation method so that relocation and destruction can be better optimized than move and destruction.

2.3 Drawbacks of the move constructor and rvalue-references

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 `std::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: Several proposals about relocation put the emphasis on performance optimization. While this is a fair point, we fear that some aspects are left short, especially class design. Notably **we argue that some**

classes could benefit from not being copyable, nor movable, but only relocatable. This is the case of classes that guarantee the ownership of some resources (like a wrapper around an open socket, or a non-null pointer, etc…) or other cases where the moved-from state would simply break the class design.

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.
3. a new kind of reference: the relocation reference. A relocation reference on a type `T` is denoted by: `T~`
4. a new attribute: `trivially_relocatable` , which allows compiler to optimize relocation + destruction into a single memcpy (like in p1144r5).

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] rhs TempDir instance to build *this from
     *
     * The new instance is built by stealing the resources from rhs.
     * rhs is left in a dirty state and cannot be reused after this call.
     *
     * TempDir~ is a relocation reference on a TempDir object.
     */
    TempDir(TempDir~ rhs) noexcept;

    /* The destructor must detect if the instance has been relocated, in which case
     * it will likely do nothing.
     */
    ~TempDir() noexcept;
};

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. */
}
```

3 Relocation constructor

The new relocation constructor is written as follows:

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

The relocation constructor creates a new instance by stealing the resources from `rhs` , and leaving it into a dirty invalid state. `rhs` can no longer be used, except when passed to its destructor. As `rhs` was emptied from its resources, its destructor is most likely a no-op. In fact, one could view the relocation constructor as a destructor for `rhs` . This is just a thought experiment, however, as the language still requires `rhs` to be passed to its destructor.

The relocation constructor leaves the object `rhs` in a dirty state. The only operation that is permitted on `rhs` after it was passed to the relocation constructor is its destructor call. Any other use of `rhs` is an undefined behavior.

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 undefined behavior.

Note: The relocation constructor is not intended to be directly called by users. Instead relocation must happen through the new `reloc` operator. The `reloc` operator ensures that:

- no object slicing will occur (e.g. if we were to relocate a derived class to a base class) ;
- the relocated object cannot be reused.

However it remains permitted to manually call the relocation constructor (given that it is accessible and not deleted) as useful on some cases, like manual memory management (e.g. `std::vector` implementation).

3.1 Trivially relocatable

The class definition or the relocation constructor can be accompanied by the `[[trivially_relocatable]]` attribute (taken from p1144r5). Like in p1144r5, the attribute may also be conditioned by a boolean parameter: `[[trivially_relocatable(bool)]]` .

```
class [[trivially_relocatable]] T
{
    // ...
};
// OR
class [[trivially_relocatable(some_constexpr_cond())]] T
{
public:
    // ...
};
// OR
class T
{
public:
    [[trivially_relocatable]] T(T~ other) noexcept = default;
};
// OR
class T
{
public:
    [[trivially_relocatable(some_constexpr_cond())]] T(T~ other) noexcept = default;
};
```

It is an error to set the attribute on both the class and the relocation constructor.

The trivially relocatable attribute gives the guarantee that both relocation and destruction operations can be safely optimized into a single `memcpy` operation. This optimization is not required to happen and is left at compiler vendors' discretion.

3.1.1 Trivially relocatable type deduction rules

As in p1144r5, the trivially relocatable attribute is deducted if not specified. We suggest to follow the deduction rules from p1144r5, slightly rephrased to account for the relocation constructor. This gives (changes from p1144r5 are written in italic):

A *relocation-constructible or* move-constructible, destructible object type T is a trivially relocatable type if it is:

- a trivially copyable type, or
- an array of trivially relocatable type, or
- a (possibly cv-qualified) class type declared with a [[trivially_relocatable]] attribute with value true, or
- a (possibly cv-qualified) class type which:
  - *has no user-provided relocation constructors,*
  - has no user-provided move constructors or move assignment operators,
  - has no user-provided copy constructors or copy assignment operators,
  - has no user-provided destructors,
  - has no virtual member functions,
  - has no virtual base classes,
  - all of whose members *(cv-qualifiers ignored)* are either of reference type or of trivially relocatable type, and
  - all of whose base classes are trivially relocatable.

3.2 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
};
```

3.3 Implicit, defaulted or deleted relocation constructor

The relocation constructor may be implicitly-declared, implicitly-defined, defaulted, or deleted just as any other constructor. The rules to define that follow the same logic as those of the move constructor.

4 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, local not ref-qualified variables). 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 and prevents programming errors where the relocated variable is reused.

4.1 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` and it is not a parameter of `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 (all of those are undefined, unaccessible or deleted).

4.1.1 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 anonymous
}

void foobar(const std::string& str)
{
    foo(reloc str); // ERROR: str is ref-qualified
}
void foobar(std::string* str)
{
    foo(reloc *str); // ERROR: *str is a pointer dereference
}
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
    }
};
```

4.2 reloc stages

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

1. Constructs a new object from `obj` using the relocation constructor (if available), the move constructor (if available) or the copy constructor.
2. 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.

4.2.1 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 that is accessible and not deleted, then does: `T{const_cast<T~>(obj)}` .
2. Otherwise if `T` defines a move constructor that is accessible and not deleted, then does: `T{const_cast<T&&>(std::move(obj))}` . `const_cast` is used to discard any cv-qualifiers ; as `obj` reached its end of life, we no longer care about those. In addition, this allows to use `reloc` on a const variable that have no relocation constructor defined.
3. Otherwise does: `T{obj}` .

Note: This stage can be optionally optimized. If `T` is trivially relocatable and that the call to the destructor of `obj` can be avoided entirely, then both this stage and the destructor call can be replaced by a simple `memcpy` operation. This optimization is left at the discretion of compiler vendors and is not enforced by the proposal. This proposal merely authorizes the optimization.

4.2.2 Stage 2: 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 branch 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, but `foo(test() ? reloc x : std::move(x));` does not.

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` .

4.3 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`).

Note: if the object is trivially relocatable then this operation can optimized into a memcpy. In which case the destructor of the relocated object must not be called. This optimization is authorized by the proposal but left at compiler vendors’ discretion.

4.4 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.

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

Note: this operation could also be optimized into a memcpy, given that the object is trivially relocatable. In which case the destructor of the temporary must not be called. This optimization is authorized by the proposal but left at compiler vendors' discretion.

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
}
```

4.5 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;
    }
};
```

Note: if `obj` is trivially relocatable and that its destructor call can be avoided, then placement reloc can be optimized into a single memcpy operation. This optimization is authorized by the proposal but left at compiler vendors' discretion.

4.6 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 is ref-qualified
}
void foo2(int (&x)[N])
{
    int y[N] = reloc x; // ERROR: x is ref-qualified
}
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` .

Note: if the element type of the array is trivially relocatable and that the array destructor call can be avoided, then the array relocation can be optimized into a single large memcpy operation over all its elements in one go. This optimization is authorized by the proposal but left at compiler vendors' discretion.

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 reached its end of life (consequence of the relocation), and which can then (after the relocation happened) only be destructed. 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.

This point enables the relocation of const objects.

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 parts 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, and that each I-th object can be constructed from the returned value of the selected `get<I>` function.

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 relocatable* and only have public data-members (static data-members are ignored).

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_relocatable;

template<class T>
inline constexpr bool is_relocation_constructible_v =
    is_relocation_constructible<T>::value;
template<class T>
inline constexpr bool is_trivially_relocatable_v =
    is_trivially_relocatable<T>::value;
}
```

6.2 Algorithm

6.2.1 relocate_at

```
namespace std
{
template <class T>
T* relocate_at(const T* src, T* 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. Returns the address of the new instance (i.e. `dst`).

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

```
template <class T>
T* relocate_at(const T* src, T* dst)
{
    memmove(dst, src, sizeof(T));
    return std::launder(dest);
}
```

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

```
template <class T>
T* relocate_at(const T* src, T* dst)
{
    T* res = std::construct_at(dst, const_cast<T~>(*src));
    std::destroy_at(src);
    return res;
}
```

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

Otherwise it behaves as if by:

```
template <class T>
T* relocate_at(const T* src, T* dst)
{
    try {
        T* res = std::construct_at(dst, consts_cast<T&&>(std::move(*src)));
        std::destroy_at(src);
        return res;
    } catch (...) {
        std::destroy_at(src);
        throw;
    }
}
```

6.2.2 uninitialized_relocate

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

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

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

template<class ExecutionPolicy, class InputIt, class Size, class ForwardIt>
std::pair<InputIt, ForwardIt> uninitialized_relocate_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_relocate` : an iterator to the element past the last element relocated;
- `uninitialized_relocate_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 relocatable and both iterator types are contiguous, it behaves as if by:

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

    if (last != first)
        std::memmove(std::addressof(*d_first),
            std::addressof(*first),
            std::distance(first, last)*sizeof(value_type));

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

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

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_relocate(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(std::addressof(*d_first),
            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_relocate(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::construct_at(std::addressof(*d_first), const_cast<value_type~>(*first));
        std::destroy_at(std::addressof(*first));
    }

    return d_first;
}
```

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

Last it behaves as if by:

```
template<class InputIt, class ForwardIt>
ForwardIt uninitialized_relocate(InputIt first, InputIt last, ForwardIt d_first)
{
    try {
        for (; first != last; ++d_first, (void) ++first) {
            std::construct_at(std::addressof(*d_first), std::move(*first));
            std::destroy_at(std::addressof(*first));
        }
    } catch (...) {
        std::destroy(first, last);
        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 pilfer was already called.
     */
    T pilfer();
};
}
```

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.pilfer()` (resp. `y.pilfer()`).

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>::pilfer();
```

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 = std::forward<T>(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 a parameter and the item to insert as next parameter (taken by value).

`std::relocate_t` is here to help distinguish from otherwise ambiguous overloads. Indeed `vec.push_back(reloc a);` would call `vector::push_back(T&&)` and the item `a` won't be relocated inside the vector. If we added `void vector::push_back(T val)` as overload then the previous call would 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 "pilfer" 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>::pilfer_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>::pilfer(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_out(
    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>::pilfer_back();
// removes the first item from the queue and returns it
template <class T, class Alloc>
T deque<T, Alloc>::pilfer_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>::pilfer(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_out(
    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>::pilfer_back();
// removes the first item from the list and returns it
template <class T, class Alloc>
T list<T, Alloc>::pilfer_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>::pilfer(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_out(
    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>::pilfer_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>::pilfer_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::pilfer(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::pilfer();
```

6.5 Iterator library

6.5.1 reloc_insert_iterator

`std::reloc_insert_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::reloc_insert_iterator` is a no-op.

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

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

We also suggest adding alternatives for `push_back` and `push_front` (replace xxx respectively by back and front):

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

```
template< class Container >
std::reloc_xxx_iterator<Container> reloc_xxx_inserter( Container& c );
```

`reloc_xxx_inserter` is a convenience function template that constructs a `std::reloc_xxx_iterator` for the container `c` 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 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`

Library	Classes
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 as 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 a 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 relocated value is not reused. Otherwise things would not be much different from the move constructor. This is the role of the `reloc` operator.

Also, we believe that trivial relocation is better achieved with language support (as claimed in P1144R5) than with a library solution.

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. It would allow to write code such as `T& y = reloc x;` (the relocation reference can be implictly cast to any reference), where `x` is no longer reachable after this call yet we allowed to bind a reference to it at the same time. It would also favor codes like `void foo(T~); void bar() { T x; foo(reloc x); }` . Although it might seem like a good idea, it also inhibits the trivial relocation optimization (x in bar cannot be memcpyed to foo, since foo expects a reference).

7.4 Why a new operator?

A relocated variable 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. It also conveys the clear intent of the developper.

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 Can reloc operator be overloaded?

We also considered the operator reloc overload in place of the relocation constructor. We considered the following code:

```
class T
{
public:
    operator reloc(T&& rhs) noexcept = default;
};
```

operator reloc would return a new instance of T, so its return type is omitted. This looks nice when defaulted, but it becomes tedious to write an implementation.

- operator reloc would somehow need to construct the object it will return. How to do this in an optimized way other than calling a constructor dedicated to relocation? This would force the user to write their own unofficial relocation constructor that reloc would simply call.
- if operator reloc always needs to call some constructor, why not give it the right to construct the object on its own, with delegating constructor and initialization list? Then reloc could be written as:

```
class T : public Base
{
    U a;
public:
    operator reloc(T&& rhs) noexcept : Base{std::move(rhs)}, a{std::move(rhs.a)} {}
};
```

But this calls the move constructor of `Base` and `a` which may not be what we want. We came up with this new syntax:

```
class T : public Base
{
    U a;
public:
    operator reloc(T&& rhs) noexcept : Base reloc{std::move(rhs)},
        a reloc{std::move(rhs.a)} {}
    // Base reloc{std::move(rhs)} would construct Base from its own reloc operator
};
```

This has the benefit of introducing no new constructor (even though this reloc operator could be considered as one), and no new type of reference. But:

- it provides a new syntax to call a specific constructor: `T reloc{some_ref}` . This clashes with C++ style that relies on overload resolution to select the right constructor.
- it allows for weird code: `T x; T y reloc{std::move(x)};` here `y` is created by relocation but `x` has not been relocated by `reloc` , so it can still be used.
- such unsafe code can be written with the current proposal with: `T x; T y{const_cast<T~>(x)};` but at least the `const_cast` call makes it obvious that this is unsafe.

For all these reasons, we decided to go with the relocation constructor and relocation reference. The reloc operator cannot be overloaded.

7.7 Why use const_cast to cast into a relocation reference?

Casting to a relocation reference discards any cv-qualifiers so `const_cast` feels appropriate. Also `const_cast` gives a sense of caution that's needed when handling a relocation reference.

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` to limit the lifetime of `b` . 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 name the extract functions pilfer and not extract in STL 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 `pilfer` instead. We prefer to have the same API across all containers to make it easier to write generic code.

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).

8 References

- https://herbsutter.com/2020/02/17/move-simply/
- P1144R5: Object relocation in terms of move plus destroy by Arthur O'Dwyer

  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1144r5.html
- P0023R0: Relocator: Efficiently moving objects by Denis Bider
  - http://open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0023r0.pdf
- N4158: Destructive Move by Pablo Halpern
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4158.pdf
- P1029R3: move = bitcopies by Niall Douglas
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p1029r3.pdf
- P0308R0: Valueless Variants Considered Harmful by Peter Dimov
  - http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0308r0.html