Document #: P2187R4
Date: 2020-08-15
Project: ISO SC22/WG21 Programming Language C++
std::swap_if, std::predictable

Title: std::swap_if, std::predictable
Reply-to: Nathan Myers <ncm@cantrip.org>
Target: C++23
Audience: SG18 LEWGI

This paper proposes new Standard Library primitives `swap_if` and `iter_swap_if`, currently used implicitly (but very sub-optimally) in nearly half of the Standard Library algorithms, and equally useful for users' algorithms, and equally useful for users' algorithms. Although trivial to code correctly, current shipping compilers generate markedly sub-optimal code for naïve implementations.

In addition, it proposes a means to indicate to Standard Library facilities that the results of an ordering predicate in a particular use have turned out to be predictable, so that a more appropriate variant of the algorithm may be substituted. Finally, it defines a customization point to identify types with non-trivial special members that nonetheless qualify for optimization.

History

P2187R4 Fix some `noexcept` specifications, use ADL `swap(x,y)`.

P2187R3 Replace `cheaply_copyable` concept with `cheaply_swappable`, and provide customization point `is_trivially_swappable_v`, to extend applicability to "pimpl" and `uniq_ptr` types. Add `noexcept`. Mention P1144.

P2187R2 Fix numerous code errors, archaisms; replace bool `is_cheaply_copyable` with concept `cheaply_copyable`.

P2187R1 Add `iter_swap_if`; restrict non-branching optimization to small `T`; use `predictable_bool` wrapper; remove `is_predictable`, add `is_cheaply_copyable`; add Example, Acknowledgements, Open Issues; evolve WP text.

P2187R0 In delayed version of 2020-06-15 mailing.

Introduction

Almost half of the algorithms in `std`, exemplified by `std::sort`, depend on a conditional-swap operation, used in circumstances where the *condition* is typically not especially predictable. For example, the conditional-swap operation might appear in the body of a partition loop as:

```
if (*right < pivot) {
  std::swap(*left, *right);
  ++left;
}
```

Testing reveals that the performance of such algorithms can be improved by more than a *factor of two*[1][2] simply by changing the implementation of conditional-swap to avoid the pipeline stalls that follow branch mispredictions. (It may be surprising, even hard to believe, that the naïve conditional-swap above often results in very poor algorithm performance. Please consult the references in case of doubt.[3])

This paper proposes new library algorithms `swap_if` and `iter_swap_if`. A portable implementation of `swap_if` for `int`, coded to avoid the worst hardware inefficiencies, might look like:

```
bool swap_if(bool c, int& a, int& b) {
  T tmp[2] = { a, b };
  b = tmp[1-c], a = tmp[c];
  return c;
}
```

Because it performs identically the same sequence of instructions for both possible values of its `bool` argument, varying only the array indices, it avoids a likely mis-predicted branch and pipeline stall. In the the partition loop mentioned above, it might be called as:

```
left += swap_if(*right < pivot, *left, *right);
```

Discussion

Testing demonstrates that the best implementation of `swap_if` for scalar values on common modern hardware uses `cmov` instructions. However, no mainstream compiler emits `cmov` instructions to implement the generic `swap_if` coded above. One compiler was found to peephole-optimize this version to use `cmov`:

```
bool swap_if(bool c, int& a, int& b) {
  int ta = a, m = -c;
  a = m&b|~m&a, b = m&ta|~m&b;
  return c;
}
```

Other compilers checked produce markedly sub-optimal code for both alternatives.

Despite such sub-optimal code generation, however, a `sort` implemented using either `swap_if` above, and built with current shipping compilers, nonetheless strongly outperforms the `std::sort` provided in current Standard Library implementations, when applied to random scalar input.

The suggested implementation is an improvement over the naïve version when copying a `T` is cheap. More precisely, it is better when two load/store pairs and a pipeline stall take longer than four unconditional load/stores, or (moreso) four loads and two taken `cmov` stores. Therefore, this proposal specifies a non-branching variant to be used only when `T` really is trivially copyable, and small enough. Excess *observable* copies or moves are likely to be too expensive, and anyway wrong. Therefore, we gate access to the non-branching `swap_if` with a restrictive concept:

```
template <typename T>
  constexpr bool is_trivially_swappable_v =
    std::is_trivially_copyable_v<T>;

template <typename T>
  concept cheaply_swappable =
    (std::is_trivially_swappable_v<T> && sizeof(T) <= N);

template <typename T>
    requires (cheaply_swappable<T> || std::swappable<T>)
  constexpr bool swap_if(bool c, T& x, T& y)
    noexcept(cheaply_swappable<T> || std::is_nothrow_swappable_v<T>)
  {
    if constexpr (cheaply_swappable<T>) {
      log('u');
      struct alignas(T) { char b[sizeof(T)]; } tmp[2];
      std::memcpy(tmp[0].b, &x, sizeof(x));
      std::memcpy(tmp[1].b, &y, sizeof(y));
      std::memcpy(&y, tmp[1-c].b, sizeof(x));
      std::memcpy(&x, tmp[c].b, sizeof(x));
      return c;
    }
    if (c) swap(x, y);
    return c;
  }
```

Implementations will determine the size limit *N* where the cost of excess copies performed in a constant-time conditional swap operation exceeds the savings from avoiding pipeline stalls. We expose `is_trivially_swappable_v` and `cheaply_swappable` for the benefit of user algorithms, where they are equally as useful as in the Standard Library. The `is_trivially_swappable_v` detour seen above will turn out to be useful. (Note that P1144 appears to be identical in purpose to `is_trivially_swappable_v`. It will be retained in this proposal until P1144 enters the WP. `cheaply_swappable` will remain. Note that any `cheaply_relocatable` would have a different `N`.)

It should be straightforward to peephole-optimize the `swap_if` above so that `cmov` instructions are used where possible. Once `std::swap_if` is provided in the

Standard Library, and used, implementers might find motivation to implement it optimally, yielding further performance gains; and might begin to use it in their Standard Library algorithms, improving them as well.

With such performance gains to be had, why are implementers not already doing this, internally, in their Standard Library code? First, the opportunity has been poorly understood until recently. More subtly, *some* programs, when choices turn out to have been predictable, are slower: users rarely thank implementers for faster programs, but complain bitterly about each slower one, labeling it a *regression*. Implementers have been burned after generating `cmov` instructions in what turned out to be predictable contexts[4]. Changing the performance characteristics of fundamental algorithms without prompting from the Standard seems too risky for most implementers to do on their own initiative, whatever the benefits for users.

Users who discover a speed regression need an answer. Which leads to...

`predictable<Predicate>`

Order-dependent algorithms are sometimes used in circumstances where the comparison results turn out to be highly predictable. (The threshold of predictability for which an otherwise sub-optimal branching `swap_if` implementation is preferred is north of 90% on current hardware.) Some users will then find that a library algorithm, typically much faster when implemented with a branchless `swap_if`, turns out to be slower for their particular data. They need a practical, portable way to roll back to a branching variant without changing their type `T`.

To that end, we additionally propose a predicate wrapper, `std::predictable`:

```
template <typename Predicate>
  struct predictable {
    Predicate pred;  // for exposition only
    predictable(Predicate&& p) : pred(std::move(p)) {}
    template <typename ...Args>
        requires predicate<Predicate, Args&&...>
      auto operator()(Args&&... args) -> predictable_bool
        { return bool(invoke(pred, (Args&&)args...)); }
    // +overloads
  };
```

Its `op()` just forwards its arguments to the predicate, but converts the result to `std::predictable_bool`, itself just a wrapper for `bool`:

```
struct predictable_bool {
  bool value{};
  predictable() = default;
  predictable_bool(bool v) : value(v) {}
  operator bool() { return value; }
};
```

For any predicate `P` such that `std::predicate<P, Args...>` is satified, so does also `std::predicate<std::predictable<P>>`.

Finally, we need an overload of `swap_if` for `predictable_bool`:

```
template <std::swappable T>
  bool swap_if(predictable_bool c, T& x, T& y) {
    if (c) swap(x, y);
    return c;
  }
```

Standard library components that take a predicate argument may be passed a predicate wrapped in `std::predictable` as a way to request that the implementation use a conditional-branching `swap_if` in preference to the default, branchless version normally used for small, simple objects; and make any other appropriate accommodation. It might be used like:

```
auto v = std::vector{ 3, 5, 2, 7, 9 };
std::sort(v.begin(), v.end()); // unpredicted
std::sort(v.begin(), v.end(),  // predicted
  std::predictable([](int a, int b) { return a > b; }));
```

Note that the wrapper does *not*, itself, affect the predicate implementation, or even (usually) how it is applied. It is purely a medium to conduct the caller's expectation of predictability deep into the algorithm's implementation, and there help to choose what it may presume has been carefully measured to be the best implementation for the input data coming. This doubles the number of such algorithm implementation variants available without need to invent and expose numerous new names for them. Furthermore, it parameterizes the choice so it is easier to use in generic code than differently-named algorithms would be.

The formal semantics of all library components are unaffected by `predictable`-wrapping, so their descriptions in the Standard are unchanged. Better-quality implementations will provide variants of each affected algorithm, visible only by improved performance when used correctly.

"Pimpl" types

Many object types have non-trivial default-construction, assignment, and copy-construction semantics, yet can safely be bitwise-swapped. The canonical example is a "pimpl" type, which may wrap a simple pointer. None of the regular operations on such a type are trivial, yet bitwise-swapping the wrapped pointer is always safe (modulo tearing).

`swap_if` would be substantially more valuable if it could be used on these types, too. To that end, the trait defined above, `is_trivially_swappable_v`, may be declared a customization point. We can customize certain existing Standard Library types as trivially swappable, too, notably `unique_ptr`:

```
template <typename T, typename D>
  constexpr bool is_trivially_swappable_v<std::unique_ptr<T,D>> = true;
```

Now `unique_ptr` can be operated on as fast as ordinary scalars. Users can specialize this for their own types, as well. Of course it would be UB to specialize it for certain types, but remarkably many qualify. (This becomes unnecessary if P1144 gets in. In that case, `cheaply_swappable` would be defined in terms of `is_trivially_relocatable_v` instead.)

Example

Here is how these facilities might be used in an implementation of a representative algorithm[5].

`iter_swap_if` is trivial, expressed minimally:

```
  template <typename Flag, typename I>
    bool iter_swap_if(Flag c, I p, I q) { return swap_if(c, *p, *q); }
```

Here is a minimal `partition` that uses `iter_swap_if`. Notice that this is the lowest level that uses the argument predicate. The predictability expectation is carried down into `iter_swap_if`, then `swap_if`, via the type of the predicate's result:

```
  template <permutable I, typename Predicate>
      requires sortable<I, Predicate>
    constexpr auto partition(I b, I e, Predicate&& pred) {
      --e;
      auto pivot = std::move(*e);
      I left = b;
      for (I right = b; right < e - 1; ++right) {
        auto do_swap = std::invoke((Predicate&&)pred, *right, pivot);
        left += std::iter_swap_if(do_swap, left, right);
      }
      *e = std::move(*left);
      *left = std::move(pivot);
      return left;
    }
```

Finally, at top level, a minimal `sort` that uses the partition:

```
  template <permutable I, predicate Predicate>
      requires sortable<I, Predicate>
    constexpr void sort(I begin, I end, Predicate&& pred) {
      while (end - begin > 1) {
        I mid = partition(begin, end, (Predicate&&)pred);
        std::sort(begin, mid, (Predicate&&)pred);
        begin = mid + 1;
```

```
    }
  }
```

Note particularly that that at intermediate levels, hardly any accommodation is needed, in the code, to select the correct version of `swap_if` at the bottom-most level; seen here is only that, in `partition`, the temporary `do_swap` is declared `auto` instead of `bool` to propagate a `predictable_bool` should it ever appear. Wrappers do the work.

Other primitives

This proposal offers WP text for `swap_if` and `iter_swap_if`. Other primitives used in common algorithms would certainly benefit from similar treatment, most notably `select`, used when descending binary trees and in binary search. Worked examples of this and others are solicited. (E.g.: `minmax`, `push_heap`, `pop_heap`.)

Proposed WP Text

In **25.8 Sorting and related operations** [**alg.sorting**], add a new subsection:

25.8.x Predictability . **[predictability]**

```
namespace std {
  template <typename T>
    constexpr bool is_trivially_swappable_v;  // customization point

  template <typename T>
    concept cheaply_swappable;

  struct predictable_bool;

  template <typename Predicate>
    struct predictable;

  template <typename T>
    constexpr bool swap_if(bool c, T& x, T& y);
  template <swappable T>
    constexpr bool swap_if(predictable_bool c, T& x, T& y);

  template <typename I>
    constexpr bool iter_swap_if(bool c, I p, I q);
  template <indirectly_swappable I>
    constexpr bool iter_swap_if(predictable_bool c, I p, I q);
}
```

1. The utilities defined here aid in modulating how algorithms rely on, or

ignore, the predictability of the results of calls to their predicate arguments, where prediction derives from the recent runtime history of such results.

2. It is intended that, when the element type operated on is *trivially-copyable* and small, the runtime performance of algorithms `swap_if` and `iter_swap_if` variants that take a `bool` argument *should not* depend strongly on the value of `c`; and that of the variants that take a `predictable_bool` or are called on larger elements *may* depend on the runtime value and predictability of `c`.

25.8.x.1 Constant `is_trivially_swappable_v` [is.trivially.swappable]

```
template <typename T>
  constexpr bool is_trivially_swappable_v =
    is_trivially_copyable_v<T>;

template <typename T, typename D>
  constexpr bool is_trivially_swappable_v<unique_ptr<T,D>> = true;
```

1. `is_trivially_swappable_v` is a customization point for use to declare that a type representation is safe to swap bitwise without using member functions. By default, it matches `is_trivially_copyable_v`, but may be customized to encompass a wider range of types.
2. `is_trivially_swappable_v` is customized for the case of `unique_ptr`.

25.8.x.2 Concept `cheaply_swappable` [cheaply.swappable]

```
template <typename T>
  concept cheaply_swappable =
    (std::is_trivially_swappable_v<T> && sizeof(T) <= N);
```

1. This concept is intended to aid algorithm selection favoring constant-time operations, for improved runtime performance, when operating on small, simple element object types *T*.

2. The value of *N* small enough for `T` to participate usefully in unpredicted operations is platform dependent, and is therefore unspecified in this Standard. *[Note: Implementations should choose a value N such as to maximize performance of Standard Library algorithms called with predicates P not wrapped as `predictable<P>`. For some build targets the correct value will be zero. – end Note]*

25.8.x.3 Struct `predictable_bool` [predictable.bool]

```
struct predictable_bool {
  bool value{};
  constexpr predictable_bool() = default;
  constexpr predictable_bool(bool v) : value(v) {}
```

```
  constexpr operator bool() { return value; }
};
```

1. This is a wrapper for a `bool` value, to indicate to an algorithm it is passed to that its value derives from a process expected to be usefully predictable, for purposes of optimization.

25.8.x.4 Function object template `predictable` [predictable]

```
template <move_constructible Predicate>
  struct predictable {
    using type = Predicate;
    Predicate pred{};              // name for exposition only
    constexpr predictable() = default;
    constexpr predictable(Predicate&& p) : pred(std::move(p)) {}
    constexpr predictable_bool operator()(auto&&... args) const;
    constexpr predictable_bool operator()(auto&&... args);
    constexpr predictable_bool operator()(auto&&... args) const&&;
    constexpr predictable_bool operator()(auto&&... args) &&;
  };
```

1. *Remarks:* `predictable` wraps argument predicate `p`, changing only its result type, to `predictable_bool`, as a means to indicate to functions passed the result that it should be treated as a predictable influence on program flow, for the purpose of selecting algorithm implementations most suitable for expected runtime conditions.

2. *[Note:* `predictable` *is provided for use in circumstances where it has been determined that the default behavior of algorithms, as applied to expected runtime patterns in input data, is sub-optimal. – end note]*

3. *[Example:*

```
  auto v = std::vector{ 3, 5, 2, 7, 9 };
  std::sort(v.begin(), v.end()); // unpredicted
  std::sort(v.begin(), v.end(),  // predicted
    std::predictable([](int a, int b) { return a > b; }));
```

—end example]

25.8.x.4.1 Constructor `predictable` [predictable.ctor]

```
constexpr predictable(Predicate&& p) : pred(std::move(p)) {}
```

1. *Effect:* Move-constructs argument `p` into member `pred`.

25.8.x.4.2 `predictable<T>::operator()` [predictable.invoke]

```
template <typename ...Args>
    requires predicate<Predicate const&, Args&&...>
```

```
  constexpr auto operator()(Args&&... args) const
    noexcept(is_nothrow_invocable_v<Predicate const&, Args&&...>)
      -> predictable_bool
    { return bool(std::invoke(pred, (Args&&)(args)...)); }

template <typename ...Args>
    requires predicate<Predicate&, Args&&...>
  constexpr auto operator()(Args&&... args)
    noexcept(is_nothrow_invocable_v<Predicate&, Args&&...>)
      -> predictable_bool
    { return bool(std::invoke(pred, (Args&&)(args)...)); }

template <typename ...Args>
    requires predicate<Predicate const&, Args&&...>
  constexpr auto operator()(Args&&... args) const &&
    noexcept(is_nothrow_invocable_v<Predicate const&&, Args&&...>)
      -> predictable_bool
    { return bool(std::invoke(pred, (Args&&)(args)...)); }

template <typename ...Args>
    requires predicate<Predicate&, Args&&...>
  constexpr auto operator()(Args&&... args) &&
    noexcept(is_nothrow_invocable_v<Predicate&, Args&&...>)
      -> predictable_bool
    { return bool(std::invoke(pred, (Args&&)(args)...)); }
```

1. *Effect:* Invokes member `pred`, passing arguments by perfect forwarding.
2. *Returns:* The result of invocation, as converted to `bool`, thence to `predictable_bool`.

25.8.x.5 `swap_if` [swap.if]

```
template <typename T>
    requires (cheaply_swappable<T> || std::swappable<T>)
  constexpr bool swap_if(bool c, T& x, T& y)
    noexcept(cheaply_swappable<T> || is_nothrow_swappable_v<T>)
```

1. *Effects:* If `cheaply_swappable<T>`, then `x` and `y` representations are swapped as if by `memcpy` if and only if `c` is `true`. Otherwise, `swap_if(predictable(c), x, y)`.
2. *Returns:* `c`.
3. *Remarks:* If `cheaply_swappable<T>`, should execute in time minimally dependent on `c`.

```
template <swappable T>
  constexpr bool swap_if(predictable_bool c, T& x, T& y)
    noexcept(is_nothrow_swappable_v<T>)
```

1. *Effects:* `if (bool(c)) swap(x, y)`.
2. *Returns:* `c`.

25.8.x.6 `iter_swap_if` [iter.swap.if]

```
template <typename I>
    requires (cheaply_swappable<iter_value_t<I>> ||
      indirectly_swappable<I>)
  constexpr bool iter_swap_if(bool c, I p, I q) noexcept;
```

1. *Effects:* If `cheaply_swappable<iter_value_t<I>>`, then `::std::swap_if(c, *p, *q)`; otherwise, `if (c) ::std::iter_swap(p, q)`.
2. *Returns:* `c`.

```
template <typename I>
    requires indirectly_swappable<I>
  constexpr bool iter_swap_if(predictable_bool c, I p, I q)
    noexcept(is_nothrow_swappable_v<iter_value_t<I>>);
```

1. *Effects:* `if (bool(c)) ::std::iter_swap(p, q)`.
2. *Returns:* `c`.

Acknowledgements

Thanks to Adam Martin for the array-indexed implementation suggestion, to Bryan St. Amour for suggesting `iter_swap_if`, to Arthur O'Dwyer for design improvements and education, Gašper Ažman for overloading and noexcept advice, and to Peter Dimov, Mark Glisse, Howard Hinnant, Corentin Jabot, Tomasz Kamiński, and Jon Wakely for helpful discussion and advice.

References

[1]: https://arxiv.org/abs/1604.06697

[2]: http://gitlab.com/ncmncm/sortfast/

[3]: http://cantrip.org/sortfast.html

[4]: https://gcc.gnu.org/bugzilla/show_bug.cgi?id=56309

[5]: http://gitlab.com/ncmncm/wg21-p2187-swap_if