```
Document #:  P2187R0
Date:        2020-06-18
Project:     ISO SC22/WG21 Programming Language C++
Title:       std::swap_if, std::predictable
Reply-to:    Nathan Myers <ncm@cantrip.org>
Target:      C++23
Audience:    SG18 LEWGI
```

`std::swap_if`, `std::predictable`

This paper proposes a new Standard Library primitive `swap_if`, currently used implicitly (but very suboptimally) in nearly half of Standard Library algorithms, and equally useful for users' algorithms. Although `swap_if` is 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.

History

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

Introduction

Almost half 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 a new library algorithm, `swap_if`. A fully generic implementation of `swap_if`, coded to avoid the worst hardware inefficiencies, might look like:

```
template <movable T>
bool swap_if(bool c, T& a, T& b) {
  T tmp[2] = { move(a), move(b) };
  b = move(tmp[1-c]), a = move(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` as coded above. One compiler peephole-optimized an explicit specialization on `int`:

```
template <> bool swap_if<int>(bool c, int& a, int& b) {
  int ta = a, tb = b;
  a = -c&tb | c-1&ta, b = -c&ta | c-1&tb;
  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.

Once `std::swap_if` is provided in the Standard Library, and finds use in user algorithms, implementers might be motivated to implement it optimally, yielding further performance gains; and to use it in Standard Library algorithms, improving them as well. It should be straightforward to peephole-optimize the generic `swap_if` above so that `cmov` instructions are emitted, where supported, in cases where `T` fits in a register.

A Refinement

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-suboptimal branching `swap_if` implementation is preferred is north of 90% on current hardware.) Users may then find that a library algorithm, usually much faster when implemented with a branchless `swap_if`, turns out to be slower for their particular data. They need a convenient way to roll back to branching version.

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

```
template <predicate Predicate, bool is = true>
struct predictable {
  std::remove_reference<Predicate>::type
    pred;  // name for exposition only

  explicit predictable(Predicate&& p) : pred(p) {}

  template <typename... Args>
    constexpr bool operator()(Args&&... args) {
      return ::std::invoke(p, args...);
  }
};
```

whose `op()` simply forwards its arguments to the captured predicate; and a corresponding customization-point trait, `std::is_predictable`:

```
template <typename> constexpr bool is_predictable = false;
template <predicate P, bool is>
  constexpr bool is_predictable<predictable<P,is> = is;
```

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, and any other accommodations that seem appropriate. 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 annotation does *not*, itself, affect the predicate implementation, or even 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 is hoped to be the best implementation for the input data being operated on. 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 the outcome of `is_predictable`, 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.

Other primitives

There may be other primitives used in common algorithms that would benefit from a similar treatment. Worked examples to add to this proposal are solicited. (Possible: `select`, `rotate_if`)

Proposed WP Text

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

25.8.x Predictability . [predictability]

```
namespace std {
  template <typename T>
    constexpr bool is_predictable;

  template <predicate Predicate, bool is = true>
    struct predictable;

  template <bool predicting, movable T>
    constexpr bool swap_if(bool c, T& x, T& y);
  template <movable T>
    constexpr bool swap_if(bool c, T& x, T& y);
}
```

1. The utilities defined here aid in modulating how algorithms rely on 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 the runtime performance of `swap_if`, according as its `bool` template argument is `true` or `false`, respectively, *should* or *should not* depend strongly on the runtime predictability of `c`.

25.8.x.1 Variable template `is_predictable` [is.predictable]

```
namespace std {
  template <typename T>
    constexpr bool is_predictable = false;

  template <predicate P, bool is>
    constexpr bool is_predictable<predictable<P, is>> = is;
}
```

1. The name `is_predictable` denotes a customization point object [**customization.point.object**].
2. *[Note: Its value may be used to select, during translation, an algorithm variant appropriate to a caller's expectation for the runtime behavior of the predicate* `P`*. –end note] [Example:*

```
    left += swap_if<is_predictable<Pred>>(
      std::invoke(pred, *right, pivot), *left, *right);
```

—end example]

25.8.x.2 Class template `predictable` [predictable]

```
namespace std {
  template <predicate Predicate, bool is = true>
    struct predictable {
      std::remove_reference<Predicate>::type
         !pred!;  // name for exposition only

      explicit predictable(Predicate&& p) : pred(p) {}

      template <typename... Args>
        constexpr bool operator()(Args&&... args) {
          std::invoke(pred, args...);
      }
      template <typename... Args>
        constexpr bool operator()(Args&&... args) const {
          std::invoke(pred, args...);
      }
  };
}
```

1. `predictable` is a wrapper used to indicate to a recipient that the result of the predicate should be treated as highly predictable, in circumstances where such predictability may have a pronounced effect on the runtime performance of the recipient.

2. *[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.2.1 Constructor . [predictable.ctor]

```
explicit predictable(Predicate&& p) : pred(p) {}
```

1. *Effects:* Moves `p` into `pred`.

25.8.x.2.2 Member `operator()` [predictable.call]

```
template <typename... Args>
  constexpr bool operator()(Args&&... args);
```

```
template <typename... Args>
  constexpr bool operator()(Args&&... args) const;
```

1. *Effects:* Invokes member `pred` passing `args` by perfect forwarding **[func.require]**.

25.8.x.3 `swap_if` **[swap.if]**

```
template <bool predicting, movable T>
  constexpr bool swap_if(bool c, T& x, T& y);
```

1. *Effects:* If and only if `c` is `true`, `x` takes the previous value of `y` and `y` takes the previous value of `x`, as if by move-assignment. May call `swap(x, y)` if `T` models `swappable` and `predicting` is `true`.
2. *Returns:* `c`.
3. *Remark:* An implementation without branching is preferred for the case where `predicting` is `false`.

```
template <movable T>
  constexpr bool swap_if(bool c, T& x, T& y);
```

1. *Returns:* `::std::swap_if<false,T>(c, x, y)`.

References

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

[2]: http://github.com/ncm/sortfast/

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