Motivation

Currently all library based type-cast wrappers are behaving as functions which mark their arguments as "consumed" in a compiler and they subsequently won't emit any warning if the resulting values are not used. Yet this is not the case of the language casts themselves (on a higher warning level.)

This paper proposes adding the attribute `[[nodiscard]]` to all library based cast function templates (which only wrap language casts) as they are meaningless without accessing the resulting value. Ignoring the return value of any of these, such as `std::move`, is most likely an error.¹

Affected function templates

- `to_integer`
- `forward`
- `move`
- `move_if_noexcept`
- `as_const`
- `to_underlying`
- `identity`
- `bit_cast`

Why these?

All selected function are based on only `static_cast` / `const_cast` / `reinterpret_cast`. Also they have no side-effects.

Comparison table

<table>
<thead>
<tr>
<th></th>
<th>before</th>
<th>after</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>void fnc(T val) {}  </code></td>
<td><code>val;                                          </code></td>
<td><code>// warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td><code>static_cast&lt;T &amp;&amp;&gt;(val);                     </code></td>
<td><code>// warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td><code>std::move(val);                              </code></td>
<td><code>// no warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>// warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>// warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>// no warning with -Wall</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>// warning in default mode</code></td>
</tr>
</tbody>
</table>

Implementation experience

Microsoft's STL already marks most of the standard library with `[[nodiscard]]`.

This paper doesn't propose

Adding `[[nodiscard]]` attribute to any other library construct which doesn't have semantics same as a language type casting.

This is also excluding `std::any_cast` as this function is not an equivalent to a simple language cast and is outside of scope of this proposal.

Future work

I would like to update all possible library functions and mark them `[[nodiscard]]` if they does not have any observable side-effects and not using the result value of a call would lead to removing the call away by a compiler.

This is already done by Microsoft's STL and in our internal experience it lead to a discovery of old long-existing bugs and it was received positively by our developers. Currently using `[[nodiscard]]` is considered a good practice.

In case LEWG is positive about doing such change, I'm willing to do a study of all library functions in the current draft and select candidates for adding the attribute.

¹ [https://twitter.com/podshumok/status/1378399767278026753](https://twitter.com/podshumok/status/1378399767278026753)

₂ Sprinkled with some magic
Changes in wording

https://github.com/cplusplus/draft/pull/4579/commits/ffda5160b821dd64582afe4047f9eb6eef190750

17.2.1 Header `<cstddef>` synopsis

```cpp
template<class IntType> [[nodiscard]] constexpr IntType to_integer(byte b) noexcept;
```

17.2.5 `byte` type operations

```cpp
template<class IntType> [[nodiscard]] constexpr IntType to_integer(byte b) noexcept;
```

20.2.1 Header `<utility>` synopsis

```cpp
template<class T> [[nodiscard]] constexpr T&& forward(remove_reference_t<T>& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr remove_reference_t<T>&& move(T&& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr conditional_t<!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&> move_if_noexcept(T& x) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr add_const_t<T>& as_const(T& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr underlying_type_t<T> to_underlying(T value) noexcept;
```

20.2.4 Forward/move helpers

```cpp
template<class T> [[nodiscard]] constexpr T&& forward(remove_reference_t<T>& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr T&& forward(remove_reference_t<T>&& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr remove_reference_t<T>&& move(T&& t) noexcept;
```

```cpp
template<class T> [[nodiscard]] constexpr conditional_t<!is_nothrow_move_constructible_v<T> && is_copy_constructible_v<T>, const T&, T&&> move_if_noexcept(T& x) noexcept;
```

20.2.5 Function template `as_const`

```cpp
template<class T> [[nodiscard]] constexpr add_const_t<T>& as_const(T& t) noexcept;
```

20.2.8 Function template `to_underlying`

```cpp
template<class T> [[nodiscard]] constexpr underlying_type_t<T> to_underlying(T value) noexcept;
```

20.14.12 Class identity

```cpp
struct identity {
    template<class T> [[nodiscard]] constexpr T&& operator()(T&& t) const noexcept;

    using is_transparent = unspecified;
};
```

26.5.2 Header `<bit>` synopsis

```cpp
template<class To, class From> [[nodiscard]] constexpr To bit_cast(const From& from) noexcept;
```

26.5.3 Function template `bit_cast`

```cpp
template<class To, class From> [[nodiscard]] constexpr To bit_cast(const From& from) noexcept;
```