Project:	ISO JTC1/SC22/WG21: Programming Language C++
Doc No:	WG21 **P0600R0**
Date:	2017-02-06
Reply to:	Nicolai Josuttis (nico@josuttis.de)
Audience:	LEWG, LWG
Prev. Version:	----

[[nodiscard]] in the Library

C++17 introduced the nodiscard attribute.

The question is, where to apply it now in the standard library.

We suggest a conservative approach:

It should be added where:

- For existing API's
  - not using the return value always is a "huge mistake" (e.g. always resulting in resource leak)
  - not using the return value is a source of trouble and easily can happen (not obvious that something is wrong)
- For new API's (not been in the C++ standard yet)
  - not using the return value is usually an error.

It should not be added when:

- For existing API's
  - not using the return value is a possible/common way of programming at least for some input
    - for example for realloc(), which acts like free when the new site is 0
  - not using the return value makes no sense but doesn't hurt.

For example:

Function	**[[nodiscard]] ?**	**Remark**
malloc()	yes	expensive call, usually not using the return value is a resource leak
realloc()	no	realloc() with new size 0 acts like free()
async()	yes	not using the return value makes the call synchronous, which might be hard to detect.
launder()	yes	new API, where not using the return value makes no sense, because launder() does not white-wash. It just the return value allows to use the corresponding data "white washed".
allocate()	yes	same as malloc()
unique_ptr::release()	no	Titus: at Google 3.5% of calls would fail, but analysis showed that it was correct (but weird ownership semantics). See reflector email.
printf(), sprint()	no	too many code not using the return value (which also is not always necessary according to programming logic)
top()	no	not very useful, but no danger and such code might exist

So [[nodiscard]] should not signal bad code if this

a) can be useful
b) is common
c) doesn't hurt

Or as Andrew Tomazos wrote in an email:

1. You almost never want to discard the return value. (Using the return value is almost always an essential part of the interface of the given function.)

and

2. People do sometimes discard the return value of that function by accident. (They do so because they misunderstand the interface of the function, incorrectly thinking it doesn't return a value, or the return value is non-essential extra information.)

As a result, I only see the following modifications for C++17:

add [[nodiscard]] to

- async()
- malloc(), allocate(), operator new

Proposed Wording

(All against N4618)

async():

30.6.1 Overview [futures.overview]

```
template <class F, class... Args>
 [[nodiscard]] future<result_of_t<decay_t<F>(decay_t<Args>...)>>
 async(F&& f, Args&&... args);
template <class F, class... Args>
 [[nodiscard]] future<result_of_t<decay_t<F>(decay_t<Args>...)>>
 async(launch policy, F&& f, Args&&... args);
```

Also in the corresponding definitions in **30.6.8 Function template async [futures.async],** if this is necessary.

allocate():

17.5.3.5 Allocator requirements [allocator.requirements]

§9, in the example:

[[nodiscard]] Tp* allocate(std::size_t n);

20.10.8 Allocator traits [allocator.traits]

static [[nodiscard]] pointer allocate(Alloc& a, size_type n);
static [[nodiscard]] pointer allocate(Alloc& a, size_type n, const_void_pointer hint);

Also in the corresponding definitions in **20.10.8.2 Allocator traits static member functions [allocator.traits.members]** if necessary.

20.10.9 The default allocator [default.allocator]

[[nodiscard]] T* allocate(size_t n);

Also in the corresponding definition in **20.10.9.1 allocator members [allocator.members],** if necessary**.**

20.12.2 Class memory_resource [mem.res.class]

[[nodiscard]] void* allocate(size_t bytes, size_t alignment = max_align);

Also in the corresponding definition in **20.12.2.1 memory_resource public member functions [mem.res.public],** if necessary.

20.12.3 Class template polymorphic_allocator [mem.poly.allocator.class]

[[nodiscard]] Tp* allocate(size_t n);

Also in the corresponding definition in **20.12.3.2 polymorphic_allocator member functions [mem.poly.allocator.mem],** if necessary.

20.13.1 Header <scoped_allocator> synopsis [allocator.adaptor.syn]

[[nodiscard]] pointer allocate(size_type n);

[[nodiscard]] pointer allocate(size_type n, const_void_pointer hint);

Also in the corresponding definition in **20.13.4 Scoped allocator adaptor members [allocator.adaptor.members],** if necessary.

malloc():

18.2.2 Header <cstdlib> synopsis [cstdlib.syn]

[[nodiscard]] void* malloc(size_t size);

Also in the corresponding definition in **20.10.11 C library memory allocation [c.malloc],** if necessary.

new:

3.7.4 Dynamic storage duration [basic.stc.dynamic]

```
[[nodiscard]] void* operator new(std::size_t);
[[nodiscard]] void* operator new(std::size_t, std::align_val_t);
…
[[nodiscard]] void* operator new[](std::size_t);
[[nodiscard]] void* operator new[](std::size_t, std::align_val_t);
```

18.6.1 Header <new> synopsis [new.syn]

```
[[nodiscard]] void* operator new(std::size_t size);
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new(std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new(std::size_t size, std::align_val_t alignment,
                                  const std::nothrow_t&) noexcept;
…
[[nodiscard]] void* operator new[](std::size_t size);
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment);
[[nodiscard]] void* operator new[](std::size_t size, const std::nothrow_t&) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, std::align_val_t alignment,
                                  const std::nothrow_t&) noexcept;
…
[[nodiscard]] void* operator new (std::size_t size, void* ptr) noexcept;
[[nodiscard]] void* operator new[](std::size_t size, void* ptr) noexcept;
```

Also in the corresponding definitions in the subsections of **18.6.2 Storage allocation and deallocation [new.delete]**, if necessary