# std::ignore C++-26, use C++-11

Sven Johannsen
sven@sven-johannsen.de
www.sven-johannsen.de

# #TLTR

Don't ignore the latest C++ standard!

But C++-11 includes a C++-26 feature:

- `std::ignore`.

With C++11 you can use a feature of a future C++ standard

# std::ignore as part of std::tuple

As part of `std::tuple` & std::tie, `std::ignore` can simplifies some use cases:

```
std::tuple<int, float, bool> foo()
{
    return std::make_tuple(42, 3.14f, true);
}

...

int i; bool b;
std::tie(i, std::ignore, b) = foo();
```

`foo()` return a tuple with 3 elements, but the calling code needs only 2 elements. Use `ignore` to ignore the additional element.

# Unspecified type (C++11 standard)

`std::ignore` is very vague specified

- "unspecified" type
- "has no effect"

**20.4.1.2 Header synopsis / [tuple.general]**

```
const unspecified ignore;
```

In the context of `std::tie`:

**(20.4.2.4.7 Tuple creation functions / [tuple.creation])**

> Returns: tuple(t...). When an argument in t is ignore, assigning any value to the corresponding tuple element has no effect.

# Implementation make the difference

The implementation of `std::ignore` allows many more use cases.

GCC: `/usr/include/c++/13/tuple`

```
struct _Swallow_assign
{
  template<class _Tp>
    _GLIBCXX14_CONSTEXPR const _Swallow_assign&
    operator=(const _Tp&) const
    { return *this; }
};
_GLIBCXX17_INLINE constexpr _Swallow_assign ignore{};
```

(Clang & MSVC: similar implementations)

# Use cases outside of `std::tuple`

acknowledge

- unused parameters,
- unused variables and
- ignored return values

# Unused parameters and `nodiscard`

Some coding guides do not allow

- unused parameters (and variables)
- ignore error results

```
enum class [[nodiscard]] Error { Ok, ... };

Error Square::calcArea(int width, int height)
{
    // Warning: unused parameter "height"
    int area = width * width; // warning: unused variable 'area'
    ...
    return Error::Ok;
}

square.calcArea(3, 4); // no error handler, because always Ok
// warning: ignoring return value of function declared with 'nodiscar
```

# Acknowledge / suppress warning (old style)

"Old style" to mark an unused variable with `void` casts

```
enum class [[nodiscard]] Error { Ok, ... };

Error Square::calcArea(int width, int height)
{
    (void)height;
    // height; MSVC
    int area = width * width;
    ...
    return Error::Ok;
}

(void)square.calcArea(4, 4);

auto unused = square.calcArea(3, 3);
(void)unused;
```

# Acknowledge / suppress warning (alternatives)

C style macros or alternative syntax

```
Error Square::calcArea(int width, int height)
{
    Q_UNUSED(height); // QT
    UNUSED(height);   // STM32 SDK
    height;           // MSVC
    ...
}

(void)square.calcArea(4, 4);

auto unused = square.calcArea(3, 3);
UNUSED(unused);
```

# Acknowledge / suppress warning (C++17)

Attribute `[[maybe_unused]]`

```
enum class [[nodiscard]] Error { Ok, ... };

Error Square::calcArea(int width, [[maybe_unused]] int height)
{
    [[maybe_unused]] int area = width * width;

    return Error::Ok;
}

/* [[maybe_unused]] */ square.calcArea(4, 4); // compile error
[[maybe_unused]] auto unused = square.calcArea(3, 3);
```

# Verbale suppressing

Use `std::ignore` to make the suppressing of the warning more verbale

```
#include<tuple>

enum class [[nodiscard]] Error { Ok, ... };

Error Square::calcArea(int width, int height)
{
    std::ignore = height;
    int area = width * width;
    std::ignore = area;       // maybe
    ...
    return Error::Ok;
}

std::ignore = square.calcArea(3, 4);
```

# Make std::ignore a first-class object

C++ proposal: P2968R2 from Peter Sommerlad

Make std::ignore a first-class object

Nice to read and easy to follow C++ proposal.

- Document history
- Non-tuple applications of std::ignore
- Discussions
- Impact on existing code
- References

# C++26 working paper (n4986.pdf)

**(22.4.1.2 General) / [tuple.general]**

> In addition to being available via inclusion of the header, ignore (22.4.2) is available when <utility> (22.2) is included.

**22.4.2 Header synopsis / [tuple.syn]**

```
// ignore
struct ignore-type {
// exposition only
constexpr const ignore-type
operator=(const auto &) const noexcept { return *this; }
};
inline constexpr ignore-type ignore;
```

# P2169 A nice placeholder with no name (also C++26)

see:
https://www.reddit.com/r/cpp/comments/1hx6zke/sandor_dargos_blog_c26_a_placeholder_v

Before

```
std::lock_guard namingIsHard(mutex);
// Structured binding
[[maybe_unused]] auto [x, y, iDontCare] = f();
// Pattern matching
inspect(foo) { __ => bar; };
```

With P2169

```
std::lock_guard _(mutex);
// Structured binding
auto [x, y, _] = f();
// Pattern matching
inspect(foo) { _ => bar; };
```

In contrast to `std::ignore` the placeholder _ are real objects (see `std::lock_guard` example)

# Any questions

a.k.a. end of the presentation