# Concepts of C++ Programming

## Lecture 11: Compile-Time Programming

Alexis Engelke

Chair of Data Science and Engineering (I25)
School of Computation, Information, and Technology
Technical University of Munich

Winter 2024/25

# Attributes[136]

- Almost everything can be annotated with attributes
- C++-style attributes: `[[ <attribute> ]]`
  - Parenthesis inside attributes must be balanced, unknown attributes ignored
- Preprocessor `__has_cpp_attribute(name)` to query support

```
#include <cassert>
int foo();
int foo(int z) {
    // Variable attribute: suppresses warning about unused variable
    int x = 5, y [[maybe_unused]] = foo();
    assert(y); // <-- in release builds, y is unused
    if (z > 10) [[likely]] // Give hint that condition is likely
        x += z * z;
    return x;
}
```

[136] https://en.cppreference.com/w/cpp/language/attributes

# Function Attributes

- [[nodiscard]] – cause warning when function result is unused
  - Beneficial to enforce error handling etc.
- [[deprecated(reason)]] – cause warning when function is used
- [[noreturn]] – indicate that function does not return

```
#include <cassert>
[[nodiscard, deprecated("use xyz instead")]] int oldFunc();
// Second attribute is unknown and ignored, causes warning
[[noreturn, unknown_and_ignored]] void myExit();
int foo(int z) {
    oldFunc();
    myExit();
    // no warning about missing return in non-void function
}
```

# Implementation-Defined Attributes

- Most attributes are implementation-defined
  - E.g., Clang[137] and GCC support hundreds of attributes

Some examples:

- `[[gnu::always_inline]]` – Always inline function when possible
- `[[clang::optnone]]` – Disable optimization for specific function
  - E.g., to ease debugging of a single function
- `[[clang::musttail]]` – Return statement must make tail call
  - Tail call = no stack frame growth for function call, useful for tail-recursive functions

[137] https://clang.llvm.org/docs/AttributeReference.html

# [[clang::lifetimebound]]

- Indicate that return value may refer to parameter object
- Causes warning when parameter's lifetime is shorter than returned value

```
#include <print>
#include <format>
#include <string>
#include <string_view>
struct Error {
    std::string_view msg;
    Error(const std::string &msg [[clang::lifetimebound]]) : msg(msg) {}
};
int main() {
    // Construct error with temporary string...
    Error err(std::format("foo! {}", 123));
    // Warning: dangling reference
    std::println("error: {}", err.msg);
}
```

# [[clang::lifetimebound]] – Example

## Quiz: Which parameter/function should get the attribute?

```
struct Foo {
    std::string msg;
    explicit Foo(const std::string& msg) : msg(msg) {}
    void addMsg(const std::string& add) { msg += add;}
    const std::string& getMsg() const { return msg; }
};
```

A. msg
B. msg and add
C. getMsg (before block, due to this)
D. getMsg, msg, and add

# GNU-Style and MSVC-Style Attributes

- GNU-style: `__attribute__((attrs))`
- MSVC-style: `__declspec(attrs)`

- Much older ($\leadsto$ more widely used) than C++ attributes
- Syntax of attributes sometimes slightly different (see manual)

- Prefer C++-style attributes when possible

# Constant Expressions

- Certain language constructs require compile-time constants
  - E.g., array bounds, non-type template parameters, bit-field length, `static_assert`, enum values, . . .
- So far limited to constant literals or simple expressions

```
static int return4() { return 4; }
int main() {
    const int x1 = 4;
    std::array<int, x1 + 3> arr1; // ok... (due to exception in standard)
    const int x2 = return4();
    std::array<int, x2> arr2; // Error! x2 is not a constant expression
}
```

- const just marks a variable as non-modifiable

# `constexpr`[138]

- `constexpr` variables – can be used as constant expressions
  - Must be initialized immediately with a constant expression
  - Implicitly `const`; some type restrictions (see reference)
- `constexpr` functions – function that is evaluatable at compile-time
  - Result can be used as constant expression
  - Destructor must be `constexpr` (or trivial)

[138] https://en.cppreference.com/w/cpp/language/constexpr

# constexpr – Example

```
int f(int x) { return x * x; }
constexpr int g(int x) { return x * x; }

int main() {
    const int x = 7; // constant expression
    const int y = f(x); // not a constant expression
    const int z = g(x); // constant expression

    constexpr int xx = g(x); // ok
    constexpr int yy = y; // ERROR: f(x) not constant expression
    constexpr int zz = z; // ok
}
```

# constexpr – Example

## Quiz: Which statement is correct?

```
constexpr int* f(int n) { return n ? new int(n) : nullptr; }
int* f1(int n) { return f(n); }
int* f2(int n) { constexpr auto r = f(0); return r; }
int* f3(int n) { constexpr auto r = f(n); return r; }
constexpr int* f4(int n) { return f(n); }
```

A. `f`: heap allocation is performed at compile-time

B. `f2`: `f(0)` is not a constant expression

C. `f3`: `f(n)` is not a constant expression

D. `f4`: must return constant expression

# `constexpr` vs. `consteval` Functions

- `constexpr` functions *can* be evaluated at compile-time
  - Implicitly `inline`
  - When constant expression is required, must yield compile-time constant
  - Other code paths can work with non-compile-time constants
  - Can be called at runtime with dynamic values
- `consteval` functions *must* be evaluated at compile-time
  - Implicitly `inline`
  - *Every* function call must yield compile-time constant
  - Cannot be mixed with `constexpr`

# consteval – Example

```
int sqr(int n) { return n * n; }
consteval int sqrConsteval(int n) { return n * n; }
constexpr int sqrConstexpr(int n) { return n * n; }

int main() {
    constexpr int p1 = sqr(100); // ERROR: not constexpr
    constexpr int p2 = sqrConsteval(100);
    constexpr int p3 = sqrConstexpr(100);
    int x = 100;
    int p4 = sqr(x);
    int p5 = sqrConsteval(x); // ERROR: x not constant expression
    int p6 = sqrConstexpr(x);
    int p7 = sqrConsteval(100); // compile-time
    int p8 = sqrConstexpr(100); // run-time or compile-time
}
```

# `constexpr`/`consteval` and Compile-Time Execution

**Quiz: Which statement is correct?**

A. Non-`constexpr`/`consteval` functions are always evaluated at runtime.
B. When possible, `constexpr` functions are evaluated at compile-time.
C. `consteval` functions can only include compile-time evaluable code.
D. `constexpr` functions can be defined in headers without ODR violations.

# Compile-Time Evaluation – Restrictions

- No undefined behavior (compile-time error)

- Calling functions that are only declared
- Accessing `volatile` variables
- Dynamic memory allocations that are not *delete*-ed in same expression
- Placement `new` (lifted in C++26)
- No `reinterpret_cast`
- Implementation-defined restrictions

- See reference for the full list[139]

[139] https://en.cppreference.com/w/cpp/language/constant_expression

# consteval – Example

## Quiz: What is problematic about this code?

```
#include <vector>
consteval int fib(int n) {
    std::vector<int> i{0, 1};
    while (i.size() <= n)
        i.push_back(i[i.size() - 2] + i.back());
    return i[n];
}
static_assert(fib(6) == 8);
```

A. Cannot use `std::vector` in `consteval` function

B. Cannot use dynamic memory allocation in `consteval` function

C. `fib(6)` $= 13 \neq 8$

D. Nothing, the code compiles without errors

# `if constexpr`[140]

- `if constexpr (...)` – compile-time if
- Not-taken code paths are *discarded*
  - For templates where condition depends on template parameters: not instantiated
- Benefit over `#if`: syntax and large parts of semantics are checked

```
// Example from cppreference
template <typename T> auto getValue(T t) {
    if constexpr (std::is_pointer_v<T>)
        return *t; // deduces return type to int for T = int*
    else
        return t; // deduces return type to int for T = int
}
```

[140] https://en.cppreference.com/w/cpp/language/if#Constexpr_if

# if consteval

- `if consteval { ... }` – execute block if in constant-evaluated context
- `if ! consteval { ... }` – if in not constant-evaluated context
- `std::is_constant_evaluated()` – "legacy" C++20 mechanism

```
constexpr int compute(int x) {
    if consteval {
        // use slower, constant-evaluable algorithm
    } else {
        // use faster algorithm that cannot be executed during compilation
    }
}
```

# constinit[141]

- Variables with static/thread-local storage duration are either
  - … constant-initialized, i.e., take compile-time constants
  - … dynamically initialized, i.e., constructor called at program start-up
- constexpr enforces the former and prevents modifications
- constinit enforces the former, but allows modifications

```
constexpr int square(int n) { return n * n; }
constinit int sq5 = square(5); // mutable variable
```

[141] https://en.cppreference.com/w/cpp/language/constinit

# `decltype`[142]

- Possibly unknown types can often be deduced with `auto`
- Sometimes, knowing the exact type of an expression is useful
  - E.g., for explicitly specifying template parameters

- `decltype(`*identifier/class member access*`)` – yield type of entity
- `decltype(`*expression*`)` – yield type of expression
  - lvalue: `T&`, xvalue: `T&&`, prvalue: `T`
  - Expression is not evaluated
- Note: `decltype(x)` and `decltype((x))` are different!
  - The first yields the type of `x`, the latter a reference as `(x)` is an lvalue

[142] https://en.cppreference.com/w/cpp/language/decltype

# decltype – Examples

```
#include <concepts>

int main() {
    int x;
    const short c = 12;
    static_assert(std::same_as<decltype(x), int>);
    static_assert(!std::same_as<decltype((x)), int>);
    static_assert(std::same_as<decltype((x)), int&>);

    static_assert(std::same_as<decltype(c), const short>);
    static_assert(std::same_as<decltype((c)), const short&>);
    static_assert(std::same_as<decltype(c + c), int>); // integer promotion
}
```

# Interlude: Integer Promotion

- Small integer types get promoted to `int` before arithmetic is performed[143]

## Quiz: Which statement is correct?

Assume `std::same_as<int, int32_t>`.

```
#include <cstdint>
constexpr mul16(uint16_t a, uint16_t b) -> auto { return a * b; }
static_assert(std::same_as<decltype(mul16(1, 1)), int>);
static_assert(mul16(0xffff, 0xffff) == 1);
```

A. The return type of `mul16` is `uint16_t`, deduced from return statement.
B. The return type of `mul16` is `unsigned`, as `uint16_t` is unsigned.
C. The second assertion fails, as it is not a constant expression.
D. The program compiles successfully.

[143] Simplified, but captures the important parts.

# Template Meta-Programming

- Templates are instantiated during compilation
- `if constexpr` makes code actually readable

```
template <unsigned N>
constexpr int templ_fib() {
    if constexpr (N <= 1)
        return N;
    else
        return templ_fib<N-2>() + templ_fib<N-1>();
}
static_assert(templ_fib<6>() == 8);
```

# Template Meta-Programming, the Old Way

- Template specializations used as recursion base case

```
template <unsigned N>
constexpr int templ_fib() {
    return templ_fib<N-2>() + templ_fib<N-1>();
}

template<>
constexpr int templ_fib<0>() { return 0; }

template<>
constexpr int templ_fib<1>() { return 1; }

static_assert(templ_fib<6>() == 8);
```

# Concepts

- Previously seen: type constraints with `requires` clause
  - `template <...> requires` *`bool-constant-expr`*
- Repeating requirements can be tedious

⇝ Concept = named set of requirements

```
template <class T>
concept IntOrFloat = std::integral<T> || std::floating_point<T>;
static_assert(IntOrFloat<int>);
static_assert(!IntOrFloat<int*>);

template <IntOrFloat T> T add(T a, T b) {
    return a + b;
}
```

# Concepts: Requirements

- For templates, the exact type is often hard to verify
- So far: "duck typing" – just assume that method/operator is available
- Concepts allow to verify that all required operations are present

```
// Parameters a,b have no storage, just used as notation for naming requirements
// NB: this is a requires expression, not a requires clause for constraints
template<typename T> concept Addable = requires (T a, T b) {
    // Verify that a + b is a valid expression.
    a + b;
};
template<typename T>
concept Graph = requires {
    // Verify that T has a member type "node_type".
    typename T::node_type;
    // ... and require that is is an integer type.
    requires std::integral<typename T::node_type>;
};
```

# Concepts: Requirements

## Quiz: Which statement is correct?

```
template<typename T>
concept Graph = requires {
    typename T::node_type;
    requires std::integral<typename T::node_type>;
};
class MyGraph {
    using node_type = char;
};
static_assert(Graph<MyGraph>);
```

A. Syntax error: cannot use concept as boolean constant expression.

B. Assertion fails: `char` is not an integer.

C. Assertion fails: `node_type` is private.

D. The program compiles successfully.

# Concepts: Compound Requirements

- We can also check the return type of an expression.

```
// Parameters a,b have no storage, just used as notation for naming requirements
template<typename T>
concept Addable = requires (T a, T b) {
    // Verify that a + b is a valid expression
    // ... and can be implicitly converted to T.
    { a + b } -> std::convertible_to<T>;

    // Alternatively:
    // ... and has the type T.
    { a + b } -> std::same_as<T>;
};
```

# Missing Requirements

- Missing requirements cause candidate to not be selected
- But: this is not an error $\rightsquigarrow$ multiple variants can be provided

```
template <class NodeT>
concept NodeHasNumber = requires(const NodeT& n) {
    { n.getNumber() } -> std::convertible_to<unsigned>;
};
template <class NodeT>
struct NumberedGraph {
    std::unordered_map<const NodeT*, unsigned> nums;
    unsigned getNumber(const NodeT& node) requires NodeHasNumber<NodeT> {
        return node.getNumber();
    }
    unsigned getNumber(const NodeT& node) requires (!NodeHasNumber<NodeT>) {
        auto [it, inserted] = nums.try_emplace(&node, nums.size());
        return it->second;
    }
};
```

# Substitution Failure Is Not An Error (SFINAE)[146]

- If substitution of template parameters fails, the candidate is simply discarded without an error[144]
- Allows to implement `requires` in pre-C++20[145]

```
template <class T>
std::enable_if_t<std::is_integral_v<T>, T> add(T a, T b) {
    return a + b;
}
template <class T>
std::enable_if_t<std::is_floating_point_v<T>, T> add(T a, T b) {
    return a + b;
}
```

- Prefer concepts if possible (i.e., code base uses C++20 or newer)

[144] See reference for details

[145] https://en.cppreference.com/w/cpp/types/enable_if

[146] https://en.cppreference.com/w/cpp/language/sfinae

# Compile-Time Programming – Summary

- Attributes allow for annotation of almost all language constructs
- Most attributes are implementation-specific
- `constexpr` permits use of functions as compile-time constant expressions
- `constexpr` variables must be initialized with compile-time constant
- `consteval` functions must always be evaluated at compile-time
- `constinit` variables must have a constant initializer, but can be mutable
- Concepts can test whether certain expressions are valid
- Failing requirements or substitution failure allows for providing type-dependent implementations (or the absence thereof)

# Compile-Time Programming – Questions

- What happens with unsupported attributes?
- What are use cases for implementation-specific attributes?
- When are `constexpr` function calls evaluated?
- Are non-`constexpr` functions always executed at runtime?
- What is the difference between `constexpr` and `constinit`?
- What is different between `decltype(x)` and `decltype((x))`?
- Which recent C++ constructs largely eliminate the need for template metaprogramming?