if statement

Conditionally executes another statement.

Used where code needs to be executed based on a run-time or compile-time (since C++17) condition, or whether the if statement is evaluated in a manifestly constant-evaluated context (since C++23).

Syntax

attr(optional) **if constexpr**(optional) **(** *init-statement*(optional) *condition* **)** *statement-true*	(1)	
attr(optional) **if constexpr**(optional) **(** *init-statement*(optional) *condition* **)** *statement-true* **else** *statement-false*	(2)	
attr(optional) **if !**(optional) **consteval** *compound-statement*	(3)	(since C++23)
attr(optional) **if !**(optional) **consteval** *compound-statement* **else** *statement*	(4)	(since C++23)

1) *if statement* without an else branch
2) *if statement* with an else branch
3) *consteval if statement* without an else branch
4) *consteval if statement* with an else branch

attr – (since C++11) any number of attributes

constexpr – (since C++17) if present, the statement becomes a *constexpr if statement*

init-statement – (since C++17) either

- an expression statement (which may be a *null statement* ";")
- a simple declaration, typically a declaration of a variable with initializer, but it may declare arbitrary many variables or be a structured binding declaration
- an alias declaration (since C++23)

Note that any *init-statement* must end with a semicolon **;**, which is why it is often described informally as an expression or a declaration followed by a semicolon.

condition – one of

- expression which is contextually convertible to `bool`
- declaration of a single non-array variable with a brace-or-equals initializer.

statement-true – any statement (often a compound statement), which is executed if *condition* evaluates to `true`

statement-false – any statement (often a compound statement), which is executed if *condition* evaluates to `false`

compound-statement – any compound statement, which is executed if the if-statement

- is evaluated in a manifestly constant-evaluated context, if ! is not preceding consteval
- is not evaluated in a manifestly constant-evaluated context, if ! is preceding consteval

statement – any statement (must be a compound statement, see below), which is executed if the if-statement

- is not evaluated in a manifestly constant-evaluated context, if ! is not preceding consteval
- is evaluated in a manifestly constant-evaluated context, if ! is preceding consteval

Explanation

If the *condition* yields `true` after conversion to `bool`, *statement-true* is executed.

If the else part of the if statement is present and *condition* yields `false` after conversion to `bool`, *statement-false* is executed.

In the second form of if statement (the one including else), if *statement-true* is also an if statement then that inner if statement must contain an else part as well (in other words, in nested if-statements, the else is associated with the closest if that doesn't have an else)

Run this code

```
#include <iostream>

int main() {
    // simple if-statement with an else clause
    int i = 2;
    if (i > 2) {
        std::cout << i << " is greater than 2\n";
    } else {
```

```
        std::cout << i << " is not greater than 2\n";
    }

    // nested if-statement
    int j = 1;
    if (i > 1)
        if (j > 2)
            std::cout << i << " > 1 and " << j << " > 2\n";
        else // this else is part of if (j > 2), not of if (i > 1)
            std::cout << i << " > 1 and " << j << " <= 2\n";

   // declarations can be used as conditions with dynamic_cast
   struct Base {
        virtual ~Base() {}
   };
   struct Derived : Base {
       void df() { std::cout << "df()\n"; }
   };
   Base* bp1 = new Base;
   Base* bp2 = new Derived;

   if (Derived* p = dynamic_cast<Derived*>(bp1)) // cast fails, returns nullptr
       p->df();  // not executed

   if (auto p = dynamic_cast<Derived*>(bp2)) // cast succeeds
       p->df();  // executed
}
```

Output:

```
2 is not greater than 2
2 > 1 and 1 <= 2
df()
```

(since C++17)

If statements with initializer

If *init-statement* is used, the if statement is equivalent to

---

{

init_statement
attr(optional) **if constexpr**(optional) **(** *condition* **)**

statement-true

}

---

or

{

init_statement
attr(optional) **if constexpr**(optional) **(** *condition* **)**

statement-true

else

statement-false

}

Except that names declared by the *init-statement* (if *init-statement* is a declaration) and names declared by *condition* (if condition is a declaration) are in the same scope, which is also the scope of both *statement*s.

```
std::map<int, std::string> m;
std::mutex mx;
extern bool shared_flag; // guarded by mx
int demo() {
   if (auto it = m.find(10); it != m.end()) { return it->second.size(); }
   if (char buf[10]; std::fgets(buf, 10, stdin)) { m[0] += buf; }
   if (std::lock_guard lock(mx); shared_flag) { unsafe_ping(); shared_flag = false; }
   if (int s; int count = ReadBytesWithSignal(&s)) { publish(count); raise(s); }
   if (const auto keywords = {"if", "for", "while"};
       std::ranges::any_of(keywords, [&tok](const char* kw) { return tok == kw; })) {
     std::cerr << "Token must not be a keyword\n";
```

```
    }
}
```

(since C++17)

Constexpr if

The statement that begins with **if constexpr** is known as the *constexpr if statement*.

In a constexpr if statement, the value of *condition* must be a contextually converted constant expression of type `bool` (until C++23) an expression contextually converted to `bool`, where the conversion is a constant expression (since C++23). If the value is `true`, then *statement-false* is discarded (if present), otherwise, *statement-true* is discarded.

The return statements in a discarded statement do not participate in function return type deduction:

```
template <typename T>
auto get_value(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
}
```

The discarded statement can odr-use a variable that is not defined

```
extern int x; // no definition of x required
int f() {
if constexpr (true)
    return 0;
else if (x)
    return x;
else
    return -x;
}
```

If a constexpr if statement appears inside a templated entity, and if *condition* is not value-dependent after instantiation, the discarded statement is not instantiated when the enclosing template is instantiated .

```
template<typename T, typename ... Rest>
void g(T&& p, Rest&& ...rs) {
    // ... handle p
    if constexpr (sizeof...(rs) > 0)
        g(rs...); // never instantiated with an empty argument list.
}
```

Outside a template, a discarded statement is fully checked. `if constexpr` is not a substitute for the #if preprocessing directive:

```
void f() {
    if constexpr(false) {
        int i = 0;
        int *p = i; // Error even though in discarded statement
    }
}
```

Note: an example where the condition remains value-dependent after instantiation is a nested template, e.g.

```
template<class T> void g() {
    auto lm = [](auto p) {
        if constexpr (sizeof(T) == 1 && sizeof p == 1) {
           // this condition remains value-dependent after instantiation of g<T>
        }
    };
}
```

This section is incomplete
Reason: the status seems to be changed by P0588R1 (https://wg21.link/P0588R1) .

Note: the discarded statement can't be ill-formed for every possible specialization:

```
template <typename T>
void f() {
     if constexpr (std::is_arithmetic_v<T>)
         // ...
```

```
    else
      static_assert(false, "Must be arithmetic"); // ill-formed: invalid for every T
}
```

The common workaround for such a catch-all statement is a type-dependent expression that is always false:

```
template<class> inline constexpr bool dependent_false_v = false;
template <typename T>
void f() {
    if constexpr (std::is_arithmetic_v<T>)
        // ...
    else
      static_assert(dependent_false_v<T>, "Must be arithmetic"); // ok
}
```

Labels (goto targets, case labels, and default:) appearing in a substatement of a constexpr if can only be referenced (by switch or goto) in the same substatement.

Note: a typedef declaration `or alias declaration (since C++23)` can be used as the init-statement of a constexpr if statement to reduce the scope of the type alias.

This section is incomplete
Reason: no example

(since C++23)

Consteval if

The statement that begins with **if consteval** is known as the *consteval if statement*. In a consteval if statement, both *compound-statement* and *statement* (if any) must be compound statements.

If *statement* is not a compound statement, it will still be treated as a part of the consteval if statement (and thus results in a compilation error):

Run this code

```
constexpr void f(bool b) {
    if (true)
        if consteval { }
        else ; // error: not a compound-statement
               // else not associated with outer if
}
```

If a consteval if statement is evaluated in a manifestly constant-evaluated context, *compound-statement* is executed. Otherwise, *statement* is executed if it is present.

A **case** or **default** label appearing within a consteval if statement shall be associated with a switch statement within the same if statement. A label declared in a substatement of a consteval if statement shall only be referred to by a statement in the same substatement.

If the statement begins with **if !consteval**, the *compound-statement* and *statement* (if any) must be both compound statements. Such statement is not considered as consteval if statement, but is equivalent to a consteval if statement:

- `if !consteval {/*stmt*/}` is equivalent to `if consteval {} else {/*stmt*/}`.
- `if !consteval {/*stmt-1*/} else {/*stmt-2*/}` is equivalent to `if consteval {/*stmt-2*/} else {/*stmt-1*/}`.

compound-statement in a consteval if statement (or *statement* in the negative form) is in an immediate function context, in which a call to an immediate function needs not to be a constant expression.

Run this code

```
#include <cmath>
#include <cstdint>
#include <cstring>
#include <iostream>

constexpr bool is_constant_evaluated() noexcept {
    if consteval { return true; } else { return false; }
}

constexpr bool is_runtime_evaluated() noexcept {
    if not consteval { return true; } else { return false; }
}

consteval std::uint64_t ipow_ct(std::uint64_t base, std::uint8_t exp) {
    if (!base) return base;
    std::uint64_t res{1};
    while (exp) {
        if (exp & 1) res *= base;
```

```
        exp /= 2;
        base *= base;
    }
    return res;
}

constexpr std::uint64_t ipow(std::uint64_t base, std::uint8_t exp) {
    if consteval { // use a compile-time friendly algorithm
        return ipow_ct(base, exp);
    }
    else { // use runtime evaluation
        return std::pow(base, exp);
    }
}

int main(int, const char* argv[]) {
    static_assert(ipow(0,10) == 0 && ipow(2,10) == 1024);
    std::cout << ipow(std::strlen(argv[0]), 3) << '\n';
}
```

Notes

If *statement-true* or *statement-false* is not a compound statement, it is treated as if it were:

```
if (x)
    int i;
// i is no longer in scope
```

is the same as

```
if (x) {
    int i;
} // i is no longer in scope
```

The scope of the name introduced by *condition*, if it is a declaration, is the combined scope of both statements' bodies:

```
if (int x = f()) {
    int x; // error: redeclaration of x
} else {
    int x; // error: redeclaration of x
}
```

If *statement-true* is entered by `goto` or `longjmp`, *statement-false* is not executed.

Built-in conversions are not allowed in the *condition* of a constexpr if statment, except for non-narrowing integral conversions to `bool`. (since C++17) (until C++23)
`switch` and `goto` are not allowed to jump into a branch of constexpr if statement or a consteval if statement (since C++23). (since C++17)

Keywords

`if`, `else`, `constexpr`, `consteval`

Defect reports

The following behavior-changing defect reports were applied retroactively to previously published C++ standards.

DR	Applied to	Behavior as published	Correct behavior
CWG 631 (https://cplusplus.github.io/CWG/issues/631.html)	C++98	the control flow was unspecified if first substatement is reached via a label	same as in C

See also

is_constant_evaluated (C++20)	detects whether the call occurs within a constant-evaluated context (function)
C documentation for **if statement**	