Chapter 1

Safe Features

ch-safe
sec-safe-cpp11 Intro text should be here.

Chapter 1 Safe Features

sec-safe-cpp14

Chapter 2

Conditionally Safe Features

ch-conditional

sec-conditional-cpp11 Intro text should be here.

Variables of Automatically Deduced Type

auto variable

The **auto** keyword was repurposed[1] in C++11 to act as a **placeholder type**. When used in place of a type as part of a variable declaration, the compiler will deduce the variable type from its initializer.

Description

description

The **auto** keyword may be used instead of an explicit type's name when declaring variables. In such cases, the variable's type is deduced from its initializer by the compiler applying the placeholder type deduction rules, which, apart from a single exception for list initializers (see *Potential Pitfalls — Surprising deduction for list initialization* on page 15), are the same as the rules for **function template argument type deduction**:

```
auto two = 2;     // Type of two is deduced to be int
auto pi = 3.14f;  // Type of pi is deduced to be float.
```

The types of the `two` and `pi` variables above are deduced in the same manner as they would be if the same initializer were passed to a function template taking a single argument of the template type by value:

```
template <typename T> void deducer(T);

void testDeduction()
{
    deducer(2);      // T is deduced to be int.
    deducer(3.14f);  // T is deduced to be float.
}
```

If the variable declared with **auto** does not have an initializer, if its name appears in the expression used to initialize it, or if the initializers of multiple variables in the same declaration don't deduce the same type, the program is ill formed:

```
auto x;               // Error, declaration of auto x has no initializer.
auto n = sizeof(n);   // Error, use of n before deduction of auto
auto i = 3, f = 0.3f; // Error, inconsistent deduction for auto
```

Just like the function template argument deduction never deduces a reference type for its by-value argument, a variable declared with unqualified **auto** is never deduced to have a reference type:

```
int  val = 3;
int& ref = val;
auto tmp = ref;  // Type of tmp is deduced to be int, not int&.
```

Augmenting **auto** with reference and cv-qualifiers, however, allows controlling whether the deduced type is a reference and whether it is **const** and/or **volatile**:

[1] Prior to C++11, the **auto** keyword could be used as a **storage class specifier** for objects declared at block scope and in function parameter lists to indicate automatic storage duration, which is the default for these kinds of declarations. The **auto** keyword was repurposed in C++11 alongside the deprecation (and subsequent removal in C++17) of the **register** keyword as a storage class specifier.

```
auto val = 3;
    // Type of val is deduced to be int.
    // same as argument for template <typename T> void deducer(T)

const auto cval = val;
    // Type of cval is deduced to be const int.
    // same as argument for template <typename T> void deducer(const T)

auto& ref = val;
    // Type of ref is deduced to be int&.
    // same as argument for template <typename T> void deducer(T&)

auto& cref1 = cval;
    // Type of cref1 is deduced to be const int&.
    // same as argument for template <typename T> void deducer(T&)

const auto& cref2 = val;
    // Type of cref2 is deduced to be const int&.
    // same as argument for template <typename T> void deducer(const T&)
```

Note that qualifying **auto** with && does *not* result in deduction of an *rvalue* reference (see Section 2.1."**??**" on page **??**), but, in line with function template argument deduction rules, would be treated as a *forwarding reference* (see Section 2.1."**??**" on page **??**). This means that a variable declared with **auto**&& will result in an *lvalue* or an *rvalue* reference depending on the value category of its initializer:

```
double doStuff();

      int val  = 3;
const int cval = 7;

// Deduction rules are the same as for template <typename T> void deducer(T&&):

auto&& lref1 = val;
    // Type of lref1 is deduced to be int&.

auto&& lref2 = cval;
    // Type of lref2 is deduced to be const int&.

auto&& rref = doStuff();
    // Type of rref is deduced to be double&&.
```

Similarly to references, explicitly specifying that a pointer type is to be deduced is possible. If the supplied initializer is not of a pointer type, the compiler will issue an error:

```
const auto* cptr = &cval;
    // Type of cptr is deduced to be const int*.
    // same as argument for template <typename T> void deducer(const T*)

auto* cptr2 = cval;  // Error, cannot deduce auto* from cval
```

The compiler can also be instructed to deduce pointers to functions, data members, and member functions (but see *Annoyances — Not all template argument deduction constructs are allowed for* **auto** on page 16):

```
float freeF(float);

struct S
{
    double d_data;
    int memberF(long);
};

auto (*fptr)(float) = &freeF;
  // Type of fptr is deduced to be float (*)(float).
  // same as argument for template <typename T> void deducer(T (*)(float))

const auto S::* mptr = &S::d_data;
  // Type of mptr is deduced to be const double S::*.
  // same as argument for template <typename T> void deducer(T S::*)

auto (S::* mfptr)(long) = &S::memberF;
  // Type of mfptr is deduced to be int (S::*)(long).
  // same as argument for template <typename T> void deducer(T (S::*)(long))
```

Unlike references, pointer types may be deduced by **auto** alone. Therefore, different forms of **auto** can be used to declare a variable of a pointer type:

```
auto  cptr1 = &cval;  // const int*
auto *cptr2 = &cval;  //   "     "

auto   fptr1         = &freeF;  // float (*)(float)
auto  *fptr2         = &freeF;  //    "         "
auto (*fptr3)(float) = &freeF;  //    "         "

auto      mptr1 = &S::d_data;  // double S::*
auto S::* mptr2 = &S::d_data;  //    "     "

auto       mfptr1        = &S::memberF;  // int (S::*)(long)
auto (S::* mfptr2)(long) = &S::memberF;  //   "    "     "
```

Note, however, that because regular and member pointers are incompatible, **auto*** cannot be used to deduce pointers to data members and member functions:

```
auto *mptr3  = &S::d_data;   // Error, cannot deduce auto* from &S::d_data
auto *mfptr3 = &S::memberF;  // Error, cannot deduce auto* from &S::memberF
```

In addition, storage class specifiers as well as the **constexpr** (see Section 2.1."**constexpr** Variables" on page 19) specifier can be applied to variables that use **auto** in their declaration:

```
thread_local     const auto* logPrefix = "mylib";
static constexpr       auto  pi        = 3.1415926535f;
```

Finally, **auto** variables may be declared in any location that allows declaring a variable supplied with an initializer with a single exception of nonstatic data members (see *Annoyances — **auto** not allowed for nonstatic data members* on page 16):

```
    std::vector
// namespace scope
auto globalNamespaceVar = 3.;

namespace ns
{
    static auto nsNamespaceVar = "...";
}

void f()
{
    // block scope
    constexpr auto blockVar = 'a';

    // condition of if, switch, and while statements
    if     (auto rc        = sendRequest())    { /* ... */ }
    switch (auto status    = responseStatus()) { /* ... */ }
    while  (auto keepGoing = haveMoreWork())   { /* ... */ }

    // init-statement of for loops
    for (auto it = vec.begin(); it != vec.end(); ++it) { /* ... */ }

    // range declaration of range-based for loops
    for (const auto& constVal : vec) { /* ... */ }
}

struct S
{
    // static data members
    static const auto k_CONSTANT = 11u;
};
```

use-cases-auto

Use Cases

variable-initialization

Ensuring variable initialization

Consider a defect introduced due to mistakenly leaving a variable uninitialized:

```
void testUninitializedInt()
{
    int recordCount;
    while (cursor.next()) { ++recordCount; }  // Bug, undefined behavior
}
```

Variables declared with **auto** must be initialized. Use of **auto** might, therefore, prevent such defects:

```
void testUnitializedAuto()
{
    auto recordCount; // Error, declaration of recordCount has no initializer
    while (cursor.next()) { ++recordCount; }
}
```

In addition, the initialization requirement might encourage a good practice of reducing the scope of local variables.

-type-name-repetition

Avoiding redundant type name repetition

Certain function templates require that the caller explicitly specify the type that the function uses as its return type. For example, the `std::make_shared<TYPE>` function returns a `std::shared_ptr<TYPE>`.[2] If a variable's type is specified explicitly, such declarations redundantly repeat the type. The use of **auto** obviates this repetition:

```
    std::shared_ptrstd::unique_ptr

// Without auto:
std::shared_ptr<RequestContext> context1 = std::make_shared<RequestContext>();
std::unique_ptr<Socket>         socket1  = std::make_unique<Socket>();

// With auto:
auto context2 = std::make_shared<RequestContext>();
auto socket2  = std::make_unique<Socket>();
```

-implicit-conversions

Preventing unexpected implicit conversions

Use of **auto** might prevent defects arising from explicitly specifying a variable's type that is distinct — yet implicitly convertible — from its initializer. As an example, the code below has a subtle defect that can lead to performance degradation or incorrect semantics:

```
    std::mapstd::pair

void testManualForLoop()
{
    std::map<int, User> users{/* ... */};
    for (const std::pair<int, User>& idUserPair : users)
    {
```

[2] Often, such functions are associated with `optional` and `variant` types, which were standardized in C++17:

```
   std::stringstd::variantstd::optional

std::variant<std::string, int, double> valueVariant;

// Without auto:
std::optional<std::string> greeting1 = std::make_optional<std::string>("Hello");
const std::string& valueString1 = std::get<std::string>(valueVariant);

// With auto:
auto greeting2 = std::make_optional<std::string>("Hello");
const auto& valueString2 = std::get<std::string>(valueVariant);
```

```
        // ...
    }
}
```

On every iteration, the `idUserPair` will be bound to a *copy* of the corresponding pair in the `users` map. This happens because the type returned by dereferencing the `map`'s iterator is `std::pair<const int, User>`, which is implicitly convertible to `std::pair<int, User>`. Using **`auto`** would allow the compiler to deduce the correct type and avoid this unnecessary and potentially expensive copy:

```
void testAutoForLoop()
{
    std::map<int, User> users{/* ... */};
    for (const auto& idUserPair : users)
    {
        // auto is deduced as std::pair<const int, User>.
    }
}
```

Declaring variables of implementation-defined or compiler-synthesized types

piler-synthesized-types

Using **`auto`** is the only way to declare variables of implementation-defined or compiler-synthesized types, such as lambda expressions (see Section 2.1."**??**" on page **??**). While in some cases using type erasure to avoid the need to spell out the type is possible, doing so typically comes with additional overhead. For example, storing a lambda closure in a `std::function` might entail an allocation on construction and virtual dispatch upon every call:

std::function

```
void testCallbacks()
{
    std::function<void()> errorCallback0 = [&]{ saveCurrentWork(); };
        // OK, implicit conversion from anonymous closure type to
        // std::function<void()>, which incurs additional overhead

    auto errorCallback1 = [&]{ saveCurrentWork(); };
        // Better, deduces the compiler-synthesized type
}
```

Declaring variables of complex and deeply nested types

and-deeply-nested-types

`auto` can be used to declare variables of types that are complex or do not convey useful information. A typical example is avoiding the need for spelling out the iterator type of a container:

std::vector

```
void doWork(const std::vector<int>& data)
{
    // without auto:
```

```
    for (std::vector<int>::const_iterator it = data.begin();
         it != data.end();
         ++it) {
        /* ... */
    }

    // with auto:
    for (auto it = data.begin(); it != data.end(); ++it) { /* ... */ }
}
```

Furthermore, the need for such types can arise, for example, when storing intermediate results of **expression templates** whose types can be deeply nested and unreadable and might even differ between versions of the same library:

```
// without auto:
TransformRange<
    FilterRange<decltype(employees), JoinedInYear>,
    &Employee::name
> newEmployeeNames1 =
    employees | filter(JoinedInYear(2019))
              | transform(&Employee::name);

// with auto:
auto newEmployeeNames2 =
    employees | filter(JoinedInYear(2019))
              | transform(&Employee::name);
```

Improving resilience to library code changes

-library-code-changes

auto may be used to indicate that code using the variable doesn't rely on a specific type but rather on certain requirements that the type must satisfy. Such an approach might give library implementers more freedom to change return types without affecting the semantics of their clients' code in projects where automated large-scale refactoring tools are not available (but see *Potential Pitfalls — Lack of interface restrictions* on page 13). As an example, consider the following library function:

```
    std::vector

std::vector<Node> getNetworkNodes();
    // Return a sequence of nodes in the current network.
```

As long as the return value of the `getNetworkNodes` function is only used for iteration, that a `std::vector` is returned is not pertinent. If clients use **auto** to initialize variables storing the return value of this function, the implementers of `getNetworkNodes` can migrate from `std::vector` to, for example, `std::deque` without breaking their clients' code.

```
// without auto:
void testConcreteContainer()
{
    const std::vector<Node> nodes = getNetworkNodes();
    for (const Node& node : nodes) { /* ... */ }
```

```
        // prevents migration
}

void testDeducedContainer()
{
    // with auto:
    const auto nodes = getNetworkNodes();
    for (const Node& node : nodes) { /* ... */ }
        // The return type of getNetworkNodes can be silently
        // changed while retaining correctness of the user code.
}
```

potential-pitfalls-auto

Potential Pitfalls

compromised-readability

Compromised readability

Use of **auto** hides essential semantic information contained in a variable's type, increasing cognitive load for readers. In conjunction with unclear variable naming, disproportionate use of **auto** can make code difficult to read and maintain.

std::vectorstd::string

```
int main(int argc, char** argv)
{
    const auto args0 = parseArgs(argc, argv);
        // The behavior of parseArgs and operations available on args0 is unclear.

    const std::vector<std::string> args1 = parseArgs(argc, argv);
        // It is clear what parseArgs does and what can be done with args1.
}
```

While reading the contract of the `parseArgs` function at least once may be necessary to fully understand its behavior, using an explicit type's name at the call site helps readers understand its purpose.

unintentional-copies

Unintentional copies

Since the rules for function template type deduction apply to **auto**, appropriate cv-qualifiers and declarator modifiers (`&`, `&&`, `*`, etc.) must be applied to avoid unnecessary copies that might negatively affect both code performance and correctness. For example, consider a function that capitalizes a user's name:

std::stringstd::toupper

```
void capitalizeName(User& user)
{
    if (user.name().empty())
    {
        return;
    }

    user.name()[0] = std::toupper(user.name()[0]);
}
```

This function was then incorrectly refactored to avoid repetition of the `user.name()` invocation. However, a missing reference qualification leads not only to an unnecessary copy of the string, but also to the function failing to perform its job:

```
void capitalizeName(User& user)
{
    auto name = user.name();  // Bug, unintended copy

    if (name.empty())
    {
        return;
    }

    name[0] = std::toupper(name[0]);  // Bug, changes the copy
}
```

Furthermore, even a fully cv-ref-qualified **auto** might still prove inadequate in cases as simple as introducing a variable for a returned-temporary value. As an example, consider refactoring the contents of this simple function:

```
void testExpression()
{
    useValue(getValue());
}
```

For debugging or readiablity, it can help to use an intermediate variable to store the results of `getValue()`

```
void testRefactoredExpression()
{
    auto&& tempValue = getValue();
    useValue(tempValue);
}
```

The above invocation of `useValue` is not equivalent to the original expression; the semantics of the program might have changed because `tempValue` is an *lvalue* expression. To get close to the original semantics, `std::forward` and **decltype** must be used to propagate the original value category of `getValue()` to the invocation of `useValue` (see Section 2.1."**??**" on page **??**):

```
#include <utility>  // std::forward
void testBetterRefactoredExpression()
{
    auto&& tempValue = getValue();
    useValue(std::forward<decltype(tempValue)>(tempValue));
}
```

Note that, even with the latest changes, the code above achieves the same result but in a somewhat different way because `std::forward<decltype(tempValue)>(tempValue)` is an *xvalue* expression whereas `getValue()` is a *prvalue* expression; see Section 2.1."**??**" on page **??**.

-lack-of-expected-ones)

Unexpected conversions (and lack of expected ones)

Compulsively declaring variables using **auto**, even in cases where the desired type has to be spelled out in the initializer, allows explicit conversions to be used where they would not be applicable otherwise. For an example of unintentional consequences of allowing such conversions, consider a function template that is intended to combine two `chrono` duration values:

```
#include <chrono>  // std::chrono::seconds
template <typename Duration1, typename Duration2>
std::chrono::seconds combine_durations(Duration1 d1, Duration2 d2)
{
    auto d = std::chrono::seconds{d1 + d2};
    // ...
}
```

This function template will be successfully instantiated and compiled even when its arguments are two **int**s. Were **auto** not used in this situation — i.e., were `d` declared as `std::chrono::seconds d = d1 + d2;` — the code would fail to compile because the conversion from **int** to `seconds` is explicit, which would indicate a likely defect in the caller's code.

Conversely, some conversions that would be expected to happen might be missed when using **auto** instead of an explicitly specified type. For example, **auto** might deduce a proxy type that might lead to difficult-to-diagnose defects:

```
    std::vector
```

```
void testProxyDeduction()
{
    std::vector<bool> flags = loadFlags();

    auto firstFlag = flags[0];  // deduces a proxy type, not bool
    flags.clear();

    if (firstFlag) // Bug, use-after-free: flags vector released its memory.
    {
        // ...
    }
}
```

-interface-restrictions

Lack of interface restrictions

Lack of any restrictions placed by **auto** on the type that is deduced might result in defects that could otherwise be detected at compile time. Consider refactoring the `getNetworkNodes` function illustrated in *Use Cases — Improving resilience to library code changes* on page 10 to return `std::deque<Node>` instead of `std::vector<Node>`:

```
    std::deque
```

```
std::deque<Node> getNetworkNodes();  // Return type changed from std::vector<Node>.
    // Return a sequence of nodes in the current network.
```

While code that uses **auto** to store the result returned by `getNetworkNodes` only to subsequently iterate over it with a range-based **for** wouldn't be affected, the behavior of code that relied on the contiguous layout of elements in `std::vector` objects *silently* becomes undefined:

```
void testUseContiguousMemory()
{
    auto nodes = getNetworkNodes();
    CLibraryProcessNodes(&nodes[0], nodes.size());
        // exhibits UB after std::vector-to-std::deque change
}
```

While specifying constraints on types deduced by **auto** with **static_assert** is possible, doing so is often cumbersome[3]:

```
const Packet* PacketCache::findFirstCorruptPacket() const
{
    auto it = std::begin(this->d_packets);

    static_assert(
        std::is_base_of<
            std::random_access_iterator_tag,
            std::iterator_traits<decltype(it)>::iterator_category>::value,
        "it must satisfy the requirements of a random access iterator.");

    /* ... */

    return it == std::end(this->d_packets) ? nullptr : &*it;
}
```

Obscuration of important properties of fundamental types

-of-fundamental-types

Use of **auto** for variables of fundamental types might hide important, context-sensitive considerations, such as overflow behavior or a mix of signed and unsigned arithmetic. In the example below, the `lowercaseEncode` function will either work correctly or enter an infinite loop depending on whether the type returned by `Encoder::encodedLengthFor` is signed.

```
    std::stringstd::tolower
void lowercaseEncode(std::string* result, const std::string& input)
{
    auto encodedLength = Encoder::encodedLengthFor(input);

    result->resize(encodedLength);
    Encoder::encode(result->begin(), input);

    while (--encodedLength >= 0)  // infinite loop if encodedLength is unsigned
    {
```

[3] C++20 introduced **concepts** — named type requirements — as well as means to constrain **auto** with a specific concept, which can be used instead of **static_assert** in such circumstances.

```
            (*result)[encodedLength] = std::tolower((*result)[encodedLength]);
        }
    }
```

for-list-initialization

Surprising deduction for list initialization

auto type-deduction rules differ from those of function templates if brace-enclosed initializer list are used. Function template argument deduction will always fail, whereas, according to C++11 rules, `std::initializer_list` will be deduced for **auto**.

```
auto example0 = 0; // copy initialization, deduced as int
auto example1(0);  // direct initialization, deduced as int
auto example2{0};  // list initialization, deduced as std::initializer_list<int>

template <typename T> void func(T);
void testFunctionDeduction()
{
    func(0);   // T deduced as int
    func({0}); // Error
}
```

This surprising behavior was, however, widely regarded as a mistake and was formally rectified in C++17 with, e.g., **auto** `i{0}` deducing **int**. Furthermore, mainstream compilers had applied this deduction-rule change retroactively as early as GCC 5.1 and Clang 3.8, with the revised rule being applied even if `-std=c++11` flag is explicitly supplied.

Nonetheless, even with this retroactive fix, the effects of the deduction rules when applied to braced initializer lists might be puzzling. In particular, `std::initializer_list` is deduced when **copy initialization** is used instead of **direct initialization**, and this requires including `<initializer_list>`:

```
auto x1 = 1;    // int
auto x2(1);     //   "
auto x3{1};     //   "

#include <initializer_list>  // std::initializer_list
auto x4 = {1};  // std::initializer_list<int>

auto x5{1, 2};     // Error, direct-list-init requires exactly 1 element.
auto x6 = {1, 2};  // std::initializer_list<int>
```

n-arrays-is-problematic

Deducing built-in arrays is problematic

Deducing built-in array types using **auto** presents multiple challenges. First, declaring an array of **auto** is ill formed:

```
auto arr1[]  = {1, 2};  // Error, array of auto is not allowed.
auto arr2[2] = {1, 2};  // Error, array of auto is not allowed.
```

Second, if the array bound is not specified, either the program does not compile or `std::initializer_list` is deduced instead of a built-in array:

```
#include <initializer_list>  // std::initializer_list
auto arr3 = {1, 2};  // std::initializer_list<int>
auto arr4{1, 2};     // Error, direct-list-init requires exactly 1 element.
```

Finally, attempting to circumvent this deficiency by using an alias template (see Section 1.1."**??**" on page **??**) will result in code that compiles but has undefined behavior:

```
    std::size_t

template <typename TYPE, std::size_t SIZE>
using BuiltInArray = TYPE[SIZE];

auto arr5 = BuiltInArray<int, 2>{1, 2};
    // Error, taking the address of a temporary array
```

Note that in this case such code also almost entirely defeats the purpose of **auto** since neither the array element's type nor the array's bound are deduced.

With that said, using **auto** to deduce references to built-in arrays is straightforward:

```
int data[] = {1, 2};

      auto&  arr6 = data;                          //       int (&) [2]
const auto&  arr7 = BuiltInArray<int, 2>{1, 2};  // const int (&) [2]
      auto&& arr8 = BuiltInArray<int, 2>{1, 2};  //       int (&&)[2]
```

Note that the `arr7` and `arr8` references in the code snippet immediately above extend the lifetime of the temporary arrays that they bind to, so subscripting them does not have the undefined behavior that subscripting `arr5` (in the previous code snippet) has.

Annoyances

annoyances-auto

`auto` not allowed for nonstatic data members

n-static-data-members

Despite C++11 allowing nonstatic data members to be initialized within class definitions, **auto** cannot be used to declare them:

```
class C
{
    auto d_i = 1;  // Error, nonstatic data member is declared with auto.
};
```

Not all template argument deduction constructs are allowed for `auto`

-are-allowed-for-auto

Despite **auto** type deduction largely following the template argument deduction rules, certain constructs that are allowed for templates are not allowed for **auto**. For example, when deducing a pointer-to-data-member type, templates allow for deducing both the data member type and the class type, whereas **auto** can deduce only the former:

```
struct Node
{
    int   d_data;
    Node* d_next;
};
```

```
template <typename TYPE>
void deduceMemberTypeFn(TYPE Node::*);

void testDeduceMemberType()
{
                 deduceMemberTypeFn   (&Node::d_data);  // OK, int Node::*
    auto Node::* deduceMemberTypeVar = &Node::d_data;   // OK,   "     "
}

template <typename TYPE>
void deduceClassTypeFn(int TYPE::*);

void test1DeduceClassType()
{
                deduceClassTypeFn   (&Node::d_data);  // OK, int Node::*
    int auto::* deduceClassTypeVar = &Node::d_data;   // Error, not allowed
}

template <typename TYPE>
void deduceBothTypesFn(TYPE* TYPE::*);

void testDeduceBothTypes()
{
                  deduceBothTypesFn   (&Node::d_next);  // OK, Node* Node::*
    auto* auto::* deduceBothTypesVar = &Node::d_next;   // Error, not allowed
}
```

Furthermore, deducing the parameter of a class template is also not allowed:

```
    std::vector

std::vector<int> vectorOfInt;

template <typename TYPE>
void deduceVectorArgFn(const std::vector<TYPE>&);

void testDeduceVectorArg()
{
                      deduceVectorArgFn   (vectorOfInt); // OK, TYPE is int
    std::vector<auto> deduceVectorArgVar = vectorOfInt;  // Error, not allowed
}
```

Instead, if **auto** type deduction is desired in such cases, **auto** alone is suitable to deduce the type from the initializer:

```
auto deduceClassTypeVar = &Node::d_data;  // OK, int Node::*
auto deduceBothTypesVar = &Node::d_next;  // OK, Node* Node::*

auto deduceVectorArgVar = vectorOfInt;    // OK, std::vector<int>
```

see-also

See Also

- "??" (§1.1, p. ??) ♦ the **auto** placeholder can be used to specify a function's return type at the end of its signature.
- "??" (§2.2, p. ??) ♦ the **auto** placeholder can be used in the argument list of a lambda to make its function call operator a template.
- "??" (§3.2, p. ??) ♦ the **auto** placeholder can be used to deduce a function's return type.

Further Reading

TODO

Compile-Time Accessible Variables

constexprvar

A variable or **variable template** of **literal type** may be declared to be **constexpr**, ensuring its successful construction and enabling its use *at compile-time*.

Description

description

Variables of *all* built-in types and certain user-defined types, collectively known as literal types, may be be declared **constexpr**, allowing them to be initialized *at compile-time* and subsequently used in **constant expressions**:

std::array

```
       int i0 = 5;
 const int i1 = 5;              // i1 is a compile-time constant.
constexpr int i2 = 5;           // i2  " "    "      "     "

 const double d1 = 5.0;
constexpr double d2 = 5.0;      // d2 is a compile-time constant.

       const char *s1 = "help";
constexpr const char *s2 = "help";  // s2 is a compile-time constant.
```

Although **const** integers initialized in the view of the compiler can be used within constant expressions — e.g., as the first argument to **static_assert**, the size of an array, or as a non-type template parameter — such is not the case for any other types:

```
static_assert(i0 == 5, "");         // Error, i0 is not a compile-time constant.
static_assert(i1 == 5, "");         // OK, const is "magical" for integers (only).
static_assert(i2 == 5, "");         // OK

static_assert(d1 == 5, "");         // Error, d1 is not a compile-time constant.
static_assert(d2 == 5, "");         // OK

static_assert(s1[1] == 'e', "");    // Error, s1 is not a compile-time constant.
static_assert(s2[1] == 'e', "");    // OK, the ASCII code for e is decimal 101.

int a[s2[1]];                       // OK, defines an array, a, of 101 integers
static_assert(sizeof a == 404, ""); // OK, we are assuming the size of int is 4.

std::array<int, s1[1]> f;           // Error, s1 is not constexpr.
std::array<int, s2[1]> e;           // OK, defines a std::array, e, of 101 integers
```

Prior to C++11, the types of variables usable in a constant expression were quite limited:

```
const int b;         // Error, const scalar variable must be initialized.
extern const int c;  // OK, declaration
const int d = c;     // OK, not constant initialized (c initializer not seen)

int ca1[c];          // Error, c initializer not visible
int da1[d];          // Error, d initializer not constant
```

```
const int c = 7;
int ca2[c];          // OK, initializer is visible
int da2[d];          // Error, d initializer (still) not constant

const int e = 17;    // OK
int ea[e];           // OK
```

For an **integral constant** to be usable at compile time (e.g, as part of constant expression), certain requirements must be satisfied:

1. The variable must be marked **const**.
2. The initializer for a variable must have been seen by the time it is used, and it must be a constant expression — this is the information needed for a compiler to be able to make use of the variable in other constant expressions.
3. The variable must be of *integral* type, e.g., **bool**, **char**, **short**, **int**, **long**, **long long**, as well as the **unsigned** variations on these and any additional **char** types; see also Section 1.1."**??**" on page **??**.

This restriction to integral types provides support for those values where compile-time constants are most frequently needed while limiting the complexity of what compilers were required to support at compile time.

Use of **constexpr** when declaring a variable (or variable template; see Section 1.2."**??**" on page **??**) enables a much richer category of types to participate in constant expressions. This generalization, however, was not made for mere **const** variables because they are not *required* to be initialized by compile-time constants:

```
int f() { return 0; }  // f() is not a compile-time constant expression.

               int x0 = f();  // OK
         const int x1 = f();  // OK, but x1 is not a compile-time constant.
constexpr      int x2 = f();  // Error, f() "    "   "     "     "     "
constexpr const int x3 = f();  // Error, f() "    "   "     "     "     "
```

As the example code above demonstrates, variables marked **constexpr** *must* satisfy the same requirements needed for integral constants to be usable as constant expression. Unlike other integral constants, their initializers *must* be constant expressions or else the program is **ill formed**.

For a variable of other than **const** integral type to be usable in a constant expression, certain criteria must hold:

1. The variable must be annotated with **constexpr**, which implicitly also declares the variable to be **const**[1]:

```
struct S  // simple (aggregate) literal type
```

[1] C++20 added the **constinit** keyword to identify a variable that is initialized at compile time (with a constant expression) but may then be modified at runtime.

```
{
    int i;  // built-in integer data member
};

constexpr S s{1};  // OK, literal type initialized via constant expression

s = S();           // Error, s is implicitly declared const via constexpr.

static_assert(s.i == 1);  // OK, subobjects of constexpr objects are constexpr.

constexpr int j = s.i;    // OK, subobjects are usable in constant expressions.
```

In the example above, we have, for expedience of exposition, used brace initialization to initialize the aggregate; see Section 2.1."??" on page ??. Note that subobjects of **constexpr** objects are also effectively **constexpr** and can be used freely in constant expressions even though they themselves might not be explicitly declared **constexpr**.

2. All **constexpr** variables must be initialized with a constant expression when they are defined. Hence, every **constexpr** variable has an initializer, and that initializer must be a valid constant expression (see Section 2.1."??" on page ??):

```
          int g() { return 17; }  // a non-constexpr function
constexpr int h() { return 34; }  // a constexpr function

constexpr int v1;        // Error, v1 is not initialized.
constexpr int v2 = 17;   // OK
constexpr int v3 = g();  // Error, g() is not constexpr.
constexpr int v4 = h();  // OK

void func(int c)
{
    constexpr int v5 = c;          // Error, c not a compile-time constant.
    constexpr int v6 = sizeof(c);  // OK, c is not evaluated.
}
```

3. Any variable declared **constexpr** must be of literal type; all literal types are, among other things, **trivially destructible**:

```
struct Lt  // literal type
{
    constexpr  Lt()  { }  // constexpr constructor
    ~Lt() = default;      // default trivial destructor
};

constexpr Lt lt;  // OK, Lt is a literal type.

struct Nlt  // nonliteral type.
{
    Nlt()  { }  // cannot initialize at compile-time
    ~Nlt() { }  // cannot skip non-trivial destruction
```

```
};

constexpr Nlt nlt;  // Error, Nlt is not a literal type.
```

Since all literal types are trivially destructible, the compiler does not need to emit any special code to manage the end of the lifetime of a **constexpr** variable, which can essentially live "forever" — i.e., until the program exits.[2]

4. Unlike integral constants, non**static** data members cannot be **constexpr**. Only variables at global or **namespace** scope, automatic variables, or **static** data members of a **class** or **struct** may be declared **constexpr**. Consequently, any given **constexpr** variable is a top-level object, never a subobject of another (possible non**constexpr**) object:

```
                constexpr int i = 17;    // OK, file scope
namespace ns { constexpr int j = 34; }  // OK, namespace scope

struct C
{
    static constexpr int k = 51;  // OK, static data member
           constexpr int l = 68;  // Error, constexpr nonstatic data member
};

void g()
{
    static constexpr int m = 85;  // OK
           constexpr int n = 92;  // OK
};
```

Recall, however, that **nonstatic data members** of **constexpr** objects are implicitly **constexpr** and therefore can be used directly in any constant expressions:

```
constexpr struct D { int i; } x{1};  // brace-initialized aggregate x
constexpr int k = x.i;  // Subobjects of constexpr objects are constexpr.
```

Initializer Undefined Behavior

It is important to note the significance of one of the differences between a **constexpr** integral variable and a **const** integral variable. Because the initializer of a **constexpr** variable is *required* to be a constant expression, it is not subject to the possibility of undefined behavior (e.g., integer overflow, out-of-bounds array access) at run time and will instead result in a compile-time error:

```
const int iA = 1 << 15;  // 2^15 = 32,768 fits in 2 bytes.
const int jA = iA * iA;  // OK
```

[2] In C++20, literal types can have non-trivial destructors, and the destructors for **constexpr** variables will be invoked under the same conditions that a destructor would be invoked for a non**constexpr** global or **static** variable.

```
          const int iB = 1 << 16;  // 2^16 = 65,536 doesn't fit in 2 bytes.
          const int jB = iB * iB;  // Bug, overflow (might warn)

constexpr const int iC = 1 << 16;
constexpr const int jC = iC * iC;  // Error, overflow in constant expression

constexpr       int iD = 1 << 16;  // Example D is the same as C, above.
constexpr       int jD = iD * iD;  // Error, overflow in constant expression
```

The code example above shows that an integer constant-expression overflow, absent **constexpr**, is not required by the C++ Standard to be treated as ill formed, whereas the initializer for a **constexpr** expression is required to report overflow as an error (not just a warning).

There is a strong association between **constexpr** variable and functions; see Section 2.1."**??**" on page **??**. Using a **constexpr** variable rather than just a **const** one forces the compiler to detect overflow within the body of **constexpr** function and report it — as a compile-time error — in a way that it would not otherwise be authorized to do.

For example, suppose we have two similar functions, `squareA` and `square`, defined for the built-in type (**signed**) **int** that each return the integral product of multiplying the argument with itself:

```
          int squareA(int i) { return i * i; }  // non-constexpr function
constexpr int squareB(int i) { return i * i; }  // constexpr function
```

Declaring a variable to be just **const** does nothing to force the compiler to check the evaluation of either function for overflow:

```
                int xA0 = squareA(1 << 15);  // OK
          const int xA1 = squareA(1 << 15);  // OK
constexpr       int xA2 = squareA(1 << 15);  // Error, squareA, not constexpr

                int yB0 = squareB(1 << 15);  // OK
          const int yB1 = squareB(1 << 15);  // OK
constexpr       int yB2 = squareB(1 << 15);  // OK

                int zC0 = squareB(1 << 16);  // Bug, zC0 is likely 0.
          const int zC1 = squareB(1 << 16);  // Bug, zC1  "   "    "
constexpr       int zC2 = squareB(1 << 16);  // Error, int overflow detected!
```

By declaring a variable to be not just **const** but **constexpr**, we make the compiler evaluate that (necessarily **constexpr**) function for those specific arguments at compile time, and, if there is overflow, immediately report the defect as being ill formed.

Internal Linkage

When a variable at file or namespace scope is either **const** or **constexpr**, and nothing explicitly gives it external like (e.g., being marked **extern**), it will have **internal linkage**. This means each translation unit will have its own copy of the variable.[3]

[3] In C++17, all **constexpr** variables are instead automatically **inline** as well, guaranteeing that there is only one instance in a program.

In general, only the values of such variables are relevant - their initializers are seen, they are utilized at compile time, and it has no effect if different translation units use different objects with the same value. Often, after compile time evaluation is completed, the variables themselves will no longer be needed and no actual address will be allocated for them at runtime. Only in cases where the address of the variable is used will this difference be observable.

Notably, **static** member variables do not get internal linkage. Because of this, if they are going to be used in a way that requires they have an address allocated at runtime they need to have a definition outside of their class; see **??** — *TODO (VR): missing annoyance, linker error with ODR-use* on page 31.

Use Cases

use-cases

Alternative to enumerated compile-time integral constants

me-integral-constants

It is not uncommon to want to express specific integral constants at compile time — e.g., for precomputed operands to be used in algorithms, mathematical constants, configuration variables, or any number of other reasons. A naive, brute-force approach might be to hard-code the constants where they are used:

```
int hoursToSeconds0(int hours)
    // Return the number of seconds in the specified hours.  The behavior is
    // undefined unless the result can be represented as an int.
{
    return hours * 3600;
}
```

This use of *magic constants* has, however, long been known[4] to make finding uses of the constants and the relationships between related ones needlessly difficult. For integral values only, we could always represent such compile-time constants symbolically by using a classic **enum** type (in deliberate preference to the modern, type-safe enumerator; see Section 2.1."**??**" on page **??**):

```
struct TimeRatios1  // explicit scope for single classic anonymous enum type
{
    enum  // anonymous enumeration comprising related symbolic constants
    {
        k_SECONDS_PER_MINUTE = 60,     // Underlying type (UT) might be int.
        k_MINUTES_PER_HOUR   = 60,
        k_SECONDS_PER_HOUR   = 60*60,  // these enumerators have the same UT
    };
};

int hoursToSeconds1(int hours)
    // ...
{
    return hours * TimeRatios1::k_SECONDS_PER_HOUR;
}
```

[4] **?**, section 1.5, pp. 19-22

This traditional solution, while often effective, gave little control to the underlying integral type of the enumerator used to represent the symbolic compile-time constant, leaving it at the mercy of the totality of values used to initialize its members. Such inflexibility might lead to compiler warnings and nonintuitive behavior resulting from **enum**-specific "integral-promotion" rules, especially when the **underlying type (UT)** used to represent the time ratios differs from the integral type with which they are ultimately used; see Section 2.1."**??**" on page **??**.

In this particular example, extending the **enum** to cover ratios up to a week and conversions down to nanoseconds would manifestly change its UT (because there are far more than 2^{32} nanoseconds in a week), altering how all of the enumerators behave when used in expressions with, say, values of type **int** (e.g., to **long**); see Section 1.1."**??**" on page **??**:

```
struct TimeRatios2  // explicit scope for single classic anonymous enum type
{
    enum  // Anonymous enumeration --- UT is governed by all of the enumerators.
    {
        k_SECONDS_PER_MINUTE = 60,    // UT might be long (or long long).
        k_MINUTES_PER_HOUR   = 60,
        k_SECONDS_PER_HOUR   = 3600,
        // ...
        k_USEC_PER_WEEK = 1000L*1000*60*60*24*7,  // same UT as all of the above
    };
};
```

The original *values* will remain unchanged, but the burden of all of the warnings resulting from the change in UT and rippling throughout a large codebase could be expensive to repair.

We would like the original values to remain unchanged (e.g., remain as **int** if that's what they were), and we want only those values that do *not* fit in an **int** to morph into a larger integral type. We might achieve this effect by placing each enumerator in its own separate anonymous enumeration:

```
struct TimeRatios3  // explicit scope for multiple classic anonymous enum types
{
    enum { k_SECONDS_PER_MINUTE = 60               };  // UT: int (likely)
    enum { k_MINUTES_PER_HOUR   = 60               };  //  "    "      "
    enum { k_SECONDS_PER_HOUR   = 60*60            };  //  "    "      "
    // ...
    enum { k_USEC_PER_SEC  = 1000*1000             };  // UT: int (v. likely)
    enum { k_USEC_PER_MIN  = 1000*1000*60          };  // UT: int (v. likely)
    enum { k_USEC_PER_HOUR = 1000U*1000*60*60      };  // UT: unsigned int
    enum { k_USEC_PER_DAY  = 1000L*1000*60*60*24   };  // UT: long
    enum { k_USEC_PER_WEEK = 1000L*1000*60*60*24*7 };  // UT: long
};
```

In this case, the original values as well as their respective UTs will remain unchanged and each new enumerated value will independently take on its own independent UT, which is either implemenation defined or else dictated by the number of bits required to represent the value, which is, in this case, non-negative.

A modern alternative to having separate anonymous **enum**s for each distinct value (or class of values) is to instead encode each ratio as an explicitly typed **constexpr** variable:

```
struct TimeRatios4
{
    static constexpr int k_SECONDS_PER_MINUTE = 60;
    static constexpr int k_MINUTES_PER_HOUR   = 60;
    static constexpr int k_SECONDS_PER_HOUR   = k_MINUTES_PER_HOUR *
                                                k_SECONDS_PER_MINUTE;
    // ...
    static constexpr long k_NANOS_PER_SECOND  = 1000*1000*1000;
    static constexpr long k_NANOS_PER_HOUR    = k_NANOS_PER_SECOND *
                                                k_SECONDS_PER_HOUR;
};

int hoursToSeconds(int hours)
    // ...
{
    return hours * TimeRatios4::k_SECONDS_PER_HOUR;
}

long hoursToNanos(int hours)
    // Return the number of nanoseconds in the specified hours.  The behavior
    // is undefined unless the result can be represented as a long.
{
    return hours * TimeRatios4::k_NANOS_PER_HOUR;
}
```

In the example above, we've rendered the **constexpr** variables as **static** members of a **struct** rather than placing them at namespace scope primarily to show that, from a user perspective, the two are syntactically indistinguishable — the substantive difference here being that a client would be prevented from unilaterally adding logical content to the "namespace" of a `TimeRatio` **struct**. When it comes to C-style **free functions**, however, the advantages for **static** members of a struct over namespace scope are many and unequivocal.[5]

lic-numeric-constants

Nonintegral symbolic numeric constants

Not all symbolic numeric constants that are needed at compile-time are necessarily integral. Consider, for example, the mathematical constants `pi` and `e`, which are typically represented as a floating point type, such as **double** (or **long double**).

The classical solution to avoid encoding this type of constant values as a *magic number* is to instead use a macro, such as is done in the `math.h` header on most operating systems:

```
#define M_E   2.7182818284590452354  /* e */
#define M_PI  3.14159265358979323846 /* pi */

double areaOfCircle(double radius)
{
```

[5] **?**, section 2.4.9, pp. 312-321, specifically Figure 2-23

```
    return 2 * M_PI * radius;
}
```

While this approach can be effective, it comes with all the well known downsides of using the C preprocessor. This approach is also fraught with risk such that the headers standard on many systems make these macros available in C or C++ only upon request by **#define**ing specific macros before including those headers.[6]

Another safer and far less error-prone solution to name collisions is to instead use a **constexpr** variable for this form of nonintegral constant. Note that, while the macro is defined with more precision to be able to initialize variables of possibly higher-precision floating-point types, here we need only enough digits to uniquely identify the appropriate **double** constant:

```
struct NumericConstants
{
    static constexpr double k_E  = 2.718'281'828'459'045;  /* e */
    static constexpr double k_PI = 3.141'592'653'589'793;  /* pi */
};

double areaOfCircle(double radius)
{
    return 2 * NumericConstants::k_PI * radius;
}
```

In the example above, we have made valuable use of a safe C++14 feature to help identify the needed precision of the numeric literal; see Section 1.2."**??**" on page **??**. Beyond the potential name collisions and global name pollution, preferring a constexpr variable over a C preprocessor macro has the added benefit of making explicit the C++ type of constant being defined. Note that supplying digits beyond what are significant will nonetheless be silently ignored.

nstexpr-data-structures

Storing `constexpr` data structures

Precomputing values (at compile time) for subsequent use (at run time) is one impactful use of **constexpr** *functions*, but see *Potential Pitfalls* — **constexpr** *function pitfalls* on page 30. Storing these values in explicitly **constexpr** variables ensures that the values are (1) guaranteed to be computed at compile time and not, for example, at startup as the result of a (dangerous) runtime initialization of a file- or namespace-scoped variable and (2) usable as part of the evaluation of any constant expression; see Section 2.1."**??**" on page **??**.

Rather than attempting to circumvent the draconian limitations of the C++11 version of **constexpr** functions, we will make use of the relaxed restrictions of C++14. For this, we will define a class template that initializes an array member with the results of a **constexpr** functor applied to each array index[7]:

```
    std::size_t
```

[6] See your library documentation for details.

[7] Note that, in C++17, most of the manipulators of `std::array` have been changed to be **constexpr** and, when combined with the relaxation of the rules for **constexpr** evaluation in C++14 (see Section 2.2."**??**" on page **??**), this compile-time-friendly container provides a simple way to define functions that populate tables of values.

```
template <typename T, std::size_t N>
struct ConstexprArray
{
private:
    T d_data[N];  // data initialized at construction

public:
    template <typename F>
    constexpr ConstexprArray(const F &func)
    : d_data{}
    {
        for (int i = 0; i < N; ++i)
        {
            d_data[i] = func(i);
        }
    }

    constexpr const T& operator[](std::size_t ndx) const
    {
        return d_data[ndx];
    }
};
```

The numerous alternative approaches to writing such data structures, vary in their complexity, trade-offs, and understandability. In this case, we default initialize our elements before populating them but do not need to rely on any other significant new language infrastructure. Other approaches could be taken; see Section 2.1."**??**" on page **??**.

Given this utility class, we can then precompute at compile time any function that we can express as a **constexpr** function, such as a simple table of the first `N` squares:

```
constexpr int square(int x) { return x * x; }

constexpr ConstexprArray<int, 500> squares(square);

static_assert(squares[1]   == 1,    "");
static_assert(squares[289] == 83521,"");
```

Note that, as with many applications of **constexpr** functions, attempting to initialize a large array of **constexpr** variables will quickly bump up against compiler-imposed template instantiation limits. What's more, attempting to perform more complex arithmetic at compile time would be likely to exceed computation limits as well.

avior-at-compile-time

Diagnosing undefined behavior at compile time

Avoiding overflow during intermediate calculations is an important consideration, especially from a security perspective, and yet is a generally difficult-to-solve problem. Forcing computations to occur at compile time brings the full power of the compiler to bear in addressing such undefined behavior.

As an academically interesting example of this eminently practical security problem, suppose we want to write a (compile-time) function in C++ to compute the **Collatz length** of

an arbitrary positive integer and generate a compilation error if any intermediate calculation would result in signed integer overflow.

First let's take a step back now to understand what we are talking about here with respect to Collatz length. Suppose we have a function `cf` that takes a positive **`int`**, `n`, and for even `n` returns `n/2` and for odd `n` returns `3n+1`.

```
int cf(int n) { return n % 2 ? 3 * n + 1 : n / 2; }  // Collatz function
```

Given a positive integer, `n`, the **Collatz sequence**, `cs(n)`, is defined as the sequence of integers generated by repeated application of the **Collatz function** — e.g, `cs(1) = { 4, 2, 1, 4, 2, 1, 4, ... };`, `cs(3) = { 10, 5, 16, 8, 4, 2, 1, 4, ... }`, and so on. A classic (but as yet unproven) conjecture in mathematics states that, for every positive integer, n, the Collatz sequence for n will eventually reach 1. The Collatz length of the positive integer n is the number of iterations of the Collatz function needed to reach 1, starting from n. Note that the Collatz sequence for `n = 1` is `{ 1, 1, 1, ... }` and its Collatz length is 0.

This example showcases the need for a **`constexpr`** variable in that we need to create a **constexpr context** – i.e., one requiring a constant expression to require that the evaluation of a **`constexpr`** function occur at compile time. Again, to avoid distractions related to implementing more complex functionality within the limitations of C++11 **`constexpr`** functions, we will make use of the relaxed restrictions of C++14; see Section 2.1."**??**" on page **??**:

```
constexpr int collatzLength(long long number)
    // Return the length of the Collatz sequence of the specified number. The
    // behavior is undefined unless each intermediate sequence member can be
    // expressed as an unsigned long long.
{
    int length = 0;         // collatLength(1) is 0.

    while (number > 1)      // The current value of number is not 1.
    {
        ++length;           // Keep track of the length of the sequence so far.

        if (number % 2)     // if the current number is odd
        {
            number = 3 * number + 1;    // advance from odd sequence value
        }
        else
        {
            number /= 2;                // advance from even sequence value
        }
    }

    return length;
}

const     int c1 = collatzLength(942488749153153);  // OK, 1862
constexpr int x1 = collatzLength(942488749153153);  // OK, 1862
```

```
const int c2 = collatzLength(104899295810901231);
    // Bug, program aborts at runtime.

constexpr int x2 = collatzLength(104899295810901231);
    // Error, overflow in constant evaluation
```

In the example above, the variables `c1` and `x1` can be initialized correctly at compile-time, but `c2` and `x2` cannot. The non**constexpr** nature of `c2` allows the overflow to occur and exhibit undefined behavior — integer overflow — at run time. On the other hand, the variable `x2`, due to its being declared **constexpr**, forces the computation to occur at compile time, thereby discovering the overflow and dutifully invoking the *non***constexpr** `std::abort` function, which in turn generates the desired error at compile time.

Potential Pitfalls

pitfalls-constexprvar

`constexpr` function pitfalls

xpr-function-pitfalls

Many of the uses of **constexpr** variables involve a corresponding use of **constexpr** functions (see Section 2.1."**??**" on page **??**). The pitfalls related to **constexpr** functions are similarly applicable to the variables in which their results might be stored. In particular, it can be profoundly advantageous to forgo use of the **constexpr** function feature altogether, do the precomputation externally to program, and embed the calculated result — along with a comment containing source text of the (e.g., Perl or Python) script that performed the calculation — into the C++ program source itself.

Annoyances

annoyances

`static` member variables require external definitions

In most situations there is little behavioral difference between a variable at file or namespace scope and a **static** member variable - primarily they differ only in name lookup and access control restrictions. When they are **constexpr**, however, they behave very differently when they need to exist at runtime. A file or namespace scope variable will have internal linkage, allowing free use of its address with the understanding that that address will be different in different translation units:

```
// common.h:

constexpr int c = 17;
const int *f1();
const int *f2();

// file1.cpp:
#include <common.h>

const int *f1() { return &c; }

// file2.cpp:
#include <common.h>
```

```
const int *f2() { return &c; }

// main.cpp:
#include <common.h>
#include <cassert>  // standard C assert macro

int main()
{
    assert( f1() != f2() );    // Different addresses in memory per TU.
    assert( *f1() == *f2() );  // Same value.
    return 0;
}
```

For **static** data members, however, things become more difficult. While the **declaration** in the class **definition** needs to have an initializer, that is not itself a definition and will not result in static storage being allocated at runtime for the object, ending in a linker error when you try to build an application that tries to reference it:[8]

```
struct S {
    static constexpr int d_i = 17;
};
void useByReference(const int& i) { /* ... */ }

int main()
{
    const int local = S::d_i;  // OK, value is only used at compile time.
    useByReference(S::d_i);    // Link-Time Error, S::d_i not defined
    return 0;
}
```

This link-time error would be removed by adding a definition of `S::d_i`. Note that the initializer needs to be skipped, as it has already been specified in the definition of `S`:

```
constexpr int S::d_i;  // Define constexpr data member.
```

TODO (VR): missing annoyance, linker error with ODR-use

TODO. We definitely need this one, it's when you have a static **constexpr** variable data member and then you get an unexpected linker error due to ODR-use. See https://stackoverflow.com/questions/43193749/linker-error-for-constexpr-static-member-variable-in-gcc-and-clang. Note that it was fixed in C++17, because static constexpr variables are inline variables by default there.

No static `constexpr` member variables defined in their own class

When implementing a class using the singleton pattern, it may seem desirable to have the single object of that type be a **constexpr private static** member of the class itself, with

[8] The C++17 change to make **constexpr** variables **inline** also applies to **static** member variables, removing the need to provide external definitions when they are used.

guaranteed compile-time (data-race-free) initialization and no direct accessibility outside the class. This does not work as easily as planned because **constexpr static** data members must have a complete type and the class being defined is not complete until its closing brace:

```
class S
{
    static const     S constVal;      // OK, initialized outside class below
    static constexpr S constexprVal;  // Error, constexpr must be initialized.
    static constexpr S constInit{};   // Error, S is not complete.
};

const S S::constVal{};  // OK, initialize static const member.
```

The "obvious" workaround of applying a more traditional singleton pattern, where the singleton object is a static local variable in a function call, also fails (see Section 1.1."**??**" on page **??**) because **constexpr** functions are not allowed to have static variables (see Section 2.1."**??**" on page **??**):

```
constexpr const S& singleton()
{
   static constexpr S object{};  // Error, even in C++14, static is not allowed.
   return object;
}
```

The only fully-featured solution available for **constexpr** objects of static storage duration is to put them outside of their type, either at global scope, **namespace** scope, or nested within a befriended helper class[9]:

```
class S
{
    friend struct T;
    S() = default;  // private
    // ...
};

struct T
{
    static constexpr S constexprS{};
};
```

see-also

See Also

- "**??**" (§2.1, p. **??**) ♦ can be used to initialize a **constexpr** variable.
- "**??**" (§2.1, p. **??**) ♦ provide a convenient way of initializing a **constexpr** variable of a UDT with a compile-time value.

[9] C++20 provides an alternate partial solution with the **constinit** keyword, allowing for compile-time initialization of static data members, but that still does not make such objects usable in a constant expression.

further-reading

Further Reading

TODO

sec-conditional-cpp14

Chapter 3

Unsafe Features

ch-unsafe

sec-unsafe-cpp11 Intro text should be here.

Chapter 3 Unsafe Features

sec-unsafe-cpp14