# Back To Basics
## Code Review

CHANDRANATH BHATTACHARYYA
& KATHLEEN BAKER

Cppcon
The C++ Conference
2025
September 13 - 19

# Motivation for this talk

- We work at Microsoft on the Edge browser, a Chromium based C++ project.
- This talk features some common code review feedback.
- Can be used as a reference when reviewing code.
- C++ Core Guidelines
- Chromium C++ style guide
- Chromium C++ Dos and Don'ts

# Overview of our project

- C++20, no exception handling
- Clang
- 100s of engineers

# Contents

- Each slide will have a cryptic title, “bad” code, “good” code, and the guideline.
- We’ll be using printf, puts, and std::cout for creating snippets of code so you can ignore those.

# Compiler Warnings and Clang-Tidy Checks



```
-Wswitch (enabled by default)
-Wswitch-enum
-Wrange-loop-construct (-Wall)
-Wpessimizing-move (-Wall)
-Wunused-variable (-Wall)
-Wunused-parameter (-Wextra)
-Wunused-private-field (-Wall)
-Wreorder-ctor (-Wall)
-Wreturn-type (-Wall)
```

```
performance-unnecessary-value-param
cppcoreguidelines-pro-type-member-init
readability-convert-member-functions-to-static
readability-make-member-function-const
performance-noexcept-move-constructor
cppcoreguidelines-rvalue-reference-param-not-moved
cppcoreguidelines-missing-std-forward
bugprone-use-after-move
modernize-use-emplace
readability-container-size-empty
modernize-use-ranges
bugprone-inaccurate-erase
readability-container-contains
```

# Scope

# Variables on the loose – if statements



```cpp
struct SomeClass {
  // Other stuff.
  bool HasPropOne() const;
  bool HasPropTwo() const;
  // Other stuff.
};
SomeClass Foo();
```

```cpp
int main() {
  const SomeClass s = Foo();
  if (s.HasPropOne()) {
    // Do stuff with `s`.
  } else if (s.HasPropTwo()) {
    // Do other stuff with `s`.
  }
}
```

```cpp
int main() {
  if (const SomeClass s = Foo(); s.HasPropOne()) {
    // Do stuff with `s`.
  } else if (s.HasPropTwo()) {
    // Do other stuff with `s`.
  }
}
```

With C++17, we can restrict the scope of the variable used in the if/else block.

# Variables on the loose – switch statements

```cpp
enum class SomeEnum { kOne, kTwo };

struct SomeClass {
  // Other stuff.
  SomeEnum GetType() const;
  // Other stuff.
};

SomeClass Foo();
```

```cpp
int main() {
  const SomeClass s = Foo();
  switch (s.GetType()) {
    case SomeEnum::kOne:
      // Some stuff.
      break;
    case SomeEnum::kTwo:
      // Some stuff.
      break;
  }
}
```

With C++17, we can restrict the scope of the variable used in the switch block.

```cpp
int main() {
  switch (const SomeClass s = Foo(); s.GetType()) {
    case SomeEnum::kOne:
      // Some stuff.
      break;
    case SomeEnum::kTwo:
      // Some stuff.
      break;
  }
}
```

# Variables on the loose – for loops

```c
int main() {
  int i;
  for (i = 0; i < 3; ++i) {
    printf("i=%d\n", i);
  }
  // i is not used from here on.
}
```

```c
int main() {
  for (int i = 0; i < 3; ++i) {
    printf(“i=%d\n", i);
  }
  // i is not used from here on.
}
```

Scope the variable to the for loop if it isn’t used in later code.

# Enums

# Case of the missing case

```
enum class Result {
  kSuccess,
  kFailure1,
  kFailure2,
};
```

```
enum class Result {
  kSuccess,
  kFailure1,
  kFailure2,
  kFailure3, // new value added.
};
```

If the enum is updated, the switch statement with default will return the wrong result.

```
constexpr std::string_view ToString(Result result) {
  switch (result) {
    case Result::kSuccess:
      return "Success";
    case Result::kFailure1:
      return "Failure1";
    default:
      return "Failure2";
  }
}
```

```
constexpr std::string_view ToString(Result result) {
  switch (result) {
    case Result::kSuccess:
      return "Success";
    case Result::kFailure1:
      return "Failure1";
    case Result::kFailure2:
      return "Failure2";
    case Result::kFailure3:
      return "Failure3";
  }
  NOTREACHED();
}
```

NOTREACHED() is a macro in Chromium that annotates unreachable code, terminates, and produces a crash dump.

Use all cases in a switch statement.

# Use all cases in switch statement

```cpp
enum class Result {
  kSuccess,
  kFailure1,
  kFailure2,
  kFailure3, // new value added.
};
```

```cpp
constexpr std::string_view ToString(Result result) {
  switch (result) {
    case Result::kSuccess:
      return "Success";
    case Result::kFailure1:
      return "Failure1";
    case Result::kFailure2:
      return "Failure2";
  }
}
```

***-Wswitch*** will detect a missing enum in a switch statement with no "default".

```
warning: enumeration value 'kFailure3' not handled in switch [-Wswitch]
   switch (result) {
           ^~~~~~
warning: non-void function does not return a value in all control paths [-Wreturn-type]
 }
 ^
```

***-Wswitch-enum*** can be used to detect a missing enum when using "default".

# The type that bit back

```cpp
enum FileStatus
{
  OPEN,
  CLOSED
};
```

```cpp
int main() {
  const FileStatus s = OPEN;
  // BAD: Allows implicit conversion.
  const int s_int = s;
  std::ignore = s_int;
}
```

```cpp
enum class FileStatus {
  OPEN,
  CLOSED
};
```

```cpp
int main() {
  const auto s = FileStatus::OPEN;
  // NOT ALLOWED: Compilation error.
  // const int s_int = s;
}
```

```cpp
enum DoorStatus
{
  OPEN,
  CLOSED
};
```

```
 OPEN,
   ^
note: previous definition is here
 enum FileStatus { OPEN, CLOSED };
                   ^
error: redefinition of enumerator 'CLOSED'
   CLOSED
   ^
note: previous definition is here
 enum FileStatus { OPEN, CLOSED };
                         ^
```

```cpp
enum class DoorStatus {
  OPEN,
  CLOSED
};
```

```cpp
enum FileStatus;

void Foo(FileStatus);
```

```
error: ISO C++ forbids forward references to 'enum' types
```

```cpp
enum class FileStatus;

void Foo(FileStatus);
```

Use enum class to get better type safety, avoid name clashes, and allow forward declaration.

# Use enum class

```cpp
enum Color { RED, GREEN };
enum Status { OK, ERROR };
```

```
warning: comparison of different enumeration types ('Status' and 'Color') is deprecated
[-Wdeprecated-enum-compare]
   if (OK == RED) {
```

```cpp
int main() {
  // Deprecated by C++20.
  if (OK == RED) {
  }
}
```

Deprecated by p1120

```cpp
enum class Color { RED, GREEN };
enum class Status { OK, ERROR };
```

```cpp
int main() {
  if (Status::OK == Color::RED) {
  }
}
```

# Name drop with style

```
namespace my_component {
enum class MyComponentSpecialEnum {
  kValue1,
  kValue2,
  kValue3,
  kValue4,
  kValue5
};
}  // namespace my_component
```

Use using enum before switch statement.

```
std::string GetStringFor(my_component::MyComponentSpecialEnum value,
                         std::string_view prefix) {
  const auto enum_str = [&]() -> std::string_view {
    switch (value) {
      case my_component::MyComponentSpecialEnum::kValue1:
        return "Value1";
      case my_component::MyComponentSpecialEnum::kValue2:
        return "Value2";
      case my_component::MyComponentSpecialEnum::kValue3:
        return "Value3";
      case my_component::MyComponentSpecialEnum::kValue4:
        return "Value4";
      case my_component::MyComponentSpecialEnum::kValue5:
        return "Value5";
    }
  }();
  return StrCat({prefix, enum_str});
}
```

```
std::string GetStringFor(my_component::MyComponentSpecialEnum value,
                         std::string_view prefix) {
  const auto enum_str = [&]() -> std::string_view {
    using enum my_component::MyComponentSpecialEnum;
    switch (value) {
      case kValue1:
        return "Value1";
      case kValue2:
        return "Value2";
      case kValue3:
        return "Value3";
      case kValue4:
        return "Value4";
      case kValue5:
        return "Value5";
    }
  }();
  return StrCat({prefix, enum_str});
}
```

# Iteration

# No more index games

```cpp
void Foo(const std::vector<int>& vec) {
  for (size_t i = 0u; i < vec.size(); ++i) {
    const auto val = vec[i];
    // Use `val`.
    std::ignore = val;
  }
}
```

```cpp
void Foo(const std::vector<int>& vec) {
  for (const auto val : vec) {
    // Use `val`.
    std::ignore = val;
  }
}
```

```cpp
void Foo(const std::list<int>& l) {
  for (auto it = l.begin(); it != l.end(); ++it) {
    const auto val = *it;
    // Use `val`.
    std::ignore = val;
  }
}
```

```cpp
void Foo(const std::list<int>& l) {
  for (const auto val : l) {
    // Use `val`.
    std::ignore = val;
  }
}
```

Use range based for loop or std::ranges::for_each.

Using ranges::for_each:

```cpp
void Foo(const std::vector<int>& vec) {
  std::ranges::for_each(vec, [](auto val) {
    // Use `val`.
    std::ignore = val;
  });
}
```

```cpp
void Foo(const std::list<int>& l) {
  std::ranges::for_each(l, [](auto val) {
    // Use `val`.
    std::ignore = val;
  });
}
```

Check this article and this article to consider use cases for for_each.

Using std::ignore in this manner was supported by all compilers since C++11, but it became standardized in C++26 with p2968

# Don’t clone what you can borrow



```
void Foo(const std::vector<std::string>& vec) {
  for (const auto val : vec) {
    // Use `val`.
    std::ignore = val;
  }
}
```

```
void Foo(const std::vector<std::string>& vec) {
  for (const auto& val : vec) {
    // Use `val`.
    std::ignore = val;
  }
}
```

```
void Foo(const std::vector<std::string>& vec) {
  for (auto&& val : vec) {
    // Use `val`.
    std::ignore = val;
  }
}
```

Use const auto& or auto&& in range based for loops with non-trivial objects.

***-Wrange-loop-construct*** is included in -***Wall***

When we use ***-Wrange-loop-construct:***

```
error: loop variable 'val' creates a copy from type 'const std::string' [-Werror,-Wrange-loop-construct]
   for (const auto val : vec) {
                   ^
<source>:18:8: note: use reference type 'const std::string &' to prevent copying
   for (const auto val : vec) {
        ^~~~~~~~~~~~~~~~
                   &
```

# The key to clarity



```
void Foo(const std::map<int, std::string>& m) {
  for (auto it : m) {
    std::cout << "Key: " << it.first << ", Val: " << it.second << '\n';
  }
}
```

```
void Foo(const std::map<int, std::string>& m) {
  for (const auto& [key, value] : m) {
    std::cout << "Key: " << key << ", Val: " << value << '\n';
  }
}
```

Use structured binding with & for map traversal.

# Passing parameters

# Don’t clone the elephant for a walk

```cpp
void Print(std::string s) {
  std::cout << s << '\n';
}
```

```cpp
void PrintBetter(const std::string& s) {
  std::cout << s << '\n';
}
```

Pass non-trivial read-only objects as const reference to prevent unnecessary copies.

The clang-tidy check ***--checks=[performance-unnecessary-value-param](performance-unnecessary-value-param)*** will flag this.

```
warning: the parameter 's' is copied for each invocation but only used as a const reference;
consider making it a const reference [performance-unnecessary-value-param]
 void Print(std::string s) {
                        ^
            const      &
```

std::string is “special”. We will see some alternate strategies for string-as-argument in later slides.

# Const without context is just noise

```
struct Point {
  int x = 0;
  int y = 0;
};
```

```
void PrintPoint(const Point pt);
```

→

```
void PrintPoint(Point pt);
```

**const** here is not part of the function signature, so both declarations are same.

```
void PrintPoint(const Point* pt);
void PrintPoint(const Point& pt);
```

const */& are part of the function signature.

const is not needed on value parameters in function declarations.

```
// Header file
void PrintPoint(const Point pt);

// Source file
void PrintPoint(const Point pt) {
  std::cout << pt.x << ", " << pt.y << '\n';
}
```

→

```
// Header file
void PrintPoint(Point pt);

// Source file
void PrintPoint(const Point pt) {
  std::cout << pt.x << ", " << pt.y << '\n';
}
```

# Feather is light, let it fly

```cpp
void PrintInt(const int& i) {
  std::cout << i << '\n';
}
```

```cpp
void PrintInt(int i) {
  std::cout << i << '\n';
}
```

```cpp
struct Point {
  int x = 0;
  int y = 0;
};
```

```cpp
void PrintPoint(const Point& pt) {
  std::cout << pt.x << ", " << pt.y << '\n';
}
```

```cpp
void PrintPoint(Point pt) {
  std::cout << pt.x << ", " << pt.y << '\n';
}
```

Passing trivial objects by value is preferred because passing by reference can prevent optimizations.

The compiler can optimize this code so i, x, and y are passed via registers and don’t need to be dereferenced.

# The final destination deserves a shortcut



```cpp
struct A {
  A(const std::string& s) : str_(s) {}

  void SinkString(const std::string& s) {
    str_ = s;
  }

  std::string str_;
};
```

```cpp
struct A {
  A(std::string&& s) : str_(std::move(s)) {}

  void SinkString(std::string&& s) {
    str_ = std::move(s);
  }

  std::string str_;
};
```

Follow the cpp core guideline: *F.18: For "will-move-from" parameters, pass by X&& and std::move the parameter.*

# Don’t hold the book to read the title

```
void Foo(const std::string& s) {
  std::cout << s << '\n';
}
```

```
void FooBetter(std::string_view s) {
  std::cout << s << '\n';
}
```

Use std::string_view instead of const std::string& for read-only strings in non-sink cases.

std::string_view can handle std::string, const char* and {const char*, len} as arguments without the need to create different functions for each. It does not do any heap allocation.

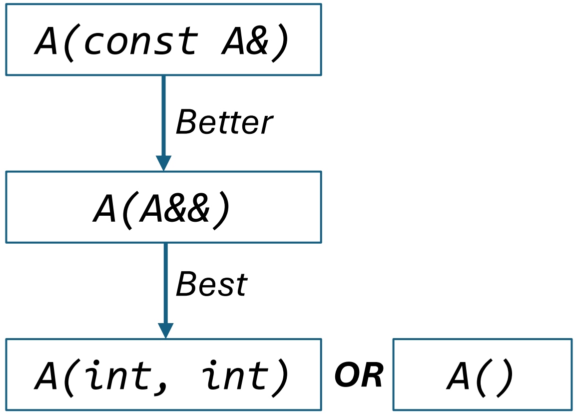
```
int main() {
  // Created at runtime.
  const std::string str("Hello");

  // Creates temporary string.
  Foo("Hello");

  // No temporary string is created.
  FooBetter("Hello"); // Allows conversion from `const char*`.
  FooBetter(str);  // Allows conversion from `std::string`.
  FooBetter(
    {str.c_str(), str.size() - 1});  // Allows conversion from {const char*, len}.
}
```

# Look, don't touch the collection

```cpp
struct A final {
  // Constructor
  A() { puts("A()"); }
  A(int, int) { puts("A(int, int)"); }

  // Destructor
  ~A() { puts("~A()"); }

  // Copy constructor
  A(const A&) { puts("A(const A&)"); }

  // Move constructor
  A(A&&) noexcept { puts("A(A&&)"); }

  // Copy assignment operator
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }

  // Move assignment operator
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

***struct A*** will be the placeholder for "non-trivial" object that we will use for most of this presentation.

The print statements in its special member functions help us in understanding whether they are getting called.



# Look, don’t touch the collection

```
struct A {
  A(int, int) { puts("A(int, int)"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
void Foo(const std::vector<A>& v) {
  // Use v.
}
```

```
void FooBetter(std::span<const A> v) {
  // Use v.
}
```

```
int main() {
  // Cannot be `constexpr` since A constructor is not `constexpr`.
  const A arr[] = {{1,2}, {3,4}, {5,6}};

  std::cout << "=== Before Foo ===\n";
  // Create temporary vector.
  Foo({arr, arr + 3});

  std::cout << "=== Before FooBetter ===\n";
  FooBetter(arr);
}
```

```
A(int, int)
A(int, int)
A(int, int)
=== Before Foo ===
A(const A&)
A(const A&)
A(const A&)
~A()
~A()
~A()
=== Before FooBetter ===
~A()
~A()
~A()
```

Use std::span instead of std::vector or std::array for read-only containers in non-sink cases to prevent vector creation.

# Parameter Passing: Use std::span

```
void FooBetter(std::span<const A> v) {
  // Use v.
}
```

```
int main() {
  std::vector<A> v = {{1, 2}, {3, 4}, {5, 6}};
  FooBetter(v);
  std::array<A, 3> a = {A{1, 2}, A{3, 4}, A{5, 6}};
  FooBetter(a);
  std::initializer_list<A> l = {A{1, 2}, A{3, 4}, A{5, 6}};
  FooBetter(l);
  FooBetter({{A{1, 2}, A{3, 4}, A{5, 6}}});
}
```

This creates “extra” objects. Appendix has slides explaining this scenario.

std::span allows the function to be used with C style arrays, vectors, arrays, initializer_list.

# Returning from functions

# Success with substance



```cpp
bool GetNameById(int id, std::string* name) {
  if (id == 42) {
    *name = "Denver";
    return true;
  }
  return false;
}
```

```cpp
int main() {
  std::string name;
  if (GetNameById(42, &name)) {
    std::cout << "Name: " << name << '\n';
  } else {
    std::cout << "Not found\n";
  }
}
```

```cpp
std::optional<std::string> GetNameById(int id) {
  if (id == 42) {
    return "Denver";
  }
  return std::nullopt;
}
```

```cpp
int main() {
  if (const auto name = GetNameById(42)) {
    std::cout << "Name: " << *name << '\n';
  } else {
    std::cout << "Not found\n";
  }
}
```

Use std::optional as return type when the function can return bool to indicate success / failure along with some value only for success case.

# A result with a story

```
bool ParseInt(const std::string& input, int* value, std::string* error) {
  try {
    *value = std::stoi(input);
    return true;
  } catch (const std::exception& e) {
    *error = e.what();
    return false;
  }
}
```

Use std::expected to converge the return type for a function when appropriate.

```
void TestParse(const std::string& str) {
  std::cout << str << ": ";
  int value = 0;
  std::string error;
  if (ParseInt(str, &value, &error)) {
    std::cout << "Parsed: " << value << '\n';
  } else {
    std::cout << "Error: " << error << '\n';
  }
}
```

```
std::expected<int, std::string> ParseInt(const std::string& input) {
  try {
    return std::stoi(input);
  } catch (const std::exception& e) {
    return std::unexpected<std::string>(e.what());
  }
}
```

```
int main() {
  TestParse("100");
  TestParse("100 and some");
  TestParse("hundred");
}
```

```
100: Parsed: 100
100 and some: Parsed: 100
hundred: Error: stoi: no conversion
```

```
void TestParse(const std::string& str) {
  std::cout << str << ": ";
  if (const auto result = ParseInt(str)) {
    std::cout << "Parsed: " << *result << '\n';
  } else {
    std::cout << "Error: " << result.error() << '\n';
  }
}
```

# Premature relocation is costly

```
struct A {
  A(int, int) { puts("A(int, int)"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
A Foo() {
  A a{10, 10};
  return std::move(a);
}
```

```
int main() {
  std::ignore = Foo();
}
```

```
A(int, int)
A(A&&)
~A()
~A()
```

→

```
A Foo() {
  A a{10, 10};
  return a;
}
```

```
A(int, int)
~A()
```

Explicit ***std::move*** calls defeats NRVO (Named Return Value Optimization)

Don't use ***std::move*** in such cases.

NRVO is not "required" by standard. But most compilers implement it for such scenarios.

When we use:
***-Wpessimizing-move***

```
error: moving a local object in a return statement prevents copy elision [-Werror,-Wpessimizing-move]
   return std::move(a);
          ^
note: remove std::move call here
   return std::move(a);
```

# Elide the middleman

```
struct A {
  A(int, int) { puts("A(int, int)"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
A Foo() {
  A a{10, 10};
  return a;
}
```

→

```
A Foo() {
  return {10, 10};
}
```

```
int main() {
  std::ignore = Foo();
}
```

```
A(int, int)
~A()
```

```
A(int, int)
~A()
```

This is guaranteed copy elision from C++17.

Also known as:

a) Deferred Temporary materialization
b) Unmaterialized value passing.

Prefer using copy elision instead of depending on Named Return Value optimization.

Named object return causes compilation errors for non-copyable & non-movable types.
Copy elision works for these cases.

# A sealed envelope for no reason

```
struct A {
  A(int a, int b) { printf("A(%d, %d)\n", a, b); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
const A Foo() {
  return {10, 10};
}
```

```
A Foo() {
  return {10, 10};
}
```

```
int main() {
  {
    std::vector<A> vec;
    vec.push_back(Foo());
  }
  puts("==========");
  {
    A a{20, 30};
    a = Foo();
  }
}
```

**const** return defeats move operations

Don't return const value from function.

```
A(10, 10)
A(const A&)
~A()
~A()
==========
A(20, 30)
A(10, 10)
A& operator=(const A&)
~A()
~A()
```

```
A(10, 10)
A(A&&)
~A()
~A()
==========
A(20, 30)
A(10, 10)
A& operator=(A&&)
~A()
~A()
```

# Don’t return const value from function

```
const A Foo() {
  return {10, 10};
}
```

```
A Foo() {
  return {10, 10};
}
```

**const** return defeats move operations

Don’t return const value from function.

Following works for both:

```
[[maybe_unused]] const A a1 = Foo();
[[maybe_unused]] A a2 = Foo();
```

```
// Fails to compile with `const A Foo()`.
Foo() = A{10, 20};
```

```
// Compiles with `A Foo()`.
Foo() = A{10, 20};
```

```
int GetInt() { return 10; }

GetInt() = 10;
```

```
struct A {
  A(int a, int b) { printf("A(%d, %d)\n", a, b); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) & {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) & noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
// Fails to compile with
// `const A Foo()`.
Foo() = A{10, 20};
```

# Class Design

# Init to win it



```cpp
struct Size {
  Size() {
    width = 0;
    height = 0;
  }

  int width;
  int height;
};
```

```cpp
struct Size {
  int width = 0;
  int height = 0;
};
```

Initializing at the point of declaration, when possible, reduces the possibility of mistakes with missing out on initialization.

```cpp
struct Size {
  Size() {
    width = 0;
    // height = 0;
  }

  int width;
  int height;
};
```

*Clang-tidy:* ***--checks=*** [cppcoreguidelines-pro-type-member-init]

```
warning: constructor does not initialize these fields: height
[cppcoreguidelines-pro-type-member-init]
   Size() {
   ^
     width = 0;
     // height = 0;
   }

   int width;
   int height;
```

# List it or risk it

```
class Person {
 public:
  Person(const std::string& name, int age) {
    name_ = name;
    age_ = age;
  }

 private:
  std::string name_;
  int age_ = 0;
};
```

Initialize in the member initialization list.

```
class Person {
 public:
  Person(const std::string& name, int age) : name_(name), age_(age) {}

  // To remove `-Wunused-private-field` error.
  int age() const { return age_; }

 private:
  std::string name_;
  int age_ = 0;
};
```

# Initialize in the member initialization list

```cpp
class MyClass {
 public:
  MyClass() {
    member_b_ = member_a_ + 1;
    member_a_ = 10;
  }

 private:
  // Declared first, but initialized second.
  int member_a_ = 0;
  // Declared second, but initialized first.
  int member_b_ = 0;
};
```

```cpp
class MyClass {
 public:
  MyClass() : member_b_(member_a_ + 1), member_a_(10) {}

 private:
  // Declared first, but initialized second in the list.
  int member_a_ = 0;
  // Declared second, but initialized first in the list.
  int member_b_ = 0;
};
```

```
-Wreorder-ctor (-Wall)
```

```
error: field 'member_b_' will be initialized after field 'member_a_'
[-Werror,-Wreorder-ctor]
   MyClass() : member_b_(member_a_ + 1), member_a_(10) {}
               ^~~~~~~~~~~~~~~~~~~~~~~~  ~~~~~~~~~~~~~
               member_a_(10)             member_b_(member_a_ + 1)
```

```cpp
class MyClass {
 public:
  MyClass() : member_a_(10), member_b_(member_a_ + 1) {}

  // To remove `-Wunused-private-field` error.
  int member_b() const { return member_b_; }

 private:
  // Declared first, but initialized first in the list.
  int member_a_ = 0;
  // Declared second, initialized second in the list.
  int member_b_ = 0;
};
```

Member initialization list usage forces us to ensure the correct initialization order.

# Detached duties

```
class Logger {
 public:
  void PrintHello() { std::cout << "Hello, world!\n"; }
};
```

```
class Logger {
 public:
  static void PrintHello() { std::cout << "Hello, world!\n"; }
};
```

```
int main() {
  Logger::PrintHello();
}
```

```
// header.h
namespace logger {
void PrintHello();
}  // namespace logger
```

If a member function does not access any non-static member variables or functions, consider making it **static**.

```
// source.cc
namespace logger {
void PrintHello() {
  std::cout << "Hello, world!\n";
}
}  // namespace logger
```

If a function is not conceptually tied to the class, move it into a namespace.

# Make member functions static as appropriate

```
class Logger {
 public:
  void PrintHello() { std::cout << "Hello, world!\n"; }
};
```

*Clang-tidy:* ***--checks=***[*readability-convert-member-functions-to-static*]()

```
warning: method 'PrintHello' can be made static [readability-convert-member-functions-to-static]
   void PrintHello() { std::cout << "Hello, world!\n"; }
        ^
   static
```

# Don’t crowd the living room

```
// header file: my_class.h
class MyClass {
 public:
  int DoSomething();

 private:
  static int CalculateResult(int a, int b);
};
```

```
// source file: my_class.cc
int MyClass::DoSomething() {
  return CalculateResult(10, 20);
}

int MyClass::CalculateResult(int a, int b) {
  return a + b;
}
```

```
// header file: my_class.h
class MyClass {
 public:
  int DoSomething();
};
```

```
// source file: my_class.cc
namespace {
int CalculateResult(int a, int b) {
  return a + b;
}
}  // namespace


int MyClass::DoSomething() {
  return CalculateResult(10, 20);
}
```

Move private static functions out of the class into unnamed namespace in the source file.

The move to the source file helps reduce dependency and keeps the header file simpler.

# The redundant ritual



```cpp
// header file: non_optimal.h
extern void MyFunction();
```

```cpp
// source file: non_optimal.cc
#include "non_optimal.h"

#include <iostream>

void MyFunction() {
  std::cout << "Hello from MyFunction!\n";
}
```

```cpp
// header file: better.h
void MyFunction();
```

```cpp
// source file: better.cc
#include "better.h"

#include <iostream>

void MyFunction() {
  std::cout << "Hello from MyFunction!\n";
}
```

In C++, the extern keyword is not required when declaring functions.
All function declarations have external linkage by default.

# Immutable intentions



```cpp
class Point {
 public:
  Point() = default;
  Point(int x, int y) : x(x), y(y) {}

  int GetX() { return x; }
  int GetY() { return y; }

  void Print() {
    std::cout << "(" << x << ", " << y << ")\n";
  }

 private:
  int x = 0;
  int y = 0;
};
```

```cpp
int main() {
  Point p{20, 30};
  p.Print();
  return 0;
}
```

```cpp
int main() {
  const Point p{20, 30};
  p.Print();
  return 0;
}
```

```
error: 'this' argument to member function 'Print' has type 'const Point',
but function is not marked const
   p.Print();
   ^
note: 'Print' declared here
   void Print() { std::cout << "(" << x << ", " << y << ")\n"; }
        ^
```

# Make member functions const as appropriate



```cpp
class Point {
 public:
  Point() = default;
  Point(int x, int y) : x(x), y(y) {}

  int GetX() { return x; }
  int GetY() { return y; }

  void Print() {
    std::cout << "(" << x << ", " << y << ")\n";
  }

 private:
  int x = 0;
  int y = 0;
};
```

```cpp
int main() {
  const Point p{20, 30};
  p.Print();
  return 0;
}
```

```cpp
class Point {
 public:
  Point() = default;
  Point(int x, int y) : x(x), y(y) {}

  int GetX() const { return x; }
  int GetY() const { return y; }

  void Print() const {
    std::cout << "(" << x << ", " << y << ")\n";
  }

 private:
  int x = 0;
  int y = 0;
};
```

```cpp
int main() {
  const Point p{20, 30};
  p.Print();
  return 0;
}
```

# Make member functions const as appropriate



```cpp
class Point {
 public:
  Point() = default;
  Point(int x, int y) : x(x), y(y) {}

  int GetX() { return x; }
  int GetY() { return y; }

  void Print() { std::cout << "(" << x << ", " << y << ")\n"; }

 private:
  int x = 0;
  int y = 0;
};
```

*Clang-tidy:*
***--checks=*** *readability-make-member-function-const*

```
warning: method 'GetX' can be made const [readability-make-member-function-const]
   int GetX() { return x; }
       ^
              const
warning: method 'GetY' can be made const [readability-make-member-function-const]
   int GetY() { return y; }
       ^
              const
method 'Print' can be made const [readability-make-member-function-const]
   void Print() { std::cout << "(" << x << ", " << y << ")\n"; }
        ^
                const
```

# Handle with reference

```cpp
class Book {
 public:
  Book(std::string title) : title_(std::move(title)) {}
  std::string GetTitle() const { return title_; }

 private:
  std::string title_;
};
```

```cpp
class Book {
 public:
  Book(std::string title) : title_(std::move(title)) {}
  const std::string& GetTitle() const { return title_; }

 private:
  std::string title_;
};
```

Returning a non-trivial member variable (like std::string) by value can be inefficient.
Prefer returning a const reference to that member.

# Returning non-trivial member object

```
struct B {
  const A& GetA() const { return a; }
  A a;
};
```

```
B GetB() {
  return {};
}
```

```
int main() {
  [[maybe_unused]] const auto& a = GetB().GetA();
  puts("===== After GetB().GetA() ======");
}
```

```
A()
~A()
===== After GetB().GetA() ======
```

\`a\` is already destroyed after this point.

*Compiling with GCC:*

```
In function 'int main()':
error: possibly dangling reference to a temporary [-Werror=dangling-reference]
   [[maybe_unused]] const auto& a = GetB().GetA();
                                ^
note: 'B' temporary created here
   [[maybe_unused]] const auto& a = GetB().GetA();
                                    ~~~~^~
```

```
struct B {
  const A& GetA() const& { return a; }
  A GetA() && { return std::move(a); }
  A a;
};
```

\`a\` is referring to a copy and hence is valid until the end of main.

```
int main() {
  [[maybe_unused]] const auto& a = GetB().GetA();
  puts("===== After GetB().GetA() ======");
}
```

```
A()
A(A&&)
~A()
===== After GetB().GetA() ======
~A()
```

# Class: Special functions

# Don’t panic while moving.

```
struct A final {
  A(int a) : a_(a) { printf("A(%d)\n", a_); }
  ~A() { puts("~A()"); }
  A(const A& rhs) : a_(rhs.a_) { printf("A(const A&): %d\n", a_); }
  A(A&& rhs) : a_(rhs.a_) { printf("A(A&&): %d\n", a_); }
  A& operator=(const A& rhs) {
    a_ = rhs.a_;
    printf("A& operator=(const A&): %d\n", a_);
    return *this;
  }
  A& operator=(A&& rhs) noexcept {
    a_ = rhs.a_;
    printf("A& operator=(A&&): %d\n", a_);
    return *this;
  }
  int a_ = 0;
};
```

```
int main() {
  std::vector<A> vec;
  // Don't consider `reserve` for the moment.
  for (int i = 0; i < 4; ++i) {
    vec.emplace_back(i);
  }
}
```

```
A(0)
A(1)
A(const A&): 0
~A()
A(2)
A(const A&): 1
A(const A&): 0
~A()
~A()
A(3)
~A()
~A()
~A()
~A()
```

As we see, even in presence of move constructor, the copy constructor gets called during resize.

This “non-trivial” object’s move constructor is “not” **noexcept**.

# Ensure that the move constructor is noexcept



```
struct A final {
  A(int a) : a_(a) { printf("A(%d)\n", a_); }
  ~A() { puts("~A()"); }
  A(const A& rhs) : a_(rhs.a_) { printf("A(const A&): %d\n", a_); }
  A(A&& rhs) noexcept : a_(rhs.a_) { printf("A(A&&): %d\n", a_); }
  A& operator=(const A& rhs) {
    a_ = rhs.a_;
    printf("A& operator=(const A&): %d\n", a_);
    return *this;
  }
  A& operator=(A&& rhs) noexcept {
    a_ = rhs.a_;
    printf("A& operator=(A&&): %d\n", a_);
    return *this;
  }
  int a_ = 0;
};
```

```
int main() {
  std::vector<A> vec;
  // Don't consider `reserve` for the moment.
  for (int i = 0; i < 4; ++i) {
    vec.emplace_back(i);
  }
}
```

```
A(0)
A(1)
A(A&&): 0
~A()
A(2)
A(A&&): 1
A(A&&): 0
~A()
~A()
A(3)
~A()
~A()
~A()
~A()
```

Previous result.

```
A(0)
A(1)
A(const A&): 0
~A()
A(2)
A(const A&): 1
A(const A&): 0
~A()
~A()
A(3)
~A()
~A()
~A()
~A()
```

Consider making the move constructor “noexcept”.

# Ensure that the move constructor is noexcept

Let's consider again the case without noexcept move constructor.

```
struct A final {
  A(int a) : a_(a) { printf("A(%d)\n", a_); }
  ~A() { puts("~A()"); }
  A(const A& rhs) : a_(rhs.a_) { printf("A(const A&): %d\n", a_); }
  A(A&& rhs) : a_(rhs.a_) { printf("A(A&&): %d\n", a_); }
  A& operator=(const A& rhs) {
    a_ = rhs.a_;
    printf("A& operator=(const A&): %d\n", a_);
    return *this;
  }
  A& operator=(A&& rhs) noexcept {
    a_ = rhs.a_;
    printf("A& operator=(A&&): %d\n", a_);
    return *this;
  }
  int a_ = 0;
};
```

```
int main() {}
```

When clang-tidy check is used: ***--checks=performance-noexcept-move-constructor***

```
warning: move constructors should be marked noexcept [performance-noexcept-move-constructor]
   A(A&& rhs) : a_(rhs.a_) { printf("A(A&&): %d\n", a_); }
   ^
               noexcept
```

# Don’t suppress the essentials

```
struct A {
  A() { puts("A()"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
int main() {
  B b;
  puts("=== Before move ===");
  [[maybe_unused]] B b2 = std::move(b);
  puts("=== After move ===");
  return 0;
}
```

```
struct B {
  B() { puts("B()"); }
  ~B() { puts("~B()"); }
  A a;
};
```

```
A()
B()
=== Before move ===
A(const A&)
=== After move ===
~B()
~A()
~B()
~A()
```

```
struct B {
  B() { puts("B()"); }
  A a;
};
```

```
A()
B()
=== Before move ===
A(A&&)
=== After move ===
~A()
~A()
```

Non-trivial destructor suppresses move operations.

Ensure that all special member functions of a class are defined appropriately.

```
struct B {
  B() { puts("B()"); }
  ~B() { puts("~B()"); };
  B(const B&) = default;
  B(B&&) noexcept = default;
  B& operator=(const B&) = default;
  B& operator=(B&&) noexcept = default;
  A a;
};
```

```
A()
B()
=== Before move ===
A(A&&)
=== After move ===
~B()
~A()
~B()
~A()
```

# Ensure appropriate special member functions

## Special Members

compiler implicitly declares

| user declares | default constructor | destructor | copy constructor | copy assignment | move constructor | move assignment |
|---|---|---|---|---|---|---|
| Nothing | defaulted | defaulted | defaulted | defaulted | defaulted | defaulted |
| Any constructor | not declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| default constructor | user declared | defaulted | defaulted | defaulted | defaulted | defaulted |
| destructor | defaulted | user declared | defaulted | defaulted | not declared | not declared |
| copy constructor | not declared | defaulted | user declared | defaulted | not declared | not declared |
| copy assignment | defaulted | defaulted | defaulted | user declared | not declared | not declared |
| move constructor | not declared | defaulted | deleted | deleted | user declared | not declared |
| move assignment | defaulted | defaulted | deleted | deleted | not declared | user declared |

[Howard Hinnant’s presentation (youtube)]    [Slides]

# Ensure appropriate special member functions



From Cpp Core guidelines:

- C.20: If you can avoid defining any default operations, do
- C.21: If you define or =delete any copy, move, or destructor function, define or =delete them all
- C.22: Make default operations consistent

# Less is more

```
struct A {
  std::string a;
  std::string b;
  int c = 0;
  bool d = false;

  bool operator==(const A& rhs) const {
    return a == rhs.a && b == rhs.b && c == rhs.c && d == rhs.d;
  }

  bool operator!=(const A& rhs) const { return !(*this == rhs); }
};
```

```
struct A {
  std::string a;
  std::string b;
  int c = 0;
  bool d = false;

  bool operator==(const A& rhs) const {
    return a == rhs.a && b == rhs.b && c == rhs.c && d == rhs.d;
  }
};
```

```
int main() {
  A a{"hello", "world", 10, false};
  A b{"hello", "world", 10, false};
  std::cout << std::boolalpha << (a != b) << '\n';  // false
}
```

With C++20, != is generated by the compiler from ==

So, it is not necessary to define != along with ==

```
struct A {
  std::string a;
  std::string b;
  int c = 0;
  bool d = false;

  bool operator==(const A&) const noexcept = default;
};
```

Since, for this class, all the member variables are being compared, consider using the “defaulted” version.

# The trident of truth



```cpp
class Point {
 public:
  Point(int x, int y) : x_(x), y_(y) {}

  bool operator==(const Point& other) const {
    return x_ == other.x_ && y_ == other.y_;
  }
  bool operator<(const Point& other) const {
    return std::tie(x_, y_) < std::tie(other.x_, other.y_);
  }
  bool operator<=(const Point& other) const { return !(*this > other); }
  bool operator>(const Point& other) const { return other < *this; }
  bool operator>=(const Point& other) const { return !(*this < other); }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

```cpp
class Point {
 public:
  Point(int x, int y) : x_(x), y_(y) {}
  auto operator<=>(const Point&) const noexcept = default;

 private:
  int x_ = 0;
  int y_ = 0;
};
```

```cpp
int main() {
  const Point p1(1, 2);
  const Point p2(3, 4);

  // Use the comparison operators.
  std::ignore = (p1 == p2);
  std::ignore = (p1 != p2);
  std::ignore = (p1 < p2);
  std::ignore = (p1 <= p2);
  std::ignore = (p1 > p2);
  std::ignore = (p1 >= p2);
}
```

With C++20, <=> helps generate all the other operators.

Since, for this class, all the member variables are being compared, consider using the “defaulted” version.

# Selective trust wins

```cpp
class Bar;

class Foo {
 public:
  int GetSecret() const { return secret_; }

 private:
  friend class Bar;

  void SetSecret(int value) { secret_ = value; }

  int secret_ = 0;
};
```

```cpp
class Bar {
 public:
  void ChangeFooSecret(Foo& foo, int value) {
    foo.SetSecret(value);  // Allowed due to friendship.
  }
};
```

```cpp
int main() {
  Foo foo;
  Bar bar;
  bar.ChangeFooSecret(foo, 42);
  std::cout << foo.GetSecret() << '\n';
}
```

```
42
```

```cpp
class Bar {
 public:
  void ChangeFooSecret(Foo& foo, int value) {
    foo.SetSecret(value);  // Allowed due to friendship.
    // Also allowed due to friendship.
    foo.secret_ = 10;
  }
};
```

```
10
```

# Selective trust wins

```cpp
class Bar;

class Foo {
 public:
  int GetSecret() const { return secret_; }

 private:
  friend class Bar;

  void SetSecret(int value) { secret_ = value; }

  int secret_ = 0;
};
```

```cpp
class Foo {
 public:
  class PassKey {
    friend class Bar;
    PassKey() = default;
  };

  void SetSecret(int value, PassKey) { secret_ = value; }
  int GetSecret() const { return secret_; }

 private:
  int secret_ = 0;
};
```

```cpp
class Bar {
 public:
  void ChangeFooSecret(Foo& foo, int value) {
    foo.SetSecret(value);  // Allowed due to friendship.
    // Also allowed due to friendship.
    foo.secret_ = 10;
  }
};
```

```cpp
class Bar {
 public:
  void ChangeFooSecret(Foo& foo, int value) {
    foo.SetSecret(value, Foo::PassKey{});
    // This will NOT COMPILE!!
    // foo.secret_ = 0;
  }
};
```

Consider using PassKey pattern instead of making a class friend.

PassKey pattern allows us to ensure very targeted access for specific classes functions.

# Move / Forward

# The traveler needs a send-off



```
struct A {
  A() { puts("A()"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
class MyClass {
 public:
  MyClass(A&& a) : a_(a) {}

 private:
  A a_;
};
```

```
A()
A(const A&)
~A()
~A()
```

```
int main() {
  [[maybe_unused]] const MyClass m{A{}};
}
```

When clang-tidy check is used: ***--checks=**[cppcoreguidelines-rvalue-reference-param-not-moved]*

```
warning: rvalue reference parameter 'a' is never moved from inside the function body [cppcoreguidelines-rvalue-reference-param-not-moved]
   MyClass(A&& a) : a_(a) {}
               ^
```

```
class MyClass {
 public:
  MyClass(A&& a) : a_(std::move(a)) {}

 private:
  A a_;
};
```

```
A()
A(A&&)
~A()
~A()
```

Consider using std::move for rvalues inside functions to take ownership when applicable.

# Pass it as it came

```
class MyClass {
 public:
  MyClass(const char* str) : s_(str) { puts("MyClass(const char*)"); }
  MyClass(const std::string& str) : s_(str) {
    puts("MyClass(const std::string&)");
  }
  MyClass(std::string&& str) : s_(std::move(str)) {
    puts("MyClass(std::string&&)");
  }
 private:
  std::string s_;
};
```

This constructor is **not** getting called.

```
template <typename T, typename... Args>
std::unique_ptr<T> Create(Args&&... args) {
  return std::make_unique<T>(args...);
}
```

```
int main() {
  // const char[6].
  std::ignore = Create<MyClass>("hello");
  // prvalue: std::string&&.
  std::ignore = Create<MyClass>(std::string{"hello"});
  std::string s{"hello"};
  // lvalue: std::string&.
  std::ignore = Create<MyClass>(s);
  // xvalue: std::string&&.
  std::ignore = Create<MyClass>(std::move(s));
}
```

*MyClass(const char*)*
*MyClass(const std::string&)*
*MyClass(const std::string&)*
*MyClass(const std::string&)*

# Pass it as it came

```
template <typename T, typename... Args>
std::unique_ptr<T> Create(Args&&... args) {
  return std::make_unique<T>(args...);
}
```

```
MyClass(const char*)
MyClass(const std::string&)
MyClass(const std::string&)
MyClass(const std::string&)
```

```
template <typename T, typename... Args>
std::unique_ptr<T> Create(Args&&... args) {
  return std::make_unique<T>(std::forward<Args>(args)...);
}
```

```
MyClass(const char*)
MyClass(std::string&&)
MyClass(const std::string&)
MyClass(std::string&&)
```

```
int main() {
  // const char[6].
  std::ignore = Create<MyClass>("hello");
  // prvalue: std::string&&.
  std::ignore = Create<MyClass>(std::string{"hello"});
  std::string s{"hello"};
  // lvalue: std::string&.
  std::ignore = Create<MyClass>(s);
  // xvalue: std::string&&.
  std::ignore = Create<MyClass>(std::move(s));
}
```

Use std::forward when dealing with forwarding references to avoid unnecessary copies.

# Use std::forward for universal references

```
class MyClass {
 public:
  MyClass(const char* str) : s_(str) { puts("MyClass(const char*)"); }
  MyClass(const std::string& str) : s_(str) {
    puts("MyClass(const std::string&)");
  }
  MyClass(std::string&& str) : s_(std::move(str)) {
    puts("MyClass(std::string&&)");
  }

 private:
  std::string s_;
};
```

```
template <typename T, typename... Args>
std::unique_ptr<T> Create(Args&&... args) {
  return std::make_unique<T>(args...);
}
```

When clang-tidy check is used: ***--checks=[cppcoreguidelines-missing-std-forward]()***

```
warning: forwarding reference parameter 'args' is never forwarded inside the function body
[cppcoreguidelines-missing-std-forward]
 std::unique_ptr<T> Create(Args&&... args) {
                                     ^
```

# Don’t drain the well repeatedly

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
             Args&&... args) {
  for (const auto& func : funcs) {
    func(std::forward<Args>(args)...);
  }
}
```

```
void Func1(std::string s) {
  printf("Func1(%s)\n", s.c_str());
}

void Func2(std::string s) {
  printf("Func2(%s)\n", s.c_str());
}
```

```
int main() {
  std::vector<std::function<void(std::string)>> funcs{Func1, Func2};
  CallAll(funcs, std::string{"hello world"});
}
```

```
Func1(hello world)
Func2()
```

In calling ***Func1***, ***std::string::move*** was called so ***Func2*** is printing empty string which is incorrect.

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
             Args&&... args) {
  for (const auto& func : funcs) {
    func(args...);
  }
}
```

```
Func1(hello world)
Func2(hello world)
```

This version makes copies of strings.

If we are calling multiple functions using variadic arguments, then consider *removing* std::forward.

# Don't use std::move / std::forward for callbacks in a loop



```
template <typename... Args>
void CallAll(const std::vector<std::function<void(const Args&...)>>& funcs,
             Args&&... args) {
  for (const auto& func : funcs) {
    func(args...);
  }
}
```

```
void Func1(const std::string& s) {
  printf("Func1(%s)\n", s.c_str());
}

void Func2(const std::string& s) {
  printf("Func2(%s)\n", s.c_str());
}
```

```
int main() {
  std::vector<std::function<void(const std::string&)>> funcs{Func1, Func2};
  CallAll(funcs, std::string{"hello world"});
}
```

```
Func1(hello world)
Func2(hello world)
```

This approach removes the copies.

# Don't use std::move / std::forward for callbacks in a loop

```
template <typename... Args>
void CallAll(const std::vector<std::function<void(Args...)>>& funcs,
             Args&&... args) {
  for (const auto& func : funcs) {
    func(std::forward<Args>(args)...);
  }
}
```

When clang-tidy check is used: ***--checks=bugprone-use-after-move***

```
warning: 'args' used after it was forwarded [bugprone-use-after-move]
     func(std::forward<Args>(args)...);
                             ^
note: forward occurred here
     func(std::forward<Args>(args)...);
          ^
note: the use happens in a later loop iteration than the forward
     func(std::forward<Args>(args)...);
                             ^
```

# Standard Template Library (STL) Containers

# Avoid the detour

```
struct A {
  A(int) { puts("A(int)"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
int main() {
  constexpr int kTestSize = 2;
  std::vector<A> vec;
  vec.reserve(kTestSize);

  for (int i = 0; i < kTestSize; ++i) {
   vec.push_back(i);
  }
}
```

```
A(int)
A(A&&)
~A()
A(int)
A(A&&)
~A()
~A()
~A()
```

When clang-tidy check is used:
***--checks=modernize-use-emplace***

```
Warning: use emplace_back instead of push_back [modernize-use-emplace]
     vec.push_back(i);
         ^~~~~~~~~~
         emplace_back(
```

**Use emplace_back instead of push_back**

```
int main() {
  constexpr int kTestSize = 2;
  std::vector<A> vec;
  vec.reserve(kTestSize);

  for (int i = 0; i < kTestSize; ++i) {
   vec.emplace_back(i);
  }
}
```

```
A(int)
A(int)
~A()
~A()
```

# Use emplace* instead of other variations

- **std::deque**:

  `push_back/push_front` → `emplace_back/emplace_front`

- **std::forward_list**:

  `insert_after/push_front` → `emplace_after/emplace_front`

- **std::list**:

  `push_back/push_front/insert` → `emplace_back/emplace_front/emplace`

- **std::stack/std::queue**:

  `push` → `emplace`

- **std::set**:

  `insert` → `emplace`

# To learn more about unnecessary objects...



Wednesday, Sept 17
15:15 MDT
Stage 2

# Ask the right question

```cpp
void Check(const std::vector<int>& v) {
  if (v.size() == 0u) {
    // Do something.
  }
}
```

```cpp
void Check(const std::vector<int>& v) {
  if (v.empty()) {
    // Do something.
  }
}
```

empty() is more readable and clearly expresses intent.

*array, deque, forward_list, iostream, list, map, queue, set, span, stack, string, string_view, unordered_map, unordered_set, vector* – All these have both size() and empty().

*string* – Also has length().

# Replace size() == 0 with empty()

```
void Check(const std::vector<int>& v) {
  if (v.size() == 0u) {
    // Do something.
  }
}
```

When clang-tidy check is used: ***--checks=readability-container-size-empty***

```
warning: the 'empty' method should be used to check for emptiness instead of 'size' [readability-container-size-empty]
   if (v.size() == 0u) {
       ^~~~~~~~~~~~~~
       v.empty()
```

This catches uses of such code for all the following STL classes: *array, deque, forward_list, iostream, list, map, queue, set, span, stack, string, string_view, unordered_map, unordered_set, vector.*

# STL Algorithms

# The range revolution

```cpp
void PrintSorted(std::vector<int> v) {
  std::sort(v.begin(), v.end());
  for (int x : v) {
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
```

```cpp
void PrintSorted(std::vector<int> v) {
  std::ranges::sort(v);
  for (int x : v) {
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
```

Using range algorithms lead to more concise and readable code and reduces chances of mistakes.

# Use range algorithms

```
void PrintSorted(std::vector<int> v) {
  std::sort(v.begin(), v.end());
  for (int x : v) {
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
```

When clang-tidy check is used: ***--checks=[modernize-use-ranges]()***

```
warning: use a ranges version of this algorithm [modernize-use-ranges]
   std::sort(v.begin(), v.end());
   ^~~~~~~~~ ~~~~~~~~~  ~~~~~~~
   std::ranges::sort v
```

This catches uses of such code for all many algorithms.

*std::adjacent_find, std::all_of, std::any_of, std::binary_search, std::copy_backward, std::copy_if, std::copy, std::destroy, std::equal_range, std::equal, std::fill, std::find_end, std::find_if_not, std::find_if, std::find, std::for_each, std::generate, std::includes, std::inplace_merge, std::iota, std::is_heap_until, std::is_heap, std::is_partitioned, std::is_permutation, std::is_sorted_until, std::is_sorted, std::lexicographical_compare, std::lower_bound, std::make_heap, std::max_element, std::merge, std::min_element, std::minmax_element, std::mismatch, std::move_backward, std::move, std::next_permutation, std::none_of, std::partial_sort_copy, std::partition_copy, std::partition_point, std::partition, std::pop_heap, std::prev_permutation, std::push_heap, std::remove_copy_if, std::remove_copy, std::remove, std::remove_if, std::replace_if, std::replace, std::reverse_copy, std::reverse, std::rotate, std::rotate_copy, std::sample, std::search, std::set_difference, std::set_intersection, std::set_symmetric_difference, std::set_union, std::shift_left, std::shift_right, std::sort_heap, std::sort, std::stable_partition, std::stable_sort, std::transform, std::uninitialized_copy, std::uninitialized_default_construct, std::uninitialized_fill, std::uninitialized_move, std::uninitialized_value_construct, std::unique_copy, std::unique, std::upper_bound.*

# Erase the drama

```cpp
void RemoveOdd(std::vector<int>& v) {
  v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 != 0; }));
}

void RemoveNumber(std::vector<int>& v, int number) {
  v.erase(std::remove(v.begin(), v.end(), number));
}
```

```cpp
int main() {
  std::vector<int> v{1, 2, 3, 4, 5};
  RemoveOdd(v);
  RemoveNumber(v, 2);
  for (int x : v) {
    std::cout << x << ' ';
  }
  std::cout << '\n';
}
```

*4 4 5*

```cpp
void RemoveOdd(std::vector<int>& v) {
  v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 != 0; }),
          v.end());
}

void RemoveNumber(std::vector<int>& v, int number) {
  v.erase(std::remove(v.begin(), v.end(), number), v.end());
}
```

*4*

```cpp
void RemoveOdd(std::vector<int>& v) {
  std::erase_if(v, [](int x) { return x % 2 != 0; });
}

void RemoveNumber(std::vector<int>& v, int number) {
  std::erase(v, number);
}
```

*4*

Use std::erase and std::erase_if instead of erase / remove idiom.

# Use erase_if instead of erase / remove

```
void RemoveOdd(std::vector<int>& v) {
  v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 != 0; }));
}

void RemoveNumber(std::vector<int>& v, int number) {
  v.erase(std::remove(v.begin(), v.end(), number));
}
```

When clang-tidy check is used: ***--checks=bugprone-inaccurate-erase***

```
warning: this call will remove at most one item even when multiple items should be removed [bugprone-inaccurate-erase]
   v.erase(std::remove_if(v.begin(), v.end(), [](int x) { return x % 2 != 0; }));
   ^
                                                                                  , v.end()
warning: this call will remove at most one item even when multiple items should be removed [bugprone-inaccurate-erase]
   v.erase(std::remove(v.begin(), v.end(), number));
   ^
                                                   , v.end()
```

# Containment clarity



```
void PrintIfContains(const std::set<int>& s, int value) {
  if (std::find(s.begin(), s.end(), value) != s.end()) {
    std::cout << "Found\n";
  }
}
```

```
void PrintIfContains2(const std::set<int>& s, int value) {
  if (std::ranges::find(s, value) != s.end()) {
    std::cout << "Found\n";
  }
}
```

```
void PrintIfContains(const std::set<int>& s, int value) {
  if (s.contains(value)) {
    std::cout << "Found\n";
  }
}
```

```
void PrintIfContains3(const std::set<int>& s, int value) {
  if (s.find(value) != s.end()) {
    std::cout << "Found\n";
  }
}
```

Consider using “contains” member function instead of “find” for containers when applicable. It simplifies the code.

```
int main() {
  std::set<int> s{1, 2, 3};
  PrintIfContains(s, 2);
  PrintIfContains2(s, 2);
  PrintIfContains3(s, 2);
}
```

*Found*
*Found*
*Found*

- **contains** is present in all associative containers (**std::set, std::map**, etc.).
- C++23 added **contains** in **std::string, std::string_view, std::flat_set, std::flat_map**, etc.

# Use contains instead of find

```
void PrintIfContains3(const std::set<int>& s, int value) {
  if (s.find(value) != s.end()) {
    std::cout << "Found\n";
  }
}
```

When clang-tidy check is used: ***--checks=[readability-container-contains]***

```
warning: use 'contains' to check for membership [readability-container-contains]
   if (s.find(value) != s.end()) {
         ^~~~         ~~~~~~~~~~
         contains
```

This check also catches usage of “count” function and recommends replacing with “contains”.

```
std::set<int> s{1, 2, 3};
if (s.count(3) > 0) {
  std::cout << "Found\n";
}
```

→

```
std::set<int> s{1, 2, 3};
if (s.contains(3)) {
  std::cout << "Found\n";
}
```

# Custom isn't always clever

```cpp
bool HasInterestingNumber(std::span<const int> sp) {
  for (const auto i : sp) {
    if (i % 3 == 0) {
      return true;
    }
  }
  return false;
}
```

Consider using existing algorithms instead of hand rolled loops when possible.

```cpp
bool HasInterestingNumber(std::span<const int> sp) {
  return std::ranges::any_of(sp, [](const auto i) { return i % 3 == 0; });
}
```

```
HasInterestingNumber(std::__1::span<int const, 18446744073709551615ul>):
        test    rsi, rsi
        je      .LBB0_1
        lea     rcx, [4*rsi - 4]
        xor     edx, edx
.LBB0_3:
        imul    eax, dword ptr [rdi + rdx], -1431655765
        add     eax, 715827882
        cmp     eax, 1431655765
        setb    al
        jb      .LBB0_5
        lea     rsi, [rdx + 4]
        cmp     rcx, rdx
        mov     rdx, rsi
        jne     .LBB0_3
.LBB0_5:
        ret
.LBB0_1:
        xor     eax, eax
        ret
```

*Clang -O3 output*

Both versions generate the same code

# Use existing algorithms

```
// Next, check if the panel has moved to the other side of another panel.

for (size_t i = 0; i < expanded_panels_.size(); ++i) {
  Panel* panel = expanded_panels_[i].get();
  if (center_x <= panel->cur_panel_center() ||
      i == expanded_panels_.size() - 1) {
    if (panel != fixed_panel) {
      // If it has, then we reorder the panels.
      ref_ptr<Panel> ref = expanded_panels_[fixed_index];
      expanded_panels_.erase(expanded_panels_.begin() + fixed_index);
      if (i < expanded_panels_.size()) {
        expanded_panels_.insert(expanded_panels_.begin() + i, ref);
      } else {
        expanded_panels_.push_back(ref);
      }
    }
    break;
  }
}
```

Sean Parent's GoingNative 2013 C++ Seasoning talk (Slides)

```
// Next, check if the panel has moved to the left side of another panel.

auto f = begin(expanded_panels_) + fixed_index;

auto p = lower_bound(begin(expanded_panels_), f, center_x,
  [](const ref_ptr<Panel>& e, int x){ return e->cur_panel_center() < x; });

// If it has, then we reorder the panels.
rotate(p, f, f + 1);
```

# Miscellaneous

# Build where you stand

```
int main() {
  const std::pair<A, A> pa{{1, 1}, {2, 2}};
}
```

```
A(int, int)
A(int, int)
A(const A&)
A(const A&)
~A()
~A()
~A()
~A()
```

```
int main() {
  const std::pair<A, A> pa{std::piecewise_construct,
          std::forward_as_tuple(1, 1),
          std::forward_as_tuple(2, 2)};
}
```

```
A(int, int)
A(int, int)
~A()
~A()
```

```
int main() {
  const std::optional<A> oa = A{10, 10};
}
```

```
A(int, int)
A(A&&)
~A()
~A()
```

```
int main() {
  const auto oa = std::make_optional<A>(10, 10);
}
```

```
A(int, int)
~A()
```

```
std::expected<A, bool> Foo() {
  return A{10, 10};
}

int main() {
  std::ignore = Foo();
}
```

```
A(int, int)
A(A&&)
~A()
~A()
```

```
std::expected<A, bool> Foo() {
  return std::expected<A, bool>{std::in_place, 10, 10};
}

int main() {
  std::ignore = Foo();
}
```

```
A(int, int)
~A()
```

```
int main() {
  std::variant<A, int> v{A{10, 10}};
}
```

```
A(int, int)
A(A&&)
~A()
~A()
```

```
int main() {
  std::variant<A, int> v{std::in_place_type<A>, 10, 10};
}
```

```
A(int, int)
~A()
```

Use in-place constructors for various STL types.

# A unified identity

```
class MyClass final {
 public:
  MyClass(int v) : int_value_(v) {}
  MyClass(std::string&& s) : str_value_(std::move(s)) {}

  bool HasIntValue() const { return !!int_value_; }
  bool HasStringValue() const { return !!str_value_; }

  const int* GetInt() const;
  const std::string* GetStr() const;

 private:
  std::optional<int> int_value_;
  std::optional<std::string> str_value_;
};

const int* MyClass::GetInt() const {
  return int_value_ ? &*int_value_ : nullptr;
}

const std::string* MyClass::GetStr() const {
  return str_value_ ? &*str_value_ : nullptr;
}
```

```
class MyClass final {
 public:
  MyClass(int v) : value_(v) {}
  MyClass(std::string&& s) : value_(std::move(s)) {}

  bool HasIntValue() const;
  bool HasStringValue() const;

  const int* GetInt() const;
  const std::string* GetStr() const;

 private:
  std::variant<int, std::string> value_;
};

bool MyClass::HasIntValue() const {
  return std::holds_alternative<int>(value_);
}

bool MyClass::HasStringValue() const {
  return std::holds_alternative<std::string>(value_);
}

const int* MyClass::GetInt() const {
  return std::get_if<int>(&value_);
}

const std::string* MyClass::GetStr() const {
  return std::get_if<std::string>(&value_);
}
```

Avoid juggling multiple variables for type alternatives - use std::variant to unify them under a single, type-safe container.

Replace C++ unions with std::variant where possible to improve safety and maintainability.

# A placeholder, not a pretender

```cpp
struct Visitor {
  void operator()(int v) const {
    if (v != 0) {  // 0 is "unset".
      std::cout << "int: " << v << '\n';
    }
  }
  void operator()(const std::string& s) const {
    std::cout << "str: " << s << '\n';
  }
};
```

```cpp
struct Visitor {
  void operator()(int v) const {
    std::cout << "int: " << v << '\n';
  }
  void operator()(const std::string& s) const {
    std::cout << "str: " << s << '\n';
  }
  void operator()(std::monostate) const { std::cout << "Unset\n"; }
};
```

```cpp
template <typename T>
  requires IsVariantV<T>
void TestVariant(const T& v) {
  std::visit(Visitor{}, v);
}
```

```cpp
int main() {
  std::variant<int, std::string> v;
  TestVariant(v);
  v.emplace<std::string>("hello");
  TestVariant(v);
}
```

```
str: hello
```

```cpp
int main() {
  std::variant<std::monostate, int, std::string> v;
  TestVariant(v);
  v.emplace<std::string>("hello");
  TestVariant(v);
}
```

```
Unset
str: hello
```

Consider using std::monostate to represent “unset” state in std::variant.

# Use std::monostate for std::variant

```
struct NoDefaultConstructor {
  explicit NoDefaultConstructor(int i) : value(i) {}
  int value;
};
```

```
struct Visitor {
  void operator()(const std::string& s) const {
    std::cout << "str: " << s << '\n';
  }
  void operator()(const NoDefaultConstructor& n) const {
    std::cout << "NoDefault: " << n.value << '\n';
  }
};
```

```
template <typename T>
  requires IsVariantV<T>
void TestVariant(const T& v) {
  std::visit(Visitor{}, v);
}
```

```
int main() {
  // Does not compile
  // std::variant<NoDefaultConstructor, std::string> v;
  std::variant<std::string, NoDefaultConstructor> v;
  TestVariant(v);
  v.emplace<NoDefaultConstructor>(10);
  TestVariant(v);
}
```

```
str:
NoDefault: 10
```

```
struct Visitor {
  void operator()(const std::string& s) const {
    std::cout << "str: " << s << '\n';
  }
  void operator()(const NoDefaultConstructor& n) const {
    std::cout << "NoDefault: " << n.value << '\n';
  }
  void operator()(std::monostate) const { std::cout << "Unset\n"; }
};
```

```
int main() {
  std::variant<std::monostate, NoDefaultConstructor, std::string> v;
  TestVariant(v);
  v.emplace<NoDefaultConstructor>(10);
  TestVariant(v);
}
```

```
Unset
NoDefault: 10
```

# Use std::monostate for std::variant



```
template <typename T>
  requires IsVariantV<T>
void TestVariant(const T& v) {
  std::visit(Visitor{}, v);
}
```

```
template <typename T>
struct IsVariant : std::false_type {};

template <typename... Args>
struct IsVariant<std::variant<Args...>> : std::true_type {};

template <typename T>
inline constexpr bool IsVariantV = IsVariant<T>::value;
```

# Don’t let it drift

```cpp
void Calculate(double radius) {
  double pi = 3.14159;
  double circumference = 2 * pi * radius;
  double area = pi * radius * radius;

  std::cout << "For a circle with radius " << radius << ":\n";
  std::cout << "Circumference: " << circumference << "\n";
  std::cout << "Area: " << area << '\n';
}
```

```cpp
void Calculate(double radius) {
  static constexpr double kPI = 3.14159;
  const double circumference = 2 * kPI * radius;
  const double area = kPI * radius * radius;

  std::cout << "For a circle with radius " << radius << ":\n";
  std::cout << "Circumference: " << circumference << "\n";
  std::cout << "Area: " << area << '\n';
}
```

```cpp
void ProcessData() {
  std::string file_path{"folder/filepath.text"};
  std::vector<int> int_data = GetIntData(file_path);
  std::string str_data = GetStringData(file_path);

  // Use `file_path`, `int_data`, `str_data` without modifying.
  std::cout << "File path: " << file_path << '\n';
  std::cout << "Integer data: ";
  for (int num : int_data) {
    std::cout << num << " ";
  }
  std::cout << "Sum: " << GetSomeValue(int_data) << '\n';
  std::cout << "String data: " << str_data << '\n';
  std::cout << GetInterestingNumber(str_data) << '\n';
}
```

```cpp
void ProcessData() {
  const std::string file_path{"folder/filepath.text"};
  const std::vector<int> int_data = GetIntData(file_path);
  const std::string str_data = GetStringData(file_path);

  // Use `file_path`, `int_data`, `str_data` without modifying.
  std::cout << "File path: " << file_path << '\n';
  std::cout << "Integer data: ";
  for (int num : int_data) {
    std::cout << num << " ";
  }
  std::cout << "Sum: " << GetSomeValue(int_data) << '\n';
  std::cout << "String data: " << str_data << '\n';
  std::cout << GetInterestingNumber(str_data) << '\n';
}
```

Consider making local variables which are not modified after initialization const to help with readability and comprehension.

# Snapshot on arrival

```cpp
void Foo(bool show, int val) {
  int final_value = 0;
  if (show) {
    if (val == 1) {
      final_value = 3;
    } else if (val <= 100) {
      final_value = 4;
    } else {
      final_value = 5;
    }
  } else {
    if (val == 1) {
      final_value = 6;
    } else {
      final_value = 7;
    }
  }
  // Use `final_value`.
  std::cout << final_value << '\n';
}
```

```cpp
void Foo(bool show, int val) {
  const int final_value = [&]() {
    if (show) {
      if (val == 1) {
        return 3;
      } else if (val <= 100) {
        return 4;
      }
      return 5;
    }
    if (val == 1) {
      return 6;
    }
    return 7;
  }();
  // Use `final_value`.
  std::cout << final_value << '\n';
}
```

Consider using immediately invoked lambda to create ‘const’ variables when applicable.

# One definition to rule them all

```
// header.h
constexpr double kPi = 3.141592653589793;

const double* GetAddressOfPi();
```

```
// another.cc
#include "header.h"

const double* GetAddressOfPi() {
  return &kPi;
}
```

```
main.cc: &kPi: 0x7ff6084a4000
another.cc: &kPi: 0x7ff6084a4058
```

```
// main.cc
#include "header.h"

namespace {
void PrintPiAddress() {
  std::cout << "main.cc: &kPi: " << (void*)&kPi << '\n';
}
}  // namespace

int main() {
  PrintPiAddress();
  std::cout << "another.cc: &kPi: " << (void*)GetAddressOfPi() << '\n';
}
```

This is One Definition Rule violation – undefined behavior

```
// header.h
inline constexpr double kPi = 3.141592653589793;

const double* GetAddressOfPi();
```

```
main.cc: &kPi: 0x7ff679454050
another.cc: &kPi: 0x7ff679454050
```

Add inline keyword to header-only definitions to ensure that one definition rule is not violated.

# Final takeaways

- **Scope Smartly**: Minimize variable scope in conditionals, loops, and switches to boost clarity and reduce bugs.
- **Iterate Elegantly**: Prefer range-based loops, structured bindings, and const& or auto&& for clean and efficient traversal.
- **Design Thoughtfully**: Use enum class, std::variant, std::optional, and std::expected to express intent and avoid ambiguity.
- **Create Classes with Care**: Initialize members early, make functions const or static when appropriate, and use idioms like PassKey and <=> for control and comparison.
- **Modernize with STL**: Embrace std::ranges, erase_if, contains, and emplace_* to replace verbose patterns with expressive, standard solutions.
- Use clang-tidy checks and warnings to help the compiler catch issues.

# Further Resources

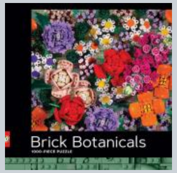
- Many more scenarios present in the appendix.
- Code snippets are present in github.

# Microsoft @ CppCon

| Monday, Sept 15 | Tuesday, Sept 16 | Wednesday, Sept 17 | Thursday, Sept 18 | Friday, Sept 19 |
|---|---|---|---|---|
| **Building Secure Applications: A Practical End-to-End Approach**<br>Chandranath Bhattacharyya & Bharat Kumar<br>16:45 – 17:45 | **What's New for Visual Studio Code: Cmake Improvements and GitHub Copilot Agents**<br>Alexandra Kemper<br>09:00 – 10:00 | **LLMs in the Trenches: Boosting System Programming with AI**<br>Ion Todirel<br>14:00 – 15:00 | **MSVC C++ Dynamic Debugging: How We Enabled Full Debuggability of Optimized Code**<br>Eric Brumer<br>14:00 – 14:30 | **Reflection-based JSON in C++ at Gigabytes per Second**<br>Daniel Lemire & Francisco Geiman Thiesen<br>09:00 – 10:00 |
| | **What's new in Visual Studio for C++ Developers in 2025**<br>Augustin Popa & David Li<br>14:00 – 15:00 | **C++ Performance Tips: Cutting Down on Unnecessary Objects**<br>Kathleen Baker & Prithvi Okade<br>15:15 – 16:15 | **It's Dangerous to Go Alone: A Game Developer Tutorial**<br>Michael Price<br>16:45 – 17:45 | **Duck-Tape Chronicles: Rust/C++ Interop**<br>Victor Ciura<br>13:30 – 14:30 |
| | **Back to Basics: Code Review**<br>Chandranath Bhattacharyya & Kathleen Baker<br>14:00 – 15:00 | **Connecting C++ Tools to AI Agents Using the Model Context Protocol**<br>Ben McMorran<br>15:50 – 16:20 | | |
| | | **Welcome to v1.0 of the meta::[[verse]]!**<br>Inbal Levi<br>16:45 – 17:45 | | |

Take our survey, win prizes

https://aka.ms/cppcon/review




# Questions?

Github link for code snippets:

https://aka.ms/cppcon/review

# Appendix

# Containers: Beware of copies in std::initializer_list.



```
int main() {
  std::vector<A> v{{1, 1}, {2, 2}, {3, 3}};
}
```

```
A(int, int)
A(int, int)
A(int, int)
A(const A&)
A(const A&)
A(const A&)
~A()
~A()
~A()
~A()
~A()
~A()
```

Three objects are created and then **copied** into the vector

```
int main() {
  constexpr int kTestSize = 3;
  std::vector<A> v;
  v.reserve(kTestSize);
  for (int i = 0; i < kTestSize; ++i) {
    v.emplace_back(i, i);
  }
}
```

```
A(int, int)
A(int, int)
A(int, int)
~A()
~A()
~A()
```

***reserve* / *emplace_back*** does in-place construction.

# Clarity: Use [[maybe_unused]] instead of (void).

```
std::expected<Data, int> LoadData() {
  // Simulate a failure for demonstration.
  return std::unexpected(42);  // Example error code
}
```

```
void ProcessData(const Data& data, bool force_reload) {
  // `force_reload` is not used in this specific implementation.
  // ... process data ...
  data.DumpData();
}
```

```
void HandleData() {
  auto result = LoadData();
  if (result) {
    ProcessData(*result, false);
  } else {
    const int error_code = result.error();
#ifndef NDEBUG
    std::cerr << "Error loading data: " << error_code << '\n';
#endif
  }
}
```

```
error: unused parameter 'force_reload' [-Werror,-Wunused-parameter]
 void ProcessData(const Data& data, bool force_reload) {
                                         ^
error: unused variable 'error_code' [-Werror,-Wunused-variable]
     const int error_code = result.error();
               ^~~~~~~~~~
```

```
-Wunused-variable (-Wall)
-Wunused-parameter (-Wextra)
```

# Clarity: Use [[maybe_unused]] instead of (void).



```
std::expected<Data, int> LoadData() {
  // Simulate a failure for demonstration.
  return std::unexpected(42);  // Example error code
}
```

```
void ProcessData(const Data& data, bool force_reload) {
  (void)force_reload;  // Unused in this specific implementation.
  // ... process data ...
  data.DumpData();
}
```

```
void HandleData() {
  auto result = LoadData();
  if (result) {
    ProcessData(*result, false);
  } else {
    const int error_code = result.error();
#ifndef NDEBUG
    std::cerr << "Error loading data: " << error_code << '\n';
#endif
    (void)error_code;
  }
}
```

```
void ProcessData(const Data& data, [[maybe_unused]] bool force_reload) {
  // ... process data ...
  data.DumpData();
}
```

```
void HandleData() {
  auto result = LoadData();
  if (result) {
    ProcessData(*result, false);
  } else {
    [[maybe_unused]] const int error_code = result.error();
#ifndef NDEBUG
    std::cerr << "Error loading data: " << error_code << '\n';
#endif
  }
}
```

Use [[maybe_unused]] instead of (void) when possible, to provide more readability.

# std::unique_ptr: Consider std::optional

```
// A.h
struct A {
  enum class WorkType { kType1, kType2 };
  void DoSomething(WorkType) {}
};
```

```
// B.h.
class B {
 public:
  // Will need to included A header to get `A::WorkType`.
  void DoSomething(A::WorkType type);

 private:
  std::unique_ptr<A> a_;
};

// B.cc
void B::DoSomething(A::WorkType type) {
  if (!a_) {
    a_ = std::make_unique<A>();
  }
  a_->DoSomething(type);
}
```

→

```
// B.h.
class B {
 public:
  // Will need to included A header to get `A::WorkType`.
  void DoSomething(A::WorkType type);

 private:
  std::optional<A> a_;
};

// B.cc
void B::DoSomething(A::WorkType type) {
  if (!a_) {
    a_.emplace();
  }
  a_->DoSomething(type);
}
```

# std::unique_ptr: Consider std::optional

```
// A.h
struct A {
  enum class WorkType { kType1, kType2 };
  void DoSomething(WorkType) {}
};
```

```
bool ShouldCreateUniquePtr(int param) {
  return param > 0;
}

bool ShouldCallDoSomething(int param) {
  return param % 2 == 0;
}

void Foo(int param) {
  std::unique_ptr<A> local_a;

  if (ShouldCreateUniquePtr(param)) {
    local_a = std::make_unique<A>();
  }
  // Do some other stuff.
  if (local_a && ShouldCallDoSomething(param)) {
    local_a->DoSomething(A::WorkType::kType1);
  }
}
```

```
bool ShouldCreateOptional(int param) {
  return param > 0;
}

bool ShouldCallDoSomething(int param) {
  return param % 2 == 0;
}

void Foo(int param) {
  std::optional<A> local_a;

  if (ShouldCreateOptional(param)) {
    local_a.emplace();
  }
  // Do some other stuff.
  if (local_a && ShouldCallDoSomething(param)) {
    local_a->DoSomething(A::WorkType::kType1);
  }
}
```

Consider using **std::optional** instead of **std::unique_ptr** for delayed creation if we are not dealing with polymorphic types. **std::optional** avoids heap allocation.

# Naming: Variables, constants, enums

```cpp
namespace MyNamespace {
enum class my_enum {
  A,
  B,
};
}  // namespace MyNamespace

namespace {
constexpr int some_Constant = 20;
}

class my_class {
 public:
  void foo(int Value);

 private:
  int _member;
};
```

```cpp
namespace my_namespace {
enum class MyEnum {
  kA,
  kB,
};
}  // namespace my_namespace

namespace {
constexpr int kSomeConstant = 20;
}

class MyClass {
 public:
  void Foo(int value);

 private:
  int member_;
};
```

# Namespace shortening



```
namespace my_product {
namespace features {
namespace my_feature {
// Stuff inside.
}  // namespace my_feature
}  // namespace features
}  // namespace my_product
```

```
namespace my_product::features::my_feature {
// Stuff inside.
}  // namespace my_product::features::my_feature
```

**clang-tidy check:** *modernize-concat-nested-namespaces*

# Copyright Header

```cpp
#ifndef COMPONENTS_MY_CLASS_H_
#define COMPONENTS_MY_CLASS_H_

class MyClass final {
 public:
  MyClass(int starting);

  int GetDouble() const;
  int GetAfterAdd(int i) const;

  // Some other functions.
 private:
  int starting_ = 0;
};

#endif  // COMPONENTS_MY_CLASS_H_
```

```cpp
//  Copyright (c) 2025 Your Name or Company. All rights reserved.

#ifndef COMPONENTS_MY_CLASS_H_
#define COMPONENTS_MY_CLASS_H_

class MyClass final {
 public:
  MyClass(int starting);

  int GetDouble() const;
  int GetAfterAdd(int i) const;

  // Some other functions.
 private:
  int starting_ = 0;
};

#endif  // COMPONENTS_MY_CLASS_H_
```

# Move variable closer to usage

```
struct SomeClass {
  // Other stuff.
  bool HasPropOne() const;
  // Other stuff.
};
SomeClass Foo();
```

```
int main() {
    const SomeClass s = Foo();
    // Code that doesn't use 's'.

    std::ignore = s.HasPropOne();
}
```

```
int main() {
    // Code that doesn't use 's'.

    const SomeClass s = Foo();
    std::ignore = s.HasPropOne();
}
```

Improves readability and ease for refactoring.

*std::ignore usage here compile fine with most major compilers but is blessed by standard in C++26 with [p2968](p2968).*

# Switch statement: Use [[fallthrough]]

```cpp
enum class LogLevel {
  kInfo,
  kWarning,
  kError
};
```

[[fallthrough]] indicates intentional fall through from the previous case label.

```cpp
void Log(LogLevel level, const std::string& log) {
  std::string log_message;
  switch (level) {
    case LogLevel::kError:
      log_message = "ERROR: ";
    case LogLevel::kWarning:
      log_message += "!";
    case LogLevel::kInfo:
      log_message += "[" + log + "]";
      break;
  }
  std::cout << log_message << '\n';
}
```

```cpp
void Log(LogLevel level, const std::string& log) {
  std::string log_message;
  switch (level) {
    case LogLevel::kError:
      log_message = "ERROR: ";
      [[fallthrough]];
    case LogLevel::kWarning:
      log_message += "!";
      [[fallthrough]];
    case LogLevel::kInfo:
      log_message += "[" + log + "]";
      break;
  }
  std::cout << log_message << '\n';
}
```

***-Wimplicit-fallthrough*** warns about unannotated fall throughs

# Use unnamed namespace instead of static

```
// non_optimal.cc
#include <iostream>

// clang-tidy check: misc-use-anonymous-namespace
// Clang-C++ warning: none
static void InternalFunction() {
  std::cout << "Internal function (static)\n";
}

static int internal_var = 42;
```

```
// better.cc
#include <iostream>

namespace {
void InternalFunction() {
  std::cout << "Internal function (unnamed namespace)\n";
}

int internal_var = 42;
}  // namespace
```

- Unnamed namespace reduces typing – Don't need to keep typing static multiple times for a group of variables and functions.
- It also allows defining "local" classes. There is no equivalent "static" class.

Some codebases have the opposite rule like LLVM: llvm-prefer-static-over-anonymous-namespace

# Parameter Passing: Use strong types

```
void RegisterUser(const std::string& name, const std::string& email) {
  std::cout << "Name: " << name << ", Email: " << email << '\n';
}
```

```
int main() {
  // Easy to swap arguments by mistake
  RegisterUser("Alice", "alice@example.com");
  RegisterUser("alice@example.com", "Alice");
}
```

```
Name: Alice, Email: alice@example.com
Name: alice@example.com, Email: Alice
```

Using strong types prevents passing in wrong arguments.

```
struct Name {
  std::string value;
};
struct Email {
  std::string value;
};

void RegisterUserBetter(Name name, Email email) {
  std::cout << "Name: " << name.value << ", Email: " << email.value << '\n';
}
```

```
int main() {
  RegisterUserBetter(
    Name{"Alice"},
    Email{"alice@example.com"});
}
```

# Function: Merge return path

```
int Baz(int some_var) {
  if (SomeConditionIsTrue()) {
    some_var += 1;
  }
  if (IsConditionOneTrue()) {
    some_var += 1;
    Foo(some_var);
    Bar(some_var);
    return some_var;
  }
  if (AnotherConditionIsTrue()) {
    some_var += 2;
  }
  if (IsConditionTwoTrue()) {
    some_var += 1;
    Foo(some_var);
    Bar(some_var);
    return some_var;
  }
  if (OneMoreConditionIsTrue()) {
    some_var += 3;
  }
  some_var += 1;
  Foo(some_var);
  Bar(some_var);
  return some_var;
}
```

```
int Baz(int some_var) {
  if (SomeConditionIsTrue()) {
    some_var += 1;
  }
  // Common exit code is grouped.
  auto exit_fn = MakeScopeExit([&] {
    some_var += 1;
    Foo(some_var);
    Bar(some_var);
  });
  if (IsConditionOneTrue()) {
    return some_var;
  }
  if (AnotherConditionIsTrue()) {
    some_var += 2;
  }
  if (IsConditionTwoTrue()) {
    return some_var;
  }
  if (OneMoreConditionIsTrue()) {
    some_var += 3;
  }
  return some_var;
}
```

Merge similar return path code into a common “exit” function to be automatically called on exit path.

# Function: Merge return path

```
int Baz(int some_var) {
  if (SomeConditionIsTrue()) {
    some_var += 1;
  }
  // Common exit code is grouped.
  auto exit_fn = MakeScopeExit([&] {
    some_var += 1;
    Foo(some_var);
    Bar(some_var);
  });
  if (IsConditionOneTrue()) {
    return some_var;
  }
  if (AnotherConditionIsTrue()) {
    some_var += 2;
  }
  if (IsConditionTwoTrue()) {
    return some_var;
  }
  if (OneMoreConditionIsTrue()) {
    some_var += 3;
  }
  return some_var;
}
```

```
template <std::invocable F>
class ScopeExit {
 public:
  constexpr ScopeExit(F&& f) : f_(std::move(f)) {}
  ScopeExit& operator=(ScopeExit&&) = delete;
  constexpr ~ScopeExit() {
    if (call_at_end_) {
      f_();
    }
  }
  constexpr void Dismiss() { call_at_end_ = false; }

 private:
  F f_;
  bool call_at_end_ = true;
};


template <std::invocable F>
constexpr auto MakeScopeExit(F&& f) {
  return ScopeExit<F>(std::forward<F>(f));
}
```

Used to ensure that “exit” function is not called in some cases.
Example: error cases call exit function, whereas return does not.

absl::Cleanup provides similar functionality.

# Use [[nodiscard]] when possible.



```cpp
struct A {
  A() { puts("A()"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```cpp
std::unique_ptr<A> CreateA() {
  return std::make_unique<A>();
}
```

```cpp
[[nodiscard]] std::unique_ptr<A> CreateA() {
  return std::make_unique<A>();
}
```

```cpp
int main() {
  CreateA();
  puts("==== After CreateA() ====");
}
```

```
warning: ignoring return value of function declared with
'nodiscard' attribute [-Wunused-result]
   CreateA();
   ^~~~~~~
```

```
A()
~A()
==== After CreateA() ====
```

```
A()
==== After CreateA() ====
~A()
```

# Use [[nodiscard]] when possible.

```
class ResourceWrapper {
 public:
  bool IsValid() const { return is_valid_; }

 private:
  bool is_valid_ = true;
};
```

```
void ProcessResource(const ResourceWrapper& resource) {
  resource.IsValid();

  // This might be incorrect if the resource is not valid.
  puts("Processing resource...\n");
```

**Clang-tidy check:** modernize-use-nodiscard

```
warning: function 'IsValid' should be marked [[nodiscard]] [modernize-use-nodiscard]
   bool IsValid() const { return is_valid_; }
   ^
   [[nodiscard]]
```

This clang-tidy check works only for certain type of member functions.

```
class ResourceWrapper {
 public:
  [[nodiscard]] bool IsValid() const { return is_valid_; }

 private:
  bool is_valid_ = true;
};
```

# Use [[nodiscard]] when possible.

```
class ErrorInfo {
  // Class stuff.
};
```

```
ErrorInfo GetError() {
  // Fill in the error with all info.
  return ErrorInfo{};
}
```

```
int main() {
  GetError();
}
```

```
class [[nodiscard]] ErrorInfo {
  // Class stuff.
};
```

```
error: ignoring return value of type 'ErrorInfo' declared with 'nodiscard' attribute [-Werror,-Wunused-value]
   GetError();
   ^~~~~~~~
```

# Use [[nodiscard]] when possible.

```
struct A {
  A() { puts("A()"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
std::unique_ptr<A> CreateA() {
  return std::make_unique<A>();
}
```

```
int main() {
  CreateA();
  puts("==== After CreateA() ====");
}
```

Clang-tidy check bugprone-unused-return-value can be used with *bugprone-unused-return-value.CheckedFunctions* and tag `CreateA` for it to start flagging this issue.

```
warning: the value returned by this function should not be disregarded; neglecting it may lead to errors [bugprone-unused-return-value]
   CreateA();
   ^~~~~~~~~
```

# Class: Ensure a class is initialized in its constructor.

```
class Person {
 public:
  Person() = default;

  void Init(const std::string& name, int age) {
    name_ = name;
    age_ = age;
  }

  [[nodiscard]] bool IsValid() const { return !name_.empty() && age_ > 0; }

 private:
  std::string name_;
  int age_ = 0;
};
```

```
int main() {
  Person p; // Creates invalid object.
  p.Init("John Doe", 42);
}
```

When possible, try to create object fully in the constructor.

```
class Person {
 public:
  Person(const std::string& name, int age) : name_(name), age_(age) {}

  [[nodiscard]] bool IsValid() const { return !name_.empty() && age_ > 0; }

 private:
  std::string name_;
  int age_ = 0;
};
```

```
int main() {
  // Valid object created.
  [[maybe_unused]] Person p("John Doe", 42);
}
```

# Class: Use template method to initialize

```
class DataProcessor {
 public:
  virtual ~DataProcessor() = default;
  virtual void ProcessData() = 0;
};

class DataProvider {
 public:
  virtual ~DataProvider() = default;
  virtual void SetProcessor(DataProcessor* processor) = 0;
};
```

```
class MockDataProvider : public DataProvider {
 public:
  void SetProcessor(DataProcessor* processor) override {
    // Store the processor for later use.
    processor_ = processor;
  }

 private:
  DataProcessor* processor_ = nullptr;
};
```

```
class DataHolder : public DataProcessor {
 public:
  DataHolder() = default;
  DataHolder& operator=(DataHolder&&) noexcept = delete;

  void Init(std::vector<int>&& data, DataProvider& provider) {
    data_ = std::move(data);
    // Needs a fully created object, so that virtual functions can be called.
    provider.SetProcessor(this);
    // Other stuff.
  }

 private:
  void ProcessData() override {
    // Process data here.
  }

  std::vector<int> data_;
};
```

```
int main() {
  DataHolder d;
  MockDataProvider provider;
  d.Init({1, 2, 3}, provider);
}
```

# Class: Use template method to initialize

```cpp
class DataHolder : public DataProcessor {
 public:
  DataHolder() = default;
  void Init(std::vector<int>&& data, DataProvider& provider) {
    data_ = std::move(data);
    // Needs a fully created object, so that virtual functions can be called.
    provider.SetProcessor(this);
    // Other stuff.
  }

 private:
  void ProcessData() override { // Process data here. }

  std::vector<int> data_;
};
```

```cpp
int main() {
  DataHolder d;
  MockDataProvider provider;
  d.Init({1, 2, 3}, provider);
}
```

```cpp
class DataHolder : public DataProcessor {
 public:
  static std::unique_ptr<DataHolder> Create(std::vector<int>&& data,
                                            DataProvider& provider) {
    std::unique_ptr<DataHolder> holder(new DataHolder(std::move(data)));
    provider.SetProcessor(holder.get());
    return holder;
  }
  DataHolder& operator=(DataHolder&&) noexcept = delete;

 private:
  DataHolder(std::vector<int>&& data) : data_(std::move(data)) {}

  void ProcessData() override { // Process data here. }

  std::vector<int> data_;
};
```

```cpp
int main() {
  MockDataProvider provider;
  [[maybe_unused]] const auto d =
     DataHolder::Create({1, 2, 3}, provider);
}
```

We always get a valid object. We needed to convert to std::unique_ptr.

# Class: Make base class constructors protected.

```
class Base {
 public:
  explicit Base(int x) : x_(x) {}
  int GetX() const { return x_; }

 private:
  const int x_;
};
```

```
class Derived : public Base {
 public:
  explicit Derived(int x) : Base(x) {}
};
```

```
int main() {
  const Base b(5);
  std::cout << b.GetX() << '\n';
}
```

Can create just Base class.

```
class Base {
 protected:
  explicit Base(int x) : x_(x) {}
  ~Base() = default;
  // Also add other special member functions if necessary.
 public:
  int GetX() const { return x_; }

 private:
  const int x_;
};
```

```
int main() {
  // Base b(5);  // Error: constructor is protected.
  const Derived d(10);
  std::cout << d.GetX() << '\n';
}
```

For a class supposed to be used only as a base class, if it does not have a pure virtual function, make the constructor protected.

# Make classes final if you don't want to derive from them.

```
class Point {
 public:
  Point(int x, int y) : x_(x), y_(y) {}
  int GetX() const { return x_; }
  int GetY() const { return y_; }

  void SetX(int x) { x_ = x; }
  void SetY(int y) { y_ = y; }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

```
class NamedPoint : public Point {
 public:
  NamedPoint(int x, int y, std::string name)
      : Point(x, y), name_(std::move(name)) {
    puts("NamedPoint()");
  }
  ~NamedPoint() { puts("~NamedPoint()"); }
  const std::string& GetName() const { return name_; }

 private:
  const std::string name_;
};
```

```
void Foo(std::unique_ptr<Point> pt) {
  // Use it.
}

int main() {
  Foo(std::make_unique<NamedPoint>(10, 20, "My point"));
}
```

```
NamedPoint()
```

Destructor for “NamedPoint” was not called.

# Make classes final if you don't want to derive from them.

```cpp
class Point final {
 public:
  Point(int x, int y) : x_(x), y_(y) {}
  int GetX() const { return x_; }
  int GetY() const { return y_; }

  void SetX(int x) { x_ = x; }
  void SetY(int y) { y_ = y; }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

```cpp
class NamedPoint : public Point {
 public:
  NamedPoint(int x, int y, std::string name)
      : Point(x, y), name_(std::move(name)) {
    puts("NamedPoint()");
  }
  ~NamedPoint() { puts("~NamedPoint()"); }
  const std::string& GetName() const { return name_; }

 private:
  const std::string name_;
};
```

```
error: base 'Point' is marked 'final'
 class NamedPoint : public Point {
                           ^
```

Making class final:

- Prevents accidental or inappropriate inheritance.
- Makes design intent explicit.

# Make classes final if you don't want to derive from them.



```cpp
class Point final {
 public:
  Point(int x, int y) : x_(x), y_(y) {}
  int GetX() const { return x_; }
  int GetY() const { return y_; }

  void SetX(int x) { x_ = x; }
  void SetY(int y) { y_ = y; }

 private:
  int x_ = 0;
  int y_ = 0;
};
```

```cpp
class NamedPoint {
 public:
  NamedPoint(int x, int y, std::string name)
      : pt_(x, y), name_(std::move(name)) {
    puts("NamedPoint()");
  }
  ~NamedPoint() { puts("~NamedPoint()"); }
  const std::string& GetName() const { return name_; }
  const Point& GetPoint() const { return pt_; }
  Point& GetPoint() { return pt_; }

 private:
  Point pt_;
  const std::string name_;
};
```

Consider using composition for such scenarios

```cpp
void Foo(const Point& pt) {
  // Use it.
}

int main() {
  Foo(NamedPoint{10, 20, "My point"}.GetPoint());
}
```

```
NamedPoint()
~NamedPoint()
```

# Mark overridden functions with override or final.

```
class Base {
 public:
  virtual ~Base() = default;
  virtual void Foo() { puts("Base::Foo"); }
};
```

```
class Derived : public Base {
 public:
  void Foo() const { puts("Derived::Foo"); }
};
```

This is not overriding the parent virtual function.

```
-Woverloaded-virtual (-Wall)
```

```
error: 'Derived::Foo' hides overloaded virtual function [-Werror,-Woverloaded-virtual]
   void Foo() const { puts("Derived::Foo"); }
        ^
note: hidden overloaded virtual function 'Base::Foo' declared here: different qualifiers (unqualified vs
'const')
   virtual void Foo() { puts("Base::Foo"); }
```

```
class Derived : public Base {
 public:
  void Foo() { puts("Derived::Foo"); }
};
```

It is best to add override to automatically catch such issues:

```
class Derived : public Base {
 public:
  void Foo() override { puts("Derived::Foo"); }
};
```

# Mark overridden functions with override or final.

```
class Base {
 public:
  virtual ~Base() = default;
  virtual void Foo() { puts("Base::Foo"); }
};
```

```
class Derived : public Base {
 public:
  void Foo() { puts("Derived::Foo"); }
};
```

```
-Wsuggest-override (-Weverything)
```

```
error: 'Foo' overrides a member function but is not marked 'override' [-Werror,-Wsuggest-override]
   void Foo() { puts("Derived::Foo"); }
        ^
note: overridden virtual function is here
   virtual void Foo() { puts("Base::Foo"); }
                ^
```

When clang-tidy check is used:
***--checks=**[modernize-use-override]()*

```
warning: annotate this function with 'override' or (rarely) 'final' [modernize-use-override]
   void Foo() { puts("Derived::Foo"); }
        ^
              override
```

# Handle multiple std::move for a single variable.

```
bool IsInteresting(const std::string& s) {
  return s.length() > 3u;
}
```

```
std::vector<std::string> GetStringVec(std::string s) {
  std::vector<std::string> vec;
  vec.push_back(std::move(s));
  if (IsInteresting(s)) {
    vec.push_back(std::move(s));
  }
  return vec;
}
```

The first move means that this code is incorrect.

```
std::vector<std::string> GetStringVec(std::string s) {
  std::vector<std::string> vec;
  if (IsInteresting(s)) {
    vec.push_back(s);
    vec.push_back(std::move(s));
  } else {
    vec.push_back(std::move(s));
  }
  return vec;
}
```

Be careful to ensure that we don't "move" from same variable multiples times.

# Handle multiple std::move for a single variable.

```
bool IsInteresting(const std::string& s) {
  return s.length() > 3u;
}
```

```
std::vector<std::string> GetStringVec(std::string s) {
  std::vector<std::string> vec;
  vec.push_back(std::move(s));
  if (IsInteresting(s)) {
    vec.push_back(std::move(s));
  }
  return vec;
}
```

When clang-tidy check is used: ***--checks=[bugprone-use-after-move]***

```
warning: 's' used after it was moved [bugprone-use-after-move]
   if (IsInteresting(s)) {
                     ^
note: move occurred here
   vec.push_back(std::move(s));
       ^
```

# Containers: Use emplace / try_emplace – non-trivial key

```
int main() {
  std::map<A, int> m;
  m[10] = 10;
}
```

```
A(10)
A(A&&): 10
~A()
~A()
```

```
int main() {
  std::map<A, int> m;
  m.emplace(10, 10);
}
```

```
A(10)
~A()
```

```
int main() {
  std::map<A, int> m;
  m.try_emplace(10, 10);
}
```

```
A(10)
A(A&&): 10
~A()
~A()
```

Only **emplace** does in-place construction if the key type is a non-trivial object.

```
struct A final {
  A() { puts("A()"); }
  A(int a) : a_(a) { printf("A(%d)\n", a_); }
  ~A() { puts("~A()"); }
  A(const A& rhs) : a_(rhs.a_) {
    printf("A(const A&): %d\n", a_);
  }
  A(A&& rhs) noexcept : a_(rhs.a_) {
    printf("A(A&&): %d\n", a_);
  }
  A& operator=(const A& rhs) {
    a_ = rhs.a_;
    printf("A& operator=(const A&): %d\n", a_);
    return *this;
  }
  A& operator=(A&& rhs) noexcept {
    a_ = rhs.a_;
    printf("A& operator=(A&&): %d\n", a_);
    return *this;
  }
  auto operator<=>(const A&) const noexcept = default;
  int a_ = 0;
};
```

This paper (P2363) was accepted for C++26 and added support for:

```
template <typename K, typename... Args>
std::pair<iterator, bool> try_emplace(K&& k, Args&&... args);
```

```
template <typename K>
mapped_type& operator[](K&& k);
```

# Containers: Use emplace / try_emplace - non-trivial value



```
int main() {
  std::map<int, A> m;
  // This code does not compile without the default
  // constructor.
  m[10] = 10;
}
```

```
A(10)
A()
A& operator=(A&&): 10
~A()
~A()
```

```
int main() {
  std::map<int, A> m;
  m.emplace(10, 10);
}
```

```
A(10)
~A()
```

```
int main() {
  std::map<int, A> m;
  m.try_emplace(10, 10);
}
```

```
A(10)
~A()
```

Both **emplace** and **try_emplace** do in-place construction if the value type is a non-trivial object.

# Algorithms: Use projections

```
struct City {
  std::string name;
  int population = 0;
  int height_in_feet = 0;
};
```

```
void SortSpecial(std::span<City> cities) {
  std::ranges::sort(cities, [](const City& a, const City& b) {
    return a.population < b.population;
  });
}

bool HasCityWithHeight(std::span<City> cities, int height) {
  return std::ranges::find_if(cities, [height](const City& a) {
           return a.height_in_feet == height;
         }) != cities.end();
}
```

```
void SortSpecial(std::span<City> cities) {
  std::ranges::sort(cities, {}, &City::population);
}

bool HasCityWithHeight(std::span<City> cities, int height) {
  return std::ranges::find(cities, height, &City::height_in_feet) !=
         cities.end();
}
```

Consider using projections in algorithm. In many cases, they can make your code simpler.

# std::unique_ptr: Use std::make_unique.

```cpp
struct Widget1 {
  Widget1(int) {}
  // Other stuff.
};
```

```cpp
struct Widget2 {
  Widget2(int) {}
  // Other stuff.
};
```

```cpp
void Foo(std::unique_ptr<Widget1> w1, std::unique_ptr<Widget2> w2) {
  std::ignore = w1;
  std::ignore = w2;
}
```

```cpp
int main() {
  Foo(std::unique_ptr<Widget1>{new Widget1{1}},
      std::unique_ptr<Widget2>{new Widget2{2}});
}
```

Has a memory leak with the following sequence:

1. new Widget1
2. new Widget2: *If this throws exception, then memory for "1" is not cleaned up.*
3. unique_ptr<Widget1>
4. unique_ptr<Widget2>

Also specifies the type (e.g., Widget1) twice.

```cpp
int main() {
  Foo(std::make_unique<Widget1>(1), std::make_unique<Widget2>(2));
}
```

Consider using std::make_unique<T>(...) instead of std::unique_ptr<T>{new T{...}} when possible.

# std::unique_ptr: Use std::make_unique.

```cpp
struct Widget1 {
  Widget1(int) {}
  // Other stuff.
};
```

```cpp
struct Widget2 {
  Widget2(int) {}
  // Other stuff.
};
```

```cpp
void Foo(std::unique_ptr<Widget1> w1, std::unique_ptr<Widget2> w2) {
  std::ignore = w1;
  std::ignore = w2;
}
```

```cpp
int main() {
  Foo(std::unique_ptr<Widget1>{new Widget1{1}},
      std::unique_ptr<Widget2>{new Widget2{2}});
}
```

When clang-tidy check is used: ***--checks=[modernize-make-unique]()***

```
warning: use std::make_unique instead [modernize-make-unique]
   Foo(std::unique_ptr<Widget1>{new Widget1{1}},
       ^~~~~~~~~~~~~~~         ~~~~~~~~~~~~~ ~~
       std::make_unique        (             )
warning: use std::make_unique instead [modernize-make-unique]
       std::unique_ptr<Widget2>{new Widget2{2}});
       ^~~~~~~~~~~~~~~         ~~~~~~~~~~~~~ ~~
       std::make_unique        (             )
```

# std::unique_ptr: No need to use

```cpp
class MyClass {
 public:
  void DoSomething() {}
  // Other stuff.
};
```

```cpp
int main() {
  auto my_class = std::make_unique<MyClass>();
  my_class->DoSomething();
  // Use `my_class` to the end of scope.
}
```

```cpp
int main() {
  MyClass my_class;
  my_class.DoSomething();
  // Use `my_class` to the end of scope.
}
```

# std::unique_ptr: No need to use nullptr constructor

```cpp
class Widget {};
```

```cpp
class Foo {
 public:
  Foo() : ptr_(nullptr) {}
  // Other stuff.

 private:
  std::unique_ptr<Widget> ptr_;
  std::unique_ptr<Widget> ptr2_{nullptr};
};

int main() {
  [[maybe_unused]] std::unique_ptr<Widget> w{nullptr};

  [[maybe_unused]] Foo f;
}
```

```cpp
class Foo {
 public:
  Foo() = default;
  // Other stuff.

 private:
  std::unique_ptr<Widget> ptr_;
  std::unique_ptr<Widget> ptr2_;
};

int main() {
  [[maybe_unused]] std::unique_ptr<Widget> w;

  [[maybe_unused]] Foo f;
}
```

Explicitly using **nullptr** in **unique_ptr** constructor is redundant.

# std::unique_ptr: Use reset instead of assigning = nullptr.

```cpp
struct A {
  // Some stuff.
};
```

```cpp
class B {
 public:
  // Functions related to creation of `a_`.

  void Reset() {
    a_ = nullptr;
  }

 private:
  std::unique_ptr<A> a_;
};

void Foo() {
  auto a = std::make_unique<A>();
  // Use.
  a = nullptr;
  // Other stuff
}
```

```cpp
class B {
 public:
  // Functions related to creation of `a_`.

  void Reset() {
    a_.reset();
  }

 private:
  std::unique_ptr<A> a_;
};

void Foo() {
  auto a = std::make_unique<A>();
  // Use.
  a.reset();
  // Other stuff
}
```

Consider using **reset()** to “reset” **std::unique_ptr**, since it makes the intent clearer.

# std::unique_ptr: No need to call reset() in destructor.

```
struct A {
  // Some stuff.
};
```

```
class B {
 public:
  ~B() {
    // Do some stuff.
    a_.reset();
    // No code after this.
  }
  // Functions related to creation of `a_`.

 private:
  std::unique_ptr<A> a_;
};
```

→

```
class B {
 public:
  ~B() {
    // Do some stuff.
  }
  // Functions related to creation of `a_`.

 private:
  std::unique_ptr<A> a_;
};
```

Consider removing **reset()** in destructors unless they are essential for destruction ordering purposes.

# Use std::visit for std::variant.

```cpp
class MyClass final {
 public:
  MyClass(int v) : int_value_(v) {}
  MyClass(std::string&& s) : str_value_(std::move(s)) {}

  bool HasIntValue() const { return !!int_value_; }
  bool HasStringValue() const { return !!str_value_; }

  const int* GetInt() const;
  const std::string* GetStr() const;

 private:
  std::optional<int> int_value_;
  std::optional<std::string> str_value_;
};
```

```cpp
void Print(const MyClass& c) {
  if (c.HasIntValue()) {
    std::cout << "Int: " << *c.GetInt() << '\n';
  } else {
    std::cout << "String: " << *c.GetStr() << '\n';
  }
}
```

```cpp
class MyClass final {
 public:
  MyClass(int v) : value_(v) {}
  MyClass(std::string&& s) : value_(std::move(s)) {}

  template <typename Visitor>
  void Visit(Visitor&& v) const {
    std::visit(v, value_);
  }
 private:
  std::variant<int, std::string> value_;
};
```

```cpp
template <class... Ts>
struct Overloaded : Ts... {
  using Ts::operator()...;
};

// Seems necessary for GCC for C++20.
template <class... Ts>
Overloaded(Ts...) -> Overloaded<Ts...>;

void Print(const MyClass& c) {
  c.Visit(Overloaded(
      [](int v) { std::cout << "Int: " << v << '\n'; },
      [](const std::string& s) { std::cout << "String: " << s << '\n'; }));
}
```

Consider using std::visit for variant since it promotes:

a) Type safety and maintainability: Compiler catches unhandled types.
b) Readability: Replacing if / else chain with cleaner syntax.

# Class: Make member variables const

If a member variable is only set in the constructor and it is a POD, consider making to const to ensure that compiler forces initialization.

```
enum class SomeEnum;

struct SomeClass {
  // Won't change this after constructor.
  SomeEnum e;

  SomeEnum GetE() const { return e; }
};
```

```
int main() {
  SomeClass s;
  printf("%d\n", static_cast<int>(s.GetE()));
}
```

*Compiles fine with clang.*

```
1953334856
```

```
struct SomeClass {
  // Won't change this after constructor.
  const SomeEnum e;

  SomeEnum GetE() const { return e; }
};
```

```
int main() {
  SomeClass s;
  printf("%d\n", static_cast<int>(s.GetE()));
}
```

```
error: call to implicitly-deleted default constructor of 'SomeClass'
   SomeClass s;
             ^
note: default constructor of 'SomeClass' is implicitly deleted because field 'e' of const-
qualified type 'const SomeEnum' would not be initialized
   const SomeEnum e;
                  ^
```

```
enum class SomeEnum {
    kA,
    kB,
    kC
};
```

```
int main() {
  SomeClass s{SomeEnum::kB};
  printf("%d\n", static_cast<int>(s.GetE()));
}
```

```
1
```

# Class: Make member variables const

If a member variable is only set in the constructor and it is a POD, consider making to const to ensure that compiler forces initialization.

```
enum class SomeEnum;

struct SomeClass {
  // Won't change this after constructor.
  SomeEnum e;

  SomeEnum GetE() const { return e; }
};
```

```
int main() {
  SomeClass s;
  printf("%d\n", static_cast<int>(s.GetE()));
}
```

*Compiling with GCC:*

```
In member function 'SomeEnum SomeClass::GetE() const',
    inlined from 'int main()':
error: 's.SomeClass::e' is used uninitialized [-Werror=uninitialized]
   SomeEnum GetE() const { return e; }
                                  ^
In function 'int main()':
note: 's' declared here
   SomeClass s;
             ^
```

*Clang-tidy: cppcoreguidelines-pro-type-member-init*

```
warning: uninitialized record type: 's' [cppcoreguidelines-pro-type-member-init]
   SomeClass s;
   ^
```

# Class: Impact of const member variables

```
struct C {
  C(MyEnum e) : e(e) {}
  void Print() const {
    printf("C::Print(): e=%s\n", ToString(e));
  }

  const MyEnum e;
  A a;
  const B b;
};
```

```
enum class MyEnum { kA, kB, kC };

const char* ToString(MyEnum e) {
  switch (e) {
    case MyEnum::kA:
      return "kA";
    case MyEnum::kB:
      return "kB";
    case MyEnum::kC:
      return "kC";
  }
}
```

```
struct A {
  A() { puts("A()"); }
  ~A() { puts("~A()"); }
  A(const A&) { puts("A(const A&)"); }
  A(A&&) noexcept { puts("A(A&&)"); }
  A& operator=(const A&) {
    puts("A& operator=(const A&)");
    return *this;
  }
  A& operator=(A&&) noexcept {
    puts("A& operator=(A&&)");
    return *this;
  }
};
```

```
struct B {
  B() { puts("B()"); }
  ~B() { puts("~B()"); }
  B(const B&) { puts("B(const B&)"); }
  B(B&&) noexcept { puts("B(B&&)"); }
  B& operator=(const B&) {
    puts("B& operator=(const B&)");
    return *this;
  }
  B& operator=(B&&) noexcept {
    puts("B& operator=(B&&)");
    return *this;
  }
};
```

```
int main() {
  C c(MyEnum::kB);
  puts("============");

  C c1 = c;
  c1.Print();
  puts("============");

  [[maybe_unused]] C c2 = std::move(c1);
  c2.Print();
  puts("============");
}
```

```
A()
B()
============
A(const A&)
B(const B&)
C::Print(): e=kB
============
A(A&&)
B(const B&)
C::Print(): e=kB
============
~B()
~A()
~B()
~A()
~B()
~A()
```

# Class: Impact of const member variables

```
struct C {
  C(MyEnum e) : e(e) {}
  void Print() const {
    printf("C::Print(): e=%s\n", ToString(e));
  }

  const MyEnum e;
  A a;
  const B b;
};
```

→

```
struct C {
  C(MyEnum e) : e(e) {}
  void Print() const {
    printf("C::Print(): e=%s\n", ToString(e));
  }

  const MyEnum e;
  A a;
  B b;
};
```

```
int main() {
  C c(MyEnum::kB);
  puts("============");

  C c1 = c;
  c1.Print();
  puts("============");

  [[maybe_unused]] C c2 = std::move(c1);
  c2.Print();
  puts("============");
}
```

```
A()
B()
============
A(const A&)
B(const B&)
C::Print(): e=kB
============
A(A&&)
B(const B&)
C::Print(): e=kB
============
~B()
~A()
~B()
~A()
~B()
~A()
```

→

```
A()
B()
============
A(const A&)
B(const B&)
C::Print(): e=kB
============
A(A&&)
B(B&&)
C::Print(): e=kB
============
~B()
~A()
~B()
~A()
~B()
~A()
```

const member variables are not “move”d.

Be careful with making non-trivial member variables “const”.

# Headers

# Header Management: Include what you use

A.h

```
#ifndef A_H_
#define A_H_

#include <string>

struct A {
  // Other stuff.
  int Foo(const std::string& str);
};

#endif  // A_H_
```

B.h

```
#ifndef B_H_
#define B_H_

#include "A.h"

struct B {
  size_t Bar(const std::string& s);
  // Other stuff.
  A a_;
};

#endif  // B_H_
```

B.h

```
#ifndef B_H_
#define B_H_

#include <string>

#include "A.h"

struct B {
  size_t Bar(const std::string& s);
  // Other stuff.
  A a_;
};

#endif  // B_H_
```

B.cc

```
#include "B.h"

size_t B::Bar(const std::string& s) {
  return s.size();
}
```

Include what you use tool can help in figuring out such issues.

# Header Management: Use forward declaration

B.h

```cpp
#ifndef B_H_
#define B_H_

#include "A.h"

class B {
 public:
  B();

  void Foo(A a);
  void Bar(const A& a);
  A Baz();

 private:
  A* pa = nullptr;
};

#endif  // B_H_
```

B.h

```cpp
#ifndef B_H_
#define B_H_

// Forward declaration.
class A;

class B {
 public:
  B();

  void Foo(A a);
  void Bar(const A& a);
  A Baz();

 private:
  A* pa = nullptr;
};

#endif  // B_H_
```

# Header Management: Use forward declaration

B.h

```
#ifndef B_H_
#define B_H_

#include "my_enum.h"

struct B {
  // Other stuff.
  bool Foo(MyEnum e);
  // Other stuff.
};

#endif  // B_H_
```

B.cc

```
#include "B.h"

bool B::Foo(MyEnum e) {
  return e >= MyEnum::kB;
}
```

B.h

```
#ifndef B_H_
#define B_H_

// Forward declaration.
enum class MyEnum;

struct B {
  // Other stuff.
  bool Foo(MyEnum e);
  // Other stuff.
};

#endif  // B_H_
```

B.cc

```
#include "B.h"

#include "my_enum.h"

bool B::Foo(MyEnum e) {
  return e >= MyEnum::kB;
}
```

# Header Management: Must include

A.h

```
#ifndef A_H_
#define A_H_

struct A {
  // Other stuff.
  struct Inner {
   // Other stuff.
  };
};

#endif  // A_H_
```

B.h

```
#ifndef B_H_
#define B_H_

#include "A.h"

struct B : A {
  // Other stuff.
 bool Foo(A::Inner inner);
  // Other stuff.
  A a_;
};

#endif  // B_H_
```

Base class

Inner class

Member object

# Header Management: Use forward declaration

B.h

```cpp
#ifndef B_H_
#define B_H_

namespace std {
template <typename T>
class optional;
}  // namespace std

struct B {
  // Other stuff.
  int Foo(const std::optional<int>& i);
  // Other stuff.
};

#endif  // B_H_
```

B.cc

```cpp
#include "B.h"

#include <optional>

int B::Foo(const std::optional<int>& i) {
  return 10 + i.value_or(20);
}
```

main.cc

```cpp
#include <optional>

#include "B.h"

int main() {
  B b;
  return b.Foo(10);
}
```

B.h

```cpp
#ifndef B_H_
#define B_H_

#include <optional>

struct B {
  // Other stuff.
  int Foo(const std::optional<int>& i);
  // Other stuff.
};

#endif  // B_H_
```

B.cc

```cpp
#include "B.h"

int B::Foo(const std::optional<int>& i) {
  return 10 + i.value_or(20);
}
```

main.cc

```cpp
#include "B.h"

int main() {
  B b;
  return b.Foo(10);
}
```

For STL classes, include the header directly instead of a forward declaration.

# Header Management: Header guards

components/my_class.h

```cpp
class MyClass final {
 public:
  MyClass(int starting);

  int GetDouble() const;
  int GetAfterAdd(int i) const;

  // Some other functions.
 private:
  int starting_ = 0;
};
```

Use header guards

```cpp
#ifndef COMPONENTS_MY_CLASS_H_
#define COMPONENTS_MY_CLASS_H_

class MyClass final {
 public:
  MyClass(int starting);

  int GetDouble() const;
  int GetAfterAdd(int i) const;

  // Some other functions.
 private:
  int starting_ = 0;
};

#endif  // COMPONENTS_MY_CLASS_H_
```

clang-tidy check: ***llvm-header-guard***

Use #pragma (not standard compliant)

```cpp
#pragma once

class MyClass final {
 public:
  MyClass(int starting);

  int GetDouble() const;
  int GetAfterAdd(int i) const;

  // Some other functions.
 private:
  int starting_ = 0;
};
```