C++ now

2025

Streamlining C++ Code

Avoiding Unnecessary Object Creation

Kathleen Baker
& Chandranath Bhattacharyya

Contents

What are extra objects?

Consider this code:

```
int main() {
  const std::string str{"Hello World!!\n"};
  const std::string str2{str};
  std::cout << str2;
}
```

```
Hello World!!
```

If our goal was to print the string, then the following code would suffice:

```
int main() {
  std::cout << "Hello World!!\n";
}
```

We don't need **str** and **str2** to achieve our goal of printing the string.

Even with compiler optimization, the second code snippet generates much less code.

We consider these objects which are not necessary to achieve our goal as extra objects.

Not all scenarios for "extra" objects are so straightforward to detect.

Motivation for this talk

- We work on a Chromium based C++ project (clang, no exceptions enabled) with over 100 engineers.
- Over the past 5 years, while reviewing pull requests, we noticed certain patterns of "extra objects" being created.
- This talk summarizes the most common patterns we've observed.
- This presentation is geared toward being guidelines of “correct” code.

Why are extra objects a problem?

C++ is a value (copy) semantic language by default.

```
MyClass c;
MyClass c1 = c;
```

c and c1 are different objects.

This statement calls the “copy constructor” to create a new object.

```
void* operator new(size_t n) {
  void* p = malloc(n);
  printf("operator new: n: %zu, p = %p\n", n, p);
  return p;
}

void operator delete(void* p) noexcept {
  printf("operator delete: p = %p\n", p);
  free(p);
}
```

The print statements in **operator new** and **operator delete** will show us when objects are created and destroyed in our examples.

Why are extra objects a problem?

C++ is a value (copy) semantic language by default.

```
MyClass c;
MyClass c1 = c;
```

c and c1 are different objects.

This statement calls the “copy constructor” to create a new object.

```
int main() {
  puts("==== Before initial string ======");
  const std::string s("This is hello world string!!");
  puts("==== Before copy constructor ======");
  const std::string s1 = s;
  puts("==== Before end ======");
}
```

```
==== Before initial string ======
operator new: n: 32, p = 0x5afba82c42b0
==== Before copy constructor ======
operator new: n: 32, p = 0x5afba82c42e0
==== Before end ======
operator delete: p = 0x5afba82c42e0
operator delete: p = 0x5afba82c42b0
```

The copy constructor of the string is called and that calls “operator new” to allocate memory.

Allocating memory is a costly runtime operation. Avoiding that is beneficial.

Why are extra objects a problem?

C++ is a value (copy) semantic language by default.

```
MyClass c;
MyClass c1 = std::move(c);
```

This statement calls the “move constructor” to create a new object.

```
int main() {
  puts("==== Before initial string ======");
  std::string s("This is hello world string!!");
  puts("==== Before move constructor ======");
  const std::string s1 = std::move(s);
  puts("==== Before end ======");
}
```

```
==== Before initial string ======
operator new: n: 32, p = 0x5f0c2817b2b0
==== Before move constructor ======
==== Before end ======
operator delete: p = 0x5f0c2817b2b0
```

The “move constructor” of string “does not” allocate memory, but just swaps memory.

Even though “move” creates a new object, it is not “costly” for runtime.

Why are extra objects a problem?

C++ is a value (copy) semantic language by default.

```
std::string GetStr() {
  return "This is hello world string!!";
}

int main() {
  puts("==== Before GetStr() ======");
  const std::string s = GetStr();
  puts("==== Before end ======");
}
```

```
==== Before GetStr() ======
operator new: n: 32, p = 0x5e83f9b412b0
==== Before end ======
operator delete: p = 0x5e83f9b412b0
```

This is in-place construction, and no extra objects are being created.

Creating objects in-place is preferrable over moving, which is preferrable over copying.

Why are extra objects a problem?

Let's consider std::vector:

```cpp
int main() {
  std::vector<int> v{1, 2, 3, 4, 5, 6};
  puts("==== Before copy constructor ======");
  const std::vector<int> v1(v);
  puts("==== Before end ======");
}
```

```
operator new: n: 24, p = 0x5770406a22a0
==== Before copy constructor ======
operator new: n: 24, p = 0x5770406a32d0
==== Before end ======
operator delete: p = 0x5770406a32d0
operator delete: p = 0x5770406a22a0
```

std::vector's copy constructor is "costly" for runtime, since it allocates memory.

```cpp
int main() {
  std::vector<int> v{1, 2, 3, 4, 5, 6};
  puts("==== Before move constructor ======");
  const std::vector<int> v1(std::move(v));
  puts("==== Before end ======");
}
```

```
operator new: n: 24, p = 0x61b6931952a0
==== Before move constructor ======
==== Before end ======
operator delete: p = 0x61b6931952a0
```

Move constructor doesn't allocate memory.

Why are extra objects a problem?

Let's consider std::list:

```
int main() {
  std::list<int> l{1, 2, 3};
  puts("==== Before copy constructor ======");
  const std::list<int> l1(l);
  puts("==== Before end ======");
}
```

```
operator new: n: 24, p = 0x562dc956e2a0
operator new: n: 24, p = 0x562dc956f2d0
operator new: n: 24, p = 0x562dc956f2f0
==== Before copy constructor ======
operator new: n: 24, p = 0x562dc956f310
operator new: n: 24, p = 0x562dc956f330
operator new: n: 24, p = 0x562dc956f350
==== Before end ======
operator delete: p = 0x562dc956f310
operator delete: p = 0x562dc956f330
operator delete: p = 0x562dc956f350
operator delete: p = 0x562dc956e2a0
operator delete: p = 0x562dc956f2d0
operator delete: p = 0x562dc956f2f0
```

std::list's copy constructor is even more expensive.

```
int main() {
  std::list<int> l{1, 2, 3};
  puts("==== Before move constructor ======");
  const std::list<int> l1(std::move(l));
  puts("==== Before end ======");
}
```

```
operator new: n: 24, p = 0x55d161e262a0
operator new: n: 24, p = 0x55d161e272d0
operator new: n: 24, p = 0x55d161e272f0
==== Before move constructor ======
==== Before end ======
operator delete: p = 0x55d161e262a0
operator delete: p = 0x55d161e272d0
operator delete: p = 0x55d161e272f0
```

Move constructor is better because it doesn't allocate memory.

Why are extra objects a problem?

- The object's creation can cause costly operations at runtime (like memory allocation, etc.)
- Can cause more code to be executed at runtime.
- Can need more stack / heap space at runtime.

What performance benefit will I get if I remove extra objects?

Please measure your scenario!

For which types are extra objects ok?

It’s fine to create extra objects for:

- Plain old data types, e.g. int, float, etc.
- Objects that are “small” in size and do “non-costly” operations in their constructors.
    - std::string_view, std::span, std::initializer_list, std::mdspan, range view types.

What are costly operations and small objects?

- Costly operations can be memory allocation, making operating system calls, running costly algorithms, etc.
- As a rule of thumb, we can consider objects C++ STL considers to be lightweight as “small” objects.
  - std::string_view, std::span are 16 bytes in 64 bit.
  - std::mdspan is 24 bytes in 64 bit.
  - If your user defined type doesn’t have any costly copy operations and is <= 24 bytes (64 bit) we can consider it “small”.

Compiler Warnings

-Wexit-time-destructors

-Wglobal-constructors

-Wpessimizing-move

-Wrange-loop-construct

-Wall

-Wextra

-Weverything

Clang-Tidy Checks

performance-unnecessary-value-param

performance-unnecessary-copy-initialization

performance-for-range-copy

modernize-pass-by-value

performance-inefficient-vector-operation

performance-noexcept-move-constructor

modernize-use-emplace

Basic Strategies to Avoid Extra Temporary Objects

2/7

Non-trivial type as read-only argument to function

```
void Foo(std::string s) {
  // Code that only reads `s`.
}
```

```
void* operator new(size_t n) {
  void* p = malloc(n);
  printf("operator new: n: %zu, p = %p\n", n, p);
  return p;
}

void operator delete(void* p) noexcept {
  printf("operator delete: p = %p\n", p);
  free(p);
}
```

```
int main() {
  std::string s("This is a hello world string");
  puts("==== Before Foo call ======");

  Foo(s);

  puts("==== After Foo call ======");
}
```

```
operator new: n: 32, p = 0x5a78e62a82a0
==== Before Foo call ======
operator new: n: 32, p = 0x5a78e62a92e0
operator delete: p = 0x5a78e62a92e0
==== After Foo call ======
operator delete: p = 0x5a78e62a82a0
```

It is better to use const & in this case:

```
void Foo(const std::string& s) {
  // Code that only reads `s`.
}

int main() {
  std::string s("This is a hello world string");
  puts("==== Before Foo call ======");

  Foo(s);

  puts("==== After Foo call ======");
}
```

```
operator new: n: 32, p = 0x577ff6ef62a0
==== Before Foo call ======
==== After Foo call ======
operator delete: p = 0x577ff6ef62a0
```

As a rule of thumb pass “non-trivial” “read-only” objects as const reference in arguments to functions.

It is better to pass read-only “string”s as std::string_view

There are some nuances with using std::string_view that we will consider in later slides.

const & is “not” the most optimal in some cases. We will cover such cases later in this presentation.

Non-trivial type as read-only argument to function

```cpp
void Foo(std::string s) {
  // Code that uses `s`.
}
```

When clang-tidy check is used: ***--checks=performance-unnecessary-value-param***

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

```cpp
void Foo(std::string_view s) {}
```
✓

```cpp
struct B {
  B() = default;
  B(const B&) = default;
  B(B&&) noexcept = default;
};

void Foo(B b) {}
```
✓

```cpp
struct B {
  B() = default;
  B(const B&);
  B(B&&) noexcept = default;
};

B::B(const B&) = default;

void Foo(B b) {}
```

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

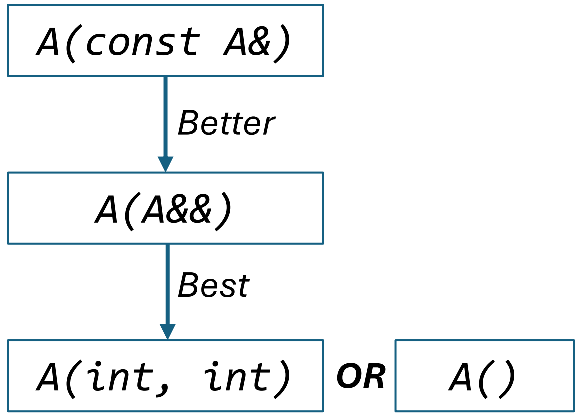
From this documentation:
The check is only applied to parameters of types that are expensive to copy which means they are not trivially copyable or have a non-trivial copy constructor or destructor.

Returning non-trivial member object

```
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 copies are being made.



We want in-place construction without copies or moves.

Returning non-trivial member object

```
class MyClass final {
 public:
  MyClass() : a_(10, 10) {}

  const A& a() const { return a_; }

  A a_not_great() const { return a_; }

 private:
  A a_;
};
```

```
int main() {
  MyClass obj;
  std::cout << "====== Before a_not_great ======\n";
  {
    const auto a = obj.a_not_great();
  }
  std::cout << "====== After a_not_great ======\n";
}
```

```
A(int, int)
====== Before a_not_great ======
A(const A&)
~A()
====== After a_not_great ======
~A()
```

```
int main() {
  MyClass obj;
  std::cout << "====== Before a() ======\n";
  {
    const auto& a = obj.a();
    (void)a; // Suppress warning.
  }
  std::cout << "====== After a() ======\n";
}
```

```
A(int, int)
====== Before a() ======
====== After a() ======
~A()
```

No object is created for the **`const&`** return. Without **`const&`** the copy constructor is called.

When returning non-trivial member variable from member function consider returning as **const&**.

Returning non-trivial member object

```
class MyClass final {
 public:
  MyClass() : a_(10, 10) {}

  const A& a() const { return a_; }

  A a_not_great() const { return a_; }

 private:
  A a_;
};
```

```
int main() {
  MyClass obj;
  std::cout << "==== Before const auto& a ====\n";
  {
    const auto& a = obj.a();
    (void)a;  // Suppress warning.
  }
  std::cout << "==== After const auto& a ====\n";
}
```

```
A(int, int)
==== Before const auto& a ====
==== After const auto& a ====
~A()
```

```
int main() {
  MyClass obj;
  std::cout << "==== Before const auto a ====\n";
  {
    const auto a = obj.a();
  }
  std::cout << "==== After const auto a ====\n";
}
```

```
A(int, int)
==== Before const auto a ====
A(const A&)
~A()
==== After const auto a ====
~A()
```

When the returned type is not held as **`const&`**, the copy constructor is still called to create an extra object.

Returning non-trivial member object

Return **const&** for a non-trivial type of member object being returned from a class's **const** member function

```
class MyClass final {
 public:
  MyClass() : a_(10, 10) {}

  const A& a() const { return a_; }

  A a_not_great() const { return a_; }

 private:
  A a_;
};
```

```
int main() {
  MyClass obj;

  const auto a = obj.a_not_great();

  const auto& a_ref = obj.a();
  (void)a_ref;  // Suppress warning.

  const auto a_copy = obj.a();

  MyClass obj2 = obj;

  // Suppress warnings.
  (void)a_copy;
  (void)obj2;
}
```

When clang-tidy check is used: ***--checks=performance-unnecessary-copy-initialization***

```
warning: the const qualified variable 'a_copy' is copy-constructed from a const reference; consider
making it a const reference [performance-unnecessary-copy-initialization]
   const auto a_copy = obj.a();
              ^
             &
warning: local copy 'obj2' of the variable 'obj' is never modified; consider avoiding the copy
[performance-unnecessary-copy-initialization]
   MyClass obj2 = obj;
           ^
   const  &
```

Range based for loop

```cpp
std::vector<A> GetVec(int n) {
  std::vector<A> vec;
  vec.reserve(n);
  for (int i = 0; i < n; ++i) {
    vec.emplace_back(i, i);
  }
  return vec;
}
```

```cpp
int main() {
  const auto vec = GetVec(2);
  std::cout << "===== Before `auto` traversal ====\n";
  {
    for (const auto obj : vec) {
      // Do stuff;
    }
  }
  std::cout << "=== Before `const auto&` traversal ===\n";
  {
    for (const auto& obj : vec) {
      // Do stuff;
    }
  }
  std::cout << "=== Before `auto&&` traversal ===\n";
  {
    for (auto&& obj : vec) { // Forwarding reference.
      // Do stuff; `obj` is of type A&.
    }
  }
  std::cout << "===== Before end ====\n";
}
```

```
A(int, int)
A(int, int)
===== Before `auto` traversal ====
A(const A&)
~A()
A(const A&)
~A()
=== Before `const auto&` traversal ===
=== Before `auto&&` traversal ===
===== Before end ====
~A()
~A()
```

const auto traversals creates extra objects.

const auto& and auto&& leads to no temporary objects.

Use const auto& (or auto&&) in range based for loop traversal for non-trivial object container.

Range based for loop

```cpp
int main() {
  const auto vec = GetVec(2);
  std::cout << "===== Before `auto` traversal ====\n";
  {
    for (const auto obj : vec) {
      // Do stuff;
    }
  }
  std::cout << "=== Before `const auto&` traversal ===\n";
  {
    for (const auto& obj : vec) {
      // Do stuff;
    }
  }
  std::cout << "=== Before `auto&&` traversal ===\n";
  {
    for (auto&& obj : vec) { // Forwarding reference.
      // Do stuff; `obj` is of type A&.
    }
  }
  std::cout << "===== Before end ====\n";
}
```

When we use:
-Wrange-loop-construct

```
error: loop variable 'obj' creates a copy from type 'const A' [-Werror,-Wrange-loop-construct]
     for (const auto obj : vec) {
                     ^
note: use reference type 'const A &' to prevent copying
     for (const auto obj : vec) {
          ^~~~~~~~~~~~~~~~
                     &
```

Range based for loop

Finding issues with ***-Wrange-loop-construct***

```
struct A {
  void Foo() const {}
  void Bar() {}
  std::string str;
};
```

```
int main() {
  std::vector<std::string> vec;
  for (auto v : vec) {
    std::cout << v << '\n';
  }
  std::vector<A> a_vec;
  for (const auto a : a_vec) {
    a.Foo();
  }
  for (auto a : a_vec) {
    a.Bar();
  }
}
```

```
error: loop variable 'a' creates a copy from type 'A const' [-Werror,-Wrange-loop-construct]
   for (const auto a : a_vec) {
                   ^
note: use reference type 'A const &' to prevent copying
   for (const auto a : a_vec) {
        ^~~~~~~~~~~~~
                   &
```

When clang-tidy check is used:
--checks=performance-for-range-copy

```
warning: loop variable is copied but only used as const reference; consider making it a const
reference [performance-for-range-copy]
   for (auto v : vec) {
             ^
        const  &
warning: the loop variable's type is not a reference type; this creates a copy in each
iteration; consider making this a reference [performance-for-range-copy]
   for (const auto a : a_vec) {
                   ^
                  &
```

Structured binding

```
struct B {
  A a{1, 1};
  int i = 0;
};
```

```
int main() {
  B b;
  [[maybe_unused]] const auto [a, _] = b;
}
```

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

```
int main() {
  B b;
  [[maybe_unused]] const auto& [a, _] = b;
}
```

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

const auto creates extra object.

const auto& leads to no temporary objects.

Consider using const & for read-only structured binding variables.

Explicitly move-ing object out of function

```
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;
}

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

```
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.

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

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

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

This is guaranteed copy elision from C++17.

Also known as:

a) Deferred Temporary materialization
b) Unmaterialized value passing.

Explicitly move-ing object out of function

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

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

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

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);
```

Scenario for explicit move on return

```
struct B {
  A a{1, 1};
  int i = 0;
};
```

```
A Foo() {
  std::cout << "---- Start of Foo ----\n";
  B b;
  std::cout << "---- Before structured binding ----\n";
  auto& [a, _] = b;
  return a;
}

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

```
---- Start of Foo ----
A(int, int)
---- Before structured binding ----
A(const A&)
~A()
~A()
```

```
A Foo() {
  std::cout << "---- Start of Foo ----\n";
  B b;
  std::cout << "---- Before structured binding ----\n";
  auto& [a, _] = b;
  return std::move(a);
}

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

```
---- Start of Foo ----
A(int, int)
---- Before structured binding ----
A(A&&)
~A()
~A()
```

Without explicit std::move, copy constructor gets called in this scenario.

Scenario for explicit move on return

```
struct B {
  A a{1, 1};
  int i = 0;
};
```

```
A Foo() {
  B b;
  return b.a;
}

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

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

```
A Foo() {
  B b;
  return std::move(b.a);
}

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

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

Without explicit std::move, copy constructor gets called in this scenario.

Lambda captures

```
int main() {
  A a;
  std::cout << "===== Before fn =====\n";
  auto fn = [a]() {
    // Do stuff.
  };
  std::cout << "=== Before fn2 = fn ===\n";
  auto fn2 = fn;
  std::cout << "=== After fn2 = fn ===\n";
}
```

```
A()
===== Before fn =====
A(const A&)
=== Before fn2 = fn ===
A(const A&)
=== After fn2 = fn ===
~A()
~A()
~A()
```

```
int main() {
  A a;
  std::cout << "===== Before fn =====\n";
  auto fn = [&a]() {
    (void)a;  // Suppress warning.
  };
  std::cout << "=== Before fn2 = fn ===\n";
  auto fn2 = fn;
  (void)fn2;  // Suppress warning.
  std::cout << "=== After fn2 = fn ===\n";
}
```

```
A()
===== Before fn =====
=== Before fn2 = fn ===
=== After fn2 = fn ===
~A()
```

Capture by reference to avoid copy.

Lambda captures

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

  auto GetCapture() {
    return [*this]() { std::cout << "Inside lambda in GetCapture\n"; };
  }
  auto GetCapture1() {
    return [this]() { std::cout << "Inside lambda in GetCapture1\n"; };
  }
  auto GetCapture2() {
    return [&]() { std::cout << "Inside lambda in GetCapture2\n"; };
  }
  auto GetCapture3() {
    return [=]() { std::cout << "Inside lambda in GetCapture3\n"; };
  }
};
```

Lambda captures

```
struct A {
  // Special member functions.

  auto GetCapture() {
    return [*this]() {
      std::cout << "Inside lambda in GetCapture\n"; };
  }
  auto GetCapture1() {
    return [this]() {
      std::cout << "Inside lambda in GetCapture1\n"; };
  }
  auto GetCapture2() {
    return [&]() {
     std::cout << "Inside lambda in GetCapture2\n"; };
  }
  auto GetCapture3() {
    return [=]() {
     std::cout << "Inside lambda in GetCapture3\n"; };
  }
};
```

```
int main() {
  A a;
  std::cout << "==== Before a.GetCapture() ====\n";
  {
    auto ln = a.GetCapture();
    ln();
  }
  std::cout << "==== Before a.GetCapture1() ====\n";
  {
    auto ln = a.GetCapture1();
    ln();
  }
  std::cout << "==== Before a.GetCapture2() ====\n";
  {
    auto ln = a.GetCapture2();
    ln();
  }
  std::cout << "==== Before a.GetCapture3() ====\n";
  {
    auto ln = a.GetCapture3();
    ln();
  }
  std::cout << "==== Before end ====\n";
}
```

```
A()
==== Before a.GetCapture() ====
A(const A&)
Inside lambda in GetCapture
~A()
==== Before a.GetCapture1() ====
Inside lambda in GetCapture1
==== Before a.GetCapture2() ====
Inside lambda in GetCapture2
==== Before a.GetCapture3() ====
Inside lambda in GetCapture3
==== Before end ====
~A()
```

`*this` will make a copy of the object.

Lambda captures

```
struct A {
  int a = 10;

  auto GetCapture() {
    return [*this]() {
      std::cout << "Inside lambda in GetCapture: a: " << a << '\n';
    };
  }
  auto GetCapture1() {
    return [this]() {
      std::cout << "Inside lambda in GetCapture1: a: " << a << '\n';
    };
  }
  auto GetCapture2() {
    return [&]() {
      std::cout << "Inside lambda in GetCapture2: a: " << a << '\n';
    };
  }
  auto GetCapture3() {
    return [=]() {
      std::cout << "Inside lambda in GetCapture3: a: " << a << '\n';
    };
  }
};
```

```
error: implicit capture of 'this' with a capture default of '=' is
deprecated [-Werror,-Wdeprecated-this-capture]

std::cout << "Inside lambda in GetCapture3: a: " << a << '\n';
```

Implicit capture of this via [=] was deprecated in C++20 (P0806R2)

Lambda captures

```cpp
template <typename... Args>
auto Foo(Args&&... args) {
  return [args...]() {};
}

template <typename... Args>
auto Foo2(Args&&... args) {
  return [... args = args]() {};
}

template <typename... Args>
auto Foo3(Args&&... args) {
  return [... args = std::forward<Args>(args)]() {};
}

template <typename... Args>
auto Foo4(Args&&... args) {
  return [&args...]() {};
}

template <typename... Args>
auto Foo5(Args&&... args) {
  return [... args = &args]() {};
}
```

```cpp
int main() {
  A a;
  std::cout << "==== Before Foo() ====\n";
  {
    Foo(a);
  }
  std::cout << "==== Before Foo2() ====\n";
  {
    Foo2(a);
  }
  std::cout << "==== Before Foo3() ====\n";
  {
    Foo3(a);
  }
  std::cout << "==== Before Foo4() ====\n";
  {
    Foo4(a);
  }
  std::cout << "==== Before Foo5() ====\n";
  {
    Foo5(a);
  }
  std::cout << "==== Before end ====\n";
}
```

```
A()
==== Before Foo() ====
A(const A&)
~A()
==== Before Foo2() ====
A(const A&)
~A()
==== Before Foo3() ====
A(const A&)
~A()
==== Before Foo4() ====
==== Before Foo5() ====
==== Before end ====
~A()
```

Use reference capture with variadic arguments to remove extra copies.

Strategies for std::string and std::vector

3/7

string_view instead of **string**

Use std::string_view to stop creating std::string at runtime.

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

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

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

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

  // No temporary string 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}.
}
```

```
Foo: s: Hello


FooBetter: s: Hello
FooBetter: s: Hello
FooBetter: s: Hell
```

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");
  // Created at compile time.
  static constexpr std::string_view kStr("Hello");
}
```

std::string_view can also be used to create the string at compile time instead of runtime.

Checkout this presentation *by Jasmine Lopez & Prithvi Okade in CppCon 2024* for more details on string_view.

string_view instead of **string**

Use std::string_view to stop creating std::string at runtime.

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

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

```
// Platform function which expects null
terminated string.
void FooPlatform(const char* p);

void Foo(const std::string& s) {
  FooPlatform(s.c_str());
}
```

Cannot use std::string_view in this case, since it may not be null terminated.

There isn’t a C++ standard “null-terminated-string-view-type”.

Chromium has base::cstring_view which is a null-terminated-string-view type.

Similar types can be used as replacement in this scenario.

string_view instead of **string**

What about the following cases?

```
struct A {
  A(const std::string& str) : str_(str) {}
  // Other functions.
  void SetStr(const std::string& str) { str_ = str; }
  // Other functions.
  std::string str_;
};
```

These are “sink” scenarios.

Instead of using std::string_view, we can follow the cpp core guideline: *F.18: For "will-move-from" parameters, pass by X&& and std::move the parameter.*

```
struct A {
  A(std::string&& str) : str_(std::move(str)) {}
  // Other functions.
  void SetStr(std::string&& str) { str_ = std::move(str); }
  // Other functions.
  std::string str_;
};
```

string_view instead of **string**

Sink scenario:

```
struct A {
  A(const std::string& str) : str_(str) {}
  // Other functions.
  void SetStr(const std::string& str) { str_ = str; }
  // Other functions.
  std::string str_;
};
```

When the following clang-tidy check is used:
--checks=modernize-pass-by-value

```
warning: pass by value and use std::move [modernize-pass-by-value]
   A(const std::string& str) : str_(str) {}
     ^~~~~~~~~~~~~~~~~~
     std::string                    std::move( )
```

From the documentation for this check:
Currently, only constructors are transformed to make use of pass-by-value. Contributions that handle other situations are welcome!

std::string::operator+ can cause extra strings

```
const std::string s = std::string{"Hello there. Good morning!"} +
                      " Hope you are doing great!" +
                      " How's the weather in Aspen?";
std::cout << s << '\n';
```

```
operator new: size: 32
operator new: size: 64
operator delete
operator new: size: 128
operator delete
Hello there. Good morning! Hope you are doing great! How's the
weather in Aspen?
operator delete
```

```
const std::string s =
    MyStrCat({"Hello there. Good morning! ",
              "Hope you are doing great!",
              " How's the weather in Aspen?"});
std::cout << s << '\n';
```

```
operator new: size: 88
Hello there. Good morning!
Hope you are doing great! How's the weather in Aspen?
operator delete
```

```
std::string MyStrCat(std::initializer_list<std::string_view> strs) {
  size_t len = 0;
  for (const auto str : strs) {
    len += str.size();
  }
  std::string final_str;
  final_str.reserve(len + 1);
  for (const auto str : strs) {
    final_str += str;
  }
  return final_str;
}
```

Only one string is created with MyStrCat

CppCon 2017 Lightning talk *by Jorg Brown* about absl::StrCat.

std::string::operator+ can cause extra strings

For "small" strings, no memory is allocated on the heap although temporary strings do get created

```
const std::string s = std::string{"Hello"} + " " + "World" + "!";
std::cout << s << '\n';
```

```
Hello World!
```

```
const std::string s = MyStrCat({"Hello", " ", "World", "!"});
std::cout << s << '\n';
```

```
Hello World!
```

MyStrCat is still optimal because it only creates one string

+ is only optimal if there is a single concatenation and either one of the parameters is std::string.

```
int main() {
  std::string s;  // Fill it in.
  const auto sr = s + "right";
  const auto sl = "left" + s;
}
```

Use **string_view::substr** to remove possible memory allocation

```cpp
static constexpr std::string_view kHttp("http://");
static constexpr std::string_view kHttps("https://");

std::string RemoveScheme(const std::string& s) {
  if (s.starts_with(kHttp)) {
    return s.substr(kHttp.size());
  }
  if (s.starts_with(kHttps)) {
    return s.substr(kHttps.size());
  }
  return {};
}

std::string_view RemoveSchemeBetter(std::string_view s) {
  if (s.starts_with(kHttp)) {
    return s.substr(kHttp.size());
  }
  if (s.starts_with(kHttps)) {
    return s.substr(kHttps.size());
  }
  return {};
}
```

```cpp
int main() {
  const std::string str("https://cppnow.org/announcements/");

  std::cout << "========= Before substr ================\n";
  const auto scheme_removed = RemoveScheme(str);
  std::cout << "Scheme removed: " << scheme_removed << '\n';

  std::cout << "========= After string substr ================\n";

  const auto scheme_removed_sv = RemoveSchemeBetter(str);
  std::cout << "Scheme removed better: " << scheme_removed_sv << '\n';

  std::cout << "========= After string_view substr ================\n";
}
```

```
operator new: size: 40
========= Before substr ================
operator new: size: 32
Scheme removed: cppnow.org/announcements/
========= After string substr ================
Scheme removed better: cppnow.org/announcements/
========= After string_view substr ================
operator delete
operator delete
```

If we don't intend to modify the result of **substr()** we can use **std::string_view::substr()** to remove possible memory allocation.

Ensure the underlying string memory is valid when the std::string_view is used.

Use reserve for vector

Use **reserve** if we know the size of the vector in advance

```cpp
int main() {
  constexpr int kTestSize = 4;
  std::cout << "sizeof(A): " << sizeof(A) << '\n';
  std::vector<A> vec;
  std::cout << "--- After `vec` creation ---\n";
  for (int i = 0; i < kTestSize; ++i) {
    vec.emplace_back(i);
  }
}
```

```
sizeof(A): 4
--- After `vec` creation ---
operator new: size: 4
A(0)
operator new: size: 8
A(1)
A(A&&): 0
~A()
operator delete
operator new: size: 16
A(2)
A(A&&): 0
A(A&&): 1
~A()
~A()
operator delete
A(3)
~A()
~A()
~A()
~A()
operator delete
```

```cpp
int main() {
  constexpr int kTestSize = 4;
  std::cout << "sizeof(A): " << sizeof(A) << '\n';
  std::vector<A> vec;
  std::cout << "--- After `vec` creation ---\n";
  vec.reserve(kTestSize);
  std::cout << "--- After `vec.reserve(kTestSize)` ---\n";
  for (int i = 0; i < kTestSize; ++i) {
    vec.emplace_back(i);
  }
}
```

```
sizeof(A): 4
--- After `vec` creation ---
operator new: size: 16
--- After `vec.reserve(kTestSize)` ---
A(0)
A(1)
A(2)
A(3)
~A()
~A()
~A()
~A()
operator delete
```

reserve ensures there's a single allocation and hence no temporaries during resize.

Use reserve for vector

Use **reserve** if we know the size of the vector in advance

```
int main() {
  std::vector<A> vec;
  std::cout << "--- After `vec` creation ---\n";
  for (int i = 0; i < 4; ++i) {
    vec.emplace_back(i);
  }
}
```

When clang-tidy check is used:
--checks=performance-inefficient-vector-operation

```
warning: 'emplace_back' is called inside a loop; consider pre-allocating the
container capacity before the loop [performance-inefficient-vector-operation]
   for (int i = 0; i < 4; ++i) {
     vec.emplace_back(i);
     ^
```

Use span to stop forcing a vector creation.

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

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

```cpp
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";

  // Temporary vector being created.
  { Foo({arr, arr + 3}); }

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

```
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
=====
===== Before end =====
~A()
~A()
~A()
```

Using **std::span** instead of **std::vector allows** the function to be used with C-array without the need to create a vector.

Use span to stop forcing a vector creation.

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

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

This allows the function to also work with other contiguous containers like vector, array, initializer_list.

✓

```
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 as we will see in the later slides.

Use explicit std::move when a non-temporary object needs to be created.

Here's an example for std::vector.

```cpp
int main() {
  std::vector<A> vec;
  vec.reserve(2);
  std::cout << "==== Before non-move push_back ====\n";
  {
    A a(10, 10);
    // Assume we update `a` based on some conditions.
    vec.push_back(a);
  }
  std::cout << "==== Before move push_back ====\n";
  {
    A a(10, 10);
    // Assume we update `a` based on some conditions.
    vec.push_back(std::move(a));  // `move` is better here.
  }
  std::cout << "==== Before end ====\n";
}
```

```
==== Before non-move push_back ====
A(int, int)
A(const A&)
~A()
==== Before move push_back ====
A(int, int)
A(A&&)
~A()
==== Before end ====
~A()
~A()
```

In this scenario, since we cannot remove the “extra” “non-trivial” object, it is best to use **std::move** to “steal” resources and gain performance.

vector: Use **noexcept move** constructor for any object

Consider this scenario where the “non-trivial” object has a no-**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() {
  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.

vector: Use **noexcept move** constructor for any object

Consider the case where we add “noexcept” to std::move specification:

```cpp
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;
};
```

```cpp
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()
```

Note: This is not applicable to code that is built with _LIBCPP_HAS_NO_EXCEPTIONS with libc++.

For such configuration, classes don’t need noexcept specification in move for this scenario. But it is a good idiomatic practice.

vector: Use **noexcept move** constructor for any object

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
```

Temporary Objects in Common STL types and Containers

4/7

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 removes the need for temporary objects and instead does in-place construction.

std::pair

```
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 auto pa = std::make_pair(A{1, 1}, A{2, 2});
}
```

make_pair ensures move construction instead of copy construction.

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

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

Also has same output with move construction.

To remove the extra objects and do in-place construction, we need to use ***std::piecewise_construct***.

```
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()
```

std::tuple

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

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

```
int main() {
  const auto t = std::make_tuple(A{1, 1}, A{2, 2});
}
```

make_tuple ensures move construction instead of copy construction.

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

```
int main() {
  const std::tuple t{A{1, 1}, A{2, 2}};
}
```

Also has same output with move construction.

There is no in-place construction for tuple. So, move instead of copy constructor is the best we can do.

This omission was discussed in this stackoverflow post.

std::optional

```
int main() {
  // std::optional<A> oa(10, 10); // Compilation ERROR.
  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()
```

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

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

Use **in_place_t** constructor or **make_optional** to do “in-place” construction.

std::optional: Deferred creation

```
int main() {
  std::optional<A> oa;
  puts("===== Before assign ======");
  oa = A{1, 1};
  puts("===== After assign ======");
}
```

```
===== Before assign ======
A(int, int)
A(A&&)
~A()
===== After assign ======
~A()
```

```
int main() {
  std::optional<A> oa;
  puts("===== Before emplace ======");
  oa.emplace(1, 1);
  puts("===== After emplace ======");
}
```

```
===== Before emplace ======
A(int, int)
===== After emplace ======
~A()
```

emplace is better than assignment.

```
int main() {
  std::optional<A> oa;
  puts("===== Before emplace ======");
  oa.emplace(1, 1);
  puts("===== Before 2nd emplace ======");
  oa.emplace(2, 2);
  puts("===== After 2nd emplace ======");
}
```

```
===== Before emplace ======
A(int, int)
===== Before 2nd emplace ======
~A()
A(int, int)
===== After 2nd emplace ======
~A()
```

emplace destroys current object, before in-place construction.

std::expected

```
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()
```

in_place constructor removes the temporary object.

For a non-trivial type being used in "success" type of **`std::expected`**, use **`in_place_t`** constructor to create the object in place.

std::unexpected

Error type of “std::expected”:

```
struct Error final {
  Error(int, int) { puts("Error(int, int)"); }

  ~Error() { puts("~Error()"); }

  Error(const Error&) { puts("Error(const Error&)"); }
  Error(Error&&) noexcept { puts("Error(Error&&)"); }

  Error& operator=(const Error&) {
    puts("Error& operator=(const Error&)");
    return *this;
  }

  Error& operator=(Error&&) noexcept {
    puts("Error& operator=(ErrorA&&)");
    return *this;
  }
};
```

```
std::expected<int, Error> Unexpected() {
  return std::unexpected{Error{10, 10}};
}

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

```
Error(int, int)
Error(Error&&)
Error(Error&&)
~Error()
~Error()
~Error()
```

```
std::expected<int, Error> Unexpected() {
  return std::unexpected<Error>{std::in_place, 10, 10};
}

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

```
Error(int, int)
Error(Error&&)
~Error()
~Error()
```

```
std::expected<int, Error> Unexpected() {
  return std::expected<int, Error>{std::unexpect, 10, 10};
}

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

```
Error(int, int)
~Error()
```

std::unexpect constructor removes the temporary object.

For a non-trivial type being used in "error" type of **`std::expected`**, use **`unexpect_t`** constructor to create the error object in place.

std::variant

```
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()
```

```
int main() {
  std::variant<A, int> v{std::in_place_index<0>, 10, 10};
}
```

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

`std::in_place_type` or **`std::in_place_index`** constructor removes the temporary object.

For a non-trivial type being used in **`variant`**, use **`std::in_place_type`** or **`std::in_place_index`** constructor to create the object in place.

std::variant: Changing value type

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

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

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

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

`emplace` is better for changing types for a variant.

For a non-trivial type being used in **`variant`**, use **`emplace`** to change the object type contained in the **`variant`**.

std::variant: Changing value of existing type

```cpp
int main() {
  std::variant<A, int> v{std::in_place_type<A>, 0, 0};
  std::cout << "------ Before assignment ------\n";
  v = A{10, 10};
  std::cout << "------ After assignment ------\n";
}
```

```
A(int, int)
------ Before assignment ------
A(int, int)
A& operator=(A&&)
~A()
------ After assignment ------
~A()
```

```cpp
int main() {
  std::variant<A, int> v{std::in_place_type<A>, 0, 0};
  std::cout << "------ Before emplace ------\n";
  v.emplace<A>(10, 10);
  std::cout << "------ After emplace ------\n";
}
```

```
A(int, int)
------ Before emplace ------
~A()
A(int, int)
------ After emplace ------
~A()
```

`emplace` causes the destructor of the object contained inside std::variant to always get called.

`emplace` performs better assignment in this non-type changing scenario too.

std::to_array

```
int main() {
  const auto arr = std::to_array<A>({{1, 2}, {3, 4}});
}
```

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

```
int main() {
  const std::array<A, 2> arr = {A{5, 6}, A{7, 8}};
}
```

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

```
int main() {
  // Uses deduction guide.
  const std::array arr = {A{5, 6}, A{7, 8}};
}
```

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

To create a **std::array** of *non-trivial* objects, consider using **std::array** constructor instead of **std::to_array**.

Adding elements to vector

```
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() {
  A a(10);
  A copy_a = a;
  A move_a = std::move(a);
}
```

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

Adding elements to vector

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

```
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(0)
A(1)
~A()
~A()
```

For `vector`, `emplace_back`, allows in-place construction.

Use **`emplace_back`** instead of **`push_back`**

std::vector: Use emplace_back

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

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

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(
```

This check also works for ***std::stack***, ***std::queue***, ***std::deque***, ***std::forward_list***, ***std::list***, ***std::priority_queue***.

Use emplace* functions for in-place construction

- **std::deque**: Use ***emplace_back/emplace_front*** instead of ***push_back/push_front***
- **std::forward_list**: Use ***emplace_after/emplace_front*** instead of ***insert_after/push_front***
- **std::list**: Use ***emplace_back/emplace_front/emplace*** instead of ***push_back/push_front/insert***
- **std::stack/std::queue**: Use ***emplace*** instead of ***push***
- **std::set**: Use ***emplace*** instead of ***insert***

Preventing Temporary Objects in Associative Containers

5/7

For `map`, use `emplace` instead of `operator[]`

```
int main() {
  std::map<std::string, int> m;
  // This is insertion. It creates a string and moves that into place.
  m["hello"] = 10;
}
```

```
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()
```

emplace does in-place construction.

In this case the non-trivial object is the “key”.

For **map**, use **emplace** instead of **operator[]**

Let’s consider the case where the non-trivial object is value instead of key.

```cpp
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;
};
```

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

For `map`, use `emplace` instead of `operator[]`

Let’s consider the case where the non-trivial object is value instead of key.

```
struct A final {
  A() { puts("A()"); }
  A(int a) : a_(a) { printf("A(%d)\n", a_); }

  // Special member functions.
};
```

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

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

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

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

Here **emplace** does in-place construction.

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

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

try_emplace also does the same for “value” object.

Special case for **emplace** versus **try_emplace**

Let's consider the case where "non-trivial" type is the key and check behavior difference between **emplace** and **try_emplace**.

```
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()
```

When key type is non-trivial object, then **only emplace** allows in-place construction.

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

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

This paper 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);
```

This will allow **try_emplace** to behave same as **emplace** in this scenario and will also let **operator[]** to construct in-place.

`emplace` for constructors with multiple arguments

```
struct B final {
  // Needed for operator[].
  B() { std::cout << "B(): " << GetStr() << '\n'; }
  B(int i, int j) : v_(i, j) { std::cout << "B(i, j): " << GetStr() << '\n'; }
  ~B() { puts("~B()"); }
  B(const B& rhs) : v_(rhs.v_) {
    std::cout << "B(const B&): " << GetStr() << '\n';
  }
  B(B&& rhs) noexcept : v_(std::move(rhs.v_)) {
    std::cout << "B(B&&): " << GetStr() << '\n';
  }
  B& operator=(const B& rhs) {
    v_ = rhs.v_;
    std::cout << "B& operator=(const B&): " << GetStr() << '\n';
    return *this;
  }
  B& operator=(B&& rhs) noexcept {
    v_ = std::move(rhs.v_);
    std::cout << "B& operator=(B&&): " << GetStr() << '\n';
    return *this;
  }
  std::string GetStr() const {
    std::stringstream ss;
    ss << "(" << v_.first << ", " << v_.second << ")";
    return ss.str();
  }
  auto operator<=>(const B&) const noexcept = default;

  std::pair<int, int> v_;
};
```

```
std::map<B, B> m;
m.emplace(10, 10, 20, 20); // COMPILATION ERROR.
```

```
int main() {
  std::map<B, B> m;
  m.emplace(B{10, 10}, B{20, 20});
}
```

```
B(i, j): (10, 10)
B(i, j): (20, 20)
B(B&&): (10, 10)
B(B&&): (20, 20)
~B()
~B()
~B()
~B()
```

```
int main() {
  std::map<B, B> m;
  m.emplace(std::piecewise_construct,
    std::forward_as_tuple(10, 10),
    std::forward_as_tuple(20, 20));
}
```

```
B(i, j): (10, 10)
B(i, j): (20, 20)
~B()
~B()
```

This approach calls the **std::piecewise_construct** constructor of **std::pair** and gets in-place construction.

```
int main() {
  std::map<B, B> m;
  m.try_emplace(B{10, 10}, 20, 20);
}
```

```
B(i, j): (10, 10)
B(B&&): (10, 10)
B(i, j): (20, 20)
~B()
~B()
~B()
```

`emplace/try_emplace` for case of existing key.

```
int main() {
  std::map<int, A> m;
  std::cout << "===== Before emplace(10, 20) ====\n";
  m.emplace(10, 20);
  std::cout << "===== Before emplace(10, 30) ====\n";
  m.emplace(10, 30);
  std::cout << "===== Before try_emplace(10, 40) ====\n";
  m.try_emplace(10, 40);
  std::cout << "===== After try_emplace(10, 40) ====\n";
}
```

```
===== Before emplace(10, 20) ====
A(20)
===== Before emplace(10, 30) ====
===== Before try_emplace(10, 40) ====
===== After try_emplace(10, 40) ====
~A()
```

For an existing key, the **value type object is not created**. However, this is not guaranteed by **emplace** specification. **try_emplace** guarantees that behavior.

insert_or_assign instead of operator[].

```
template <const char* name>
struct Type {
  Type() { printf("%s(): (%d, %d)\n", name, i, j); }
  Type(int i, int j) : i(i), j(j) {
    printf("%s(i, j): (%d, %d)\n", name, i, j);
  }
  ~Type() { printf("~%s()\n", name); }

  Type(const Type& rhs) : i(rhs.i), j(rhs.j) {
    printf("%s(const %s&): (%d, %d)\n", name, name, i, j);
  }
  Type(Type&& rhs) noexcept : i(rhs.i), j(rhs.j) {
    printf("%s(%s&&): (%d, %d)\n", name, name, i, j);
  }
  Type& operator=(const Type& rhs) {
    i = rhs.i;
    j = rhs.j;
    printf("%s& operator=(const %s&): (%d, %d)\n", name, name, i, j);
    return *this;
  }
  Type& operator=(Type&& rhs) noexcept {
    i = std::move(rhs.i);
    j = std::move(rhs.j);
    printf("%s& operator=(%s&&): (%d, %d)\n", name, name, i, j);
    return *this;
  }
  auto operator<=>(const Type&) const noexcept = default;
  int i = 0, j = 0;
};

static constexpr char kKey[] = "Key";
static constexpr char kValue[] = "Value";
using Key = Type<kKey>;
using Value = Type<kValue>;
```

```
int main() {
  Key key;
  Key key2(20, 20);
  Value val;
  Value val2(30, 30);
}
```

```
Key(): (0, 0)
Key(i, j): (20, 20)
Value(): (0, 0)
Value(i, j): (30, 30)
~Value()
~Value()
~Key()
~Key()
```

This creates two types, Key and Value, so we can see when the map key and value elements are created.

insert_or_assign instead of operator[].

```
int main() {
  std::map<Key, Value> m;
  Key k{10, 10};
  std::cout << "---- Before 1st[] ----\n";
  m[k] = Value{20, 20};
  std::cout << "---- Before 2nd[] ----\n";
  m[k] = Value{30, 30};
  std::cout << "---- After [] ----\n";
}
```

```
Key(i, j): (10, 10)
---- Before 1st[] ----
Value(i, j): (20, 20)
Key(const Key&): (10, 10)
Value(): (0, 0)
Value& operator=(Value&&): (20, 20)
~Value()
---- Before 2nd[] ----
Value(i, j): (30, 30)
Value& operator=(Value&&): (30, 30)
~Value()
---- After [] ----
~Key()
~Value()
~Key()
```

```
int main() {
  std::map<Key, Value> m;
  Key k{10, 10};
  std::cout << "---- Before 1st ----\n";
  m.insert_or_assign(k, Value{20, 20});
  std::cout << "---- Before 2nd ----\n";
  m.insert_or_assign(k, Value{30, 30});
  std::cout << "---- After both ----\n";
}
```

```
Key(i, j): (10, 10)
---- Before 1st ----
Value(i, j): (20, 20)
Key(const Key&): (10, 10)
Value(Value&&): (20, 20)
~Value()
---- Before 2nd ----
Value(i, j): (30, 30)
Value& operator=(Value&&): (30, 30)
~Value()
---- After both ----
~Key()
~Value()
~Key()
```

Using **insert_or_assign** improved the insertion case.

If a single function does both “insert” and “assign” use **insert_or_assign** instead of **operator[]**.

Using Transparent Comparators to Avoid Extra Objects

6/7

std::set<std::string>

```
void* operator new(size_t n) {
  void* p = malloc(n);
  printf("operator new: n: %zu, p = %p\n", n, p);
  return p;
}

void operator delete(void* p) noexcept {
  printf("operator delete: p = %p\n", p);
  free(p);
}
```

We will use this overridden **operator new / delete** to “detect” string creations.

We will check the methods ***count***, ***contains*** and ***find***.

```
int main() {
  std::set<std::string> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
find: Not found
```

std::set<std::string>

```cpp
int main() {
  std::set<std::string> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
find: Not found
```

std::set<std::string>

```cpp
int main() {
  std::set<std::string> s;
  const std::string test_str{"Hello, do you contain this string?"};
  std::cout << "----- Using count -----\n";

  const auto n = s.count(test_str);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(test_str);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(test_str);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
operator new: n: 40, p = 0x57d53ad182a0
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
operator delete: p = 0x57d53ad182a0
```

As we see, if we don’t use **const std::string&**, the functions **count**, **contains**, **find** will create temporary objects.

This applies to all of **const char***, **const char [N]** as argument.

std::set<std::string>

```
int main() {
  std::set<std::string> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x5b138382f2b0
operator delete: p = 0x5b138382f2b0
find: Not found
```

std::set<std::string>: transparent comparator

```
int main() {
  std::set<std::string, std::less<>> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
```

std::set<std::string>: transparent comparator

```cpp
int main() {
  std::set<std::string, std::less<>> s;
  const std::string test_str{"Hello, do you contain this string?"};
  std::cout << "----- Using count -----\n";

  const auto n = s.count(test_str);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(test_str);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(test_str);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
operator new: n: 40, p = 0x5cc0a549e2a0
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
operator delete: p = 0x5cc0a549e2a0
```

```cpp
std::set<std::string> s;
```

Use std::less<> to get transparent comparison.

```cpp
std::set<std::string, std::less<>> s;
```

Transparent comparator

Transparent comparators allow comparisons to happen without the need to *create temporary objects* in certain scenarios.

They are identified by a typedef **`is_transparent`**.

Here's an example of **`std::less<void>`** specialization code from libc++ code.

```
template <>
struct _LIBCPP_TEMPLATE_VIS less<void> {
  template <class _T1, class _T2>
  _LIBCPP_CONSTEXPR_SINCE_CXX14 _LIBCPP_HIDE_FROM_ABI auto operator()(
      _T1&& __t,
      _T2&& __u) const
      noexcept(noexcept(std::forward<_T1>(__t) < std::forward<_T2>(__u)))  //
      -> decltype(std::forward<_T1>(__t) < std::forward<_T2>(__u)) {
    return std::forward<_T1>(__t) < std::forward<_T2>(__u);
  }
  typedef void is_transparent;
};
```

std::map<std::string, AnotherType>

```
int main() {
  std::map<std::string, int> m;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = m.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = m.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = m.find(kTestStr);
  if (it == m.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x5f5b4b9c82b0
operator delete: p = 0x5f5b4b9c82b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x5f5b4b9c82b0
operator delete: p = 0x5f5b4b9c82b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x5f5b4b9c82b0
operator delete: p = 0x5f5b4b9c82b0
find: Not found
```

std::map<std::string, AnotherType>: Transparent comparator

```cpp
int main() {
  std::map<std::string, int, std::less<>> m;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = m.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = m.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = m.find(kTestStr);
  if (it == m.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
```

```cpp
std::map<std::string, int> m;
```

Use std::less<> to get transparent comparison.

```cpp
std::map<std::string, int, std::less<>> m;
```

std::unordered_set<std::string>

```
int main() {
  std::unordered_set<std::string> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x5bbcb3a372b0
operator delete: p = 0x5bbcb3a372b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x5bbcb3a372b0
operator delete: p = 0x5bbcb3a372b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x5bbcb3a372b0
operator delete: p = 0x5bbcb3a372b0
find: Not found
```

std::unordered_set<std::string>: Transparent comparator

```
int main() {
  std::unordered_set<std::string, MyStringHash, std::equal_to<>> s;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = s.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = s.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = s.find(kTestStr);
  if (it == s.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
```

```
struct MyStringHash final {
  using hash_type = std::hash<std::string_view>;
  using is_transparent = void;

  std::size_t operator()(const char* str) const {
     return hash_type{}(str);
  }
  std::size_t operator()(std::string_view str) const {
    return hash_type{}(str);
  }
  std::size_t operator()(std::string const& str) const {
    return hash_type{}(str);
  }
};
```

```
std::unordered_set<std::string> s;
```

Use MyStringHash and std::equal_to to get transparent comparison.

```
std::unordered_set<std::string, MyStringHash, std::equal_to<>> s;
```

std::unordered_map<std::string, AnotherType>

```
int main() {
  std::unordered_map<std::string, int> m;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = m.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = m.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = m.find(kTestStr);
  if (it == m.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
operator new: n: 40, p = 0x596aeaadf2b0
operator delete: p = 0x596aeaadf2b0
Count: 0
----- Using contains -----
operator new: n: 40, p = 0x596aeaadf2b0
operator delete: p = 0x596aeaadf2b0
contains: Not found
----- Using find -----
operator new: n: 40, p = 0x596aeaadf2b0
operator delete: p = 0x596aeaadf2b0
find: Not found
```

unordered_map<string, OtherType> : Transparent comparator

```cpp
int main() {
  std::unordered_map<std::string, int, MyStringHash, std::equal_to<>> m;
  constexpr char kTestStr[] = "Hello, do you contain this string?";
  std::cout << "----- Using count -----\n";

  const auto n = m.count(kTestStr);

  std::cout << "Count: " << n << std::endl;
  std::cout << "----- Using contains -----\n";

  const auto found = m.contains(kTestStr);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "----- Using find -----\n";

  auto it = m.find(kTestStr);
  if (it == m.end()) {
    std::cout << "find: Not found\n";
  }
}
```

```
----- Using count -----
Count: 0
----- Using contains -----
contains: Not found
----- Using find -----
find: Not found
```

```cpp
std::unordered_map<std::string, int> m;
```

Use MyStringHash and std::equal_to to get transparent comparison.

```cpp
std::unordered_map<std::string, int, MyStringHash, std::equal_to<>> m;
```

Use transparent comparators for std::string for associative containers

```
std::set<std::string> s;
```

```
std::set<std::string, std::less<>> s;
```

```
std::map<std::string, int> m;
```

```
std::map<std::string, int, std::less<>> m;
```

```
std::unordered_set<std::string> s;
```

```
std::unordered_set<std::string, MyStringHash, std::equal_to<>> s;
```

```
std::unordered_map<std::string, int> m;
```

```
std::unordered_map<std::string, int, MyStringHash, std::equal_to<>> m;
```

Transparent comparator for user-defined types

```
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;
  }

  auto operator<=>(const A&) const noexcept = default;
  int a_ = 0;
};
```

This is needed for code to compile for A to be used as key in *set*: **std::set<A>**

std::set<UserType>

```
int main() {
  std::set<A> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

We will check the methods ***count***, ***contains*** and ***find.***

std::set<UserType>: Transparent comparator

```cpp
int main() {
  std::set<A, std::less<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
A(10)
~A()
Not Found
------ After find ------
~A()
```

More objects are getting created with transparent comparator.

std::set<UserType>: Transparent comparator

```cpp
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;
  }

  auto operator<=>(const A&) const noexcept = default;

  auto operator<=>(int i) const noexcept { return a_ <=> i; }

  int a_ = 0;
};
```

This allows ***A*** to compare with ***int***.

std::set<UserType>: Transparent comparator

```cpp
int main() {
  std::set<A, std::less<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
A(10)
~A()
Not Found
------ After find ------
~A()
```

More objects are getting created with transparent comparator.

std::set<UserType>: Transparent comparator

```cpp
int main() {
  std::set<A, std::less<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
Count: 0
------ Using contains ------
contains: Not found
------ Using find ------
Not Found
------ After find ------
~A()
```

With the operator <=> (int):

```cpp
struct A final {
  // << snipped >>
  auto operator<=>(const A&) const noexcept = default;

  auto operator<=>(int i) const noexcept { return a_ <=> i; }
  // << snipped >>
};
```

std::map<UserType, T>

```cpp
int main() {
  std::map<A, int> m;
  m.emplace(20, 20);
  std::cout << "------ Using count ------\n";

  const auto n = m.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = m.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = m.find(10);
  if (it == m.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

std::map<UserType, T>: Transparent comparator

```cpp
int main() {
  std::map<A, int, std::less<>> m;
  m.emplace(20, 20);
  std::cout << "------ Using count ------\n";

  const auto n = m.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = m.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = m.find(10);
  if (it == m.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
Count: 0
------ Using contains ------
contains: Not found
------ Using find ------
Not Found
------ After find ------
~A()
```

std::unordered_set<UserType>

```cpp
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;
  }

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

  int a_ = 0;
};

template <>
struct std::hash<A> {
  std::size_t operator()(const A& a) const noexcept {
    return std::hash<int>{}(a.a_);
  }
};
```

These are necessary for code to compile for A to be used as key in *unordered_set*: **std::unordered_set<A>**

std::unordered_set<UserType>

```cpp
int main() {
  std::unordered_set<A> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

std::unordered_set<UserType>: Transparent comparator

```cpp
int main() {
  std::unordered_set<A, std::hash<A>, std::equal_to<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

std::unordered_set<UserType>: Transparent comparator

```
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;
  }

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

  int a_ = 0;
};
```

```
template <>
struct std::hash<A> {
  std::size_t operator()(const A& a) const noexcept {
    return std::hash<int>{}(a.a_);
  }
};
```

std::unordered_set<UserType>: Transparent comparator

```cpp
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;
  }

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

  bool operator==(int i) const noexcept { return a_ == i; }

  int a_ = 0;
};
```

```cpp
template <>
struct std::hash<A> {
  std::size_t operator()(const A& a) const noexcept {
    return std::hash<int>{}(a.a_);
  }
  std::size_t operator()(int i) const noexcept {
    return std::hash<int>{}(i);
  }
  using is_transparent = void;
};
```

These are needed to ensure “transparent operators” work properly for user defined types as “key” of **std::unordered_set**.

std::unordered_set<UserType>: Transparent comparator

```
int main() {
  std::unordered_set<A, std::hash<A>, std::equal_to<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

std::unordered_set<UserType>: Transparent comparator

```cpp
int main() {
  std::unordered_set<A, std::hash<A>, std::equal_to<>> s;
  s.emplace(20);
  std::cout << "------ Using count ------\n";

  const auto n = s.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = s.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = s.find(10);
  if (it == s.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
Count: 0
------ Using contains ------
contains: Not found
------ Using find ------
Not Found
------ After find ------
~A()
```

```cpp
struct A final {
  // << snipped >>
  bool operator==(const A&) const noexcept = default;

  bool operator==(int i) const noexcept { return a_ == i; }
  // << snipped >>
};
```

```cpp
template <>
struct std::hash<A> {
  std::size_t operator()(const A& a) const noexcept {
    return std::hash<int>{}(a.a_);
  }
  std::size_t operator()(int i) const noexcept {
    return std::hash<int>{}(i);
  }
  using is_transparent = void;
};
```

std::unordered_map<UserType, T>

```cpp
int main() {
  std::unordered_map<A, int> m;
  m.emplace(20, 20);
  std::cout << "------ Using count ------\n";

  const auto n = m.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = m.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = m.find(10);
  if (it == m.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
A(10)
~A()
Count: 0
------ Using contains ------
A(10)
~A()
contains: Not found
------ Using find ------
A(10)
~A()
Not Found
------ After find ------
~A()
```

std::unordered_map<UserType, T>: Transparent comparator

```
int main() {
  std::unordered_map<A, int, std::hash<A>, std::equal_to<>> m;
  m.emplace(20, 20);
  std::cout << "------ Using count ------\n";

  const auto n = m.count(10);

  std::cout << "Count: " << n << '\n';
  std::cout << "------ Using contains ------\n";

  const bool found = m.contains(10);

  if (!found) {
    std::cout << "contains: Not found\n";
  }
  std::cout << "------ Using find ------\n";

  auto it = m.find(10);
  if (it == m.end()) {
    std::cout << "Not Found\n";
  }
  std::cout << "------ After find ------\n";
}
```

```
A(20)
------ Using count ------
Count: 0
------ Using contains ------
contains: Not found
------ Using find ------
Not Found
------ After find ------
~A()
```

Moving Data to Compile Time

7/7

Why move to compile time?

- Data is processed during compilation
- Objects don’t need to be constructed at runtime

Move “const” data structures to compile time if possible

Runtime const std::string / std::vector:

```
int main() {
  const std::string str("hello");
  const std::vector arr{1, 2, 3};
}
```

Can be moved to compile time using std::string_view / std::array:

```
int main() {
  static constexpr std::string_view kStr("hello");
  static constexpr char kStrArr[] = "hello";
  static constexpr int kCArr[] = {1, 2, 3};
  static constexpr auto kArr = std::to_array({1, 2, 3});
}
```

Move “const” data structures to compile time if possible

Global const std::string / std::vector:

```
const std::string global_str("this is a really long string");
const std::vector global_arr{1, 2, 3};

int main() {}
```

When built with ***-Wglobal-constructors -Wexit-time-destructors***

```
error: declaration requires an exit-time destructor [-Werror,-Wexit-time-destructors]
 const std::string global_str("this is a really long string");
                   ^
error: declaration requires a global destructor [-Werror,-Wglobal-constructors]
error: declaration requires an exit-time destructor [-Werror,-Wexit-time-destructors]
 const std::vector global_arr{1, 2, 3};
                   ^
error: declaration requires a global destructor [-Werror,-Wglobal-constructors]
```

These flags help in figuring out opportunities for moving global const objects to compile time.

```
constexpr std::string_view kStr("this is a really long string");
constexpr std::array kArr{1, 2, 3};

int main() {}
```

✓

Move “const” data structures to compile time if possible

Global const std::string / std::vector:

```
const std::string global_str("this is a really long string");
const std::vector global_arr{1, 2, 3};

int main() {}
```

```
const std::string& GetStr() {
  static const std::string global_str("this is a really long string");
  return global_str;
}

const std::vector<int>& GetVec() {
  static const std::vector global_arr{1, 2, 3};
  return global_arr;
}

int main() {}
```

-Wexit-time-destructors also points out magic statics.

```
error: declaration requires an exit-time destructor [-Werror,-Wexit-time-destructors]
   static const std::string global_str("this is a really long string");
                            ^
error: declaration requires an exit-time destructor [-Werror,-Wexit-time-destructors]
   static const std::vector global_arr{1, 2, 3};
                            ^
```

Move “const” data structures to compile time if possible

Transformation may need changes to usage interface.

```
// Header
const std::vector<int>& GetVec();
```

```
// Source file.
namespace {
const std::vector global_arr{1, 2, 3};
}  // namespace

const std::vector<int>& GetVec() {
  return global_arr;
}
```

```
// Header
std::span<const int> GetVec();
```

```
// Source file.
namespace {
constexpr std::array kArr{1, 2, 3};
}  // namespace

std::span<const int> GetVec() {
  return kArr;
}
```

Moving User defined “const” data structures to compile time

```
// Header.
bool ContainsStr1(const std::string& str);
```

```
// Source file
namespace {
struct SomeClass {
  std::string str1;
  std::string str2;
  int value;
};

const std::vector<SomeClass> global_arr{{"one", "two", 12}, {"three", "four", 34}};
}  // namespace

bool ContainsStr1(const std::string& str) {
  return std::ranges::any_of(
      global_arr, [&str](const auto& str1) { return str1 == str; },
      &SomeClass::str1);
}
```

```
error: declaration requires an exit-time destructor [-Werror,-Wexit-time-destructors]
 const std::vector<SomeClass> global_arr{{"one", "two", 12}, {"three", "four", 34}};
                                ^
error: declaration requires a global destructor [-Werror,-Wglobal-constructors]
```

Moving User defined “const” data structures to compile time

```
// Header.
bool ContainsStr1(const std::string& str);
```

```
// Source file
namespace {
struct SomeClass {
  std::string str1;
  std::string str2;
  int value;
};

const std::vector<SomeClass>& GetArr() {
  static const auto* global_arr = new std::vector<SomeClass>{{"one", "two", 12}, {"three", "four", 34}};
  return *global_arr;
}
}  // namespace

bool ContainsStr1(const std::string& str) {
  return std::ranges::any_of(
      GetArr(), [&str](const auto& str1) { return str1 == str; },
      &SomeClass::str1);
}
```

absl::NoDestructor can also better “annotate” this “will not delete” scenario.

Moving User defined “const” data structures to compile time

```
// Header.
bool ContainsStr1(std::string_view str);
```

```
// Source file
namespace {
struct SomeClass {
  const std::string_view str1;
  const std::string_view str2;
  const int value;
};

constexpr auto kArr =
    std::to_array<SomeClass>({{"one", "two", 12}, {"three", "four", 34}});
}  // namespace

bool ContainsStr1(std::string_view str) {
  return std::ranges::any_of(
      kArr, [str](auto str1) { return str1 == str; }, &SomeClass::str1);
}
```

In our code base, we have seen quite a few instances where user defined data structures could be converted to compile time.

Concatenating string at compile time

```
#include "mystrcat.h"

int main() {
  constexpr std::string_view kHello = "Hello, ";
  constexpr std::string_view kWorld = "World!!";
  const std::string result = MyStrCat({kHello, kWorld});
  std::cout << result << std::endl;
}
```

```
Hello, World!!
```

This is a runtime string created which will always be the same.

MyCompileTimeStringJoiner can be used to do compile time concatenation.

```
#include "my_compile_time_string_joiner.h"

int main() {
  static constexpr std::string_view kHello = "Hello, ";
  static constexpr std::string_view kWorld = "World!!";
  static constexpr std::string_view kJoinResultStr =
      MyCompileTimeStringJoinerV<kHello, kWorld>;
  static_assert(kJoinResultStr == "Hello, World!!");
  std::cout << kJoinResultStr << std::endl;
}
```

```
Hello, World!!
```

Concatenating string at compile time

```
#include "my_compile_time_string_joiner.h"

int main() {
  static constexpr std::string_view kHello = "Hello, ";
  static constexpr std::string_view kWorld = "World!!";
  static constexpr std::string_view kJoinResultStr =
      MyCompileTimeStringJoinerV<kHello, kWorld>;
  static_assert(kJoinResultStr == "Hello, World!!");
  std::cout << kJoinResultStr << std::endl;
}
```

```
Hello, World!!
```

```
template <std::string_view const&... Strs>
struct MyCompileTimeStringJoiner final {
  // Join all strings into a single std::array of chars.
  static constexpr auto JoinImpl() noexcept {
    constexpr std::size_t len = (Strs.size() + ... + 0);
    std::array<char, len + 1> arr{};
    auto append = [i = 0, &arr](auto const s) mutable {
      for (auto c : s) {
        arr[i++] = c;
      }
    };
    (append(Strs), ...);
    arr[len] = 0;
    return arr;
  }
  // Give the joined string static storage.
  static constexpr auto kJoinedArray = JoinImpl();
  // View as a std::string_view.
  static constexpr std::string_view kJoinedArrayAsStringView = {
      kJoinedArray.data(), kJoinedArray.size() - 1};
};
// Helper to get the value out.
template <std::string_view const&... Strs>
static constexpr auto MyCompileTimeStringJoinerV =
    MyCompileTimeStringJoiner<Strs...>::kJoinedArrayAsStringView;
```

Copied from https://stackoverflow.com/a/62823211

Also check this C++ On Sea 2024 session by Jason Turner for approaches to create compile time strings.

Global std::set, std::map: Can they move to compile time?

```
std::set<std::string> global_set{"one", "two", "three"};

std::map<std::string, int, std::less<>> global_map{{"one", 1}, {"two", 2}};
```

There is no standard compliant way to do this.

```
constexpr auto kSet =
    base::MakeFixedFlatSet<std::string_view>({"one", "two", "three"});

constexpr auto kMap = base::MakeFixedFlatMap<std::string_view, int>(
    {{"one", 1}, {"two", 2}, {"three", 3}});
```

Chromium has fixed_flat_map and fixed_flat_set which can be used to create compile time set / map equivalents.

std::flat_map and std::flat_set will be made constexpr in C++26. Once constexpr, some additional code can be written to create instances at compile time.

Conclusion

Key Points

- We want in-place construction without copies or moves.
- Pass non-trivial objects by reference
- Use view types (std::string_view, std::span)
- Use in-place constructors for STL types
- Use emplace
- Use transparent comparators for std::string in associative containers
- Move data to compile time
- Use clang-tidy checks and warnings

References

- CppCon 2024: How to Use string_view in C++ - Basics, Benefits, and Best Practices - Jasmine Lopez & Prithvi Okade.
- CppCon 2018: Jon Kalb “Copy Elision”.
- CppCon 2024: C++ RVO: Return Value Optimization for Performance in Bloomberg C++ Codebases - Michelle Fae D'Souza.
- CppCon 2017: Jorg Brown “The design of absl::StrCat...”
- Understanding The constexpr 2-Step - Jason Turner - C++ on Sea 2024.
- C++Now 2018: Jason Turner “Initializer Lists Are Broken, Let's Fix Them”.
- C++ Weekly - Ep 421 - You're Using optional, variant, pair, tuple, any, and expected Wrong!
- Why is there no piecewise tuple construction?
- CppCon 2018: Andrei Alexandrescu “Expect the expected”.
- In-Place Construction for std::any, std::variant and std::optional: Bartlomiej Filipek (www.cppstories.com).

References

- is_transparent: How to search a C++ set with another type than its key: Jonathan Boccara (www.fluentcpp.com).
- Overview of std::map's Insertion / Emplacement Methods in C++17 - Fluent C++
- c++ - How to concatenate static strings at compile time? - Stack Overflow
- P2363R3: Extending associative containers with the remaining heterogeneous overloads
- C++ Core Guidelines
- clang-tidy checks
- Diagnostic flags in Clang
- absl::StrCat
- absl::NoDestructor
- Chromium: fixed_flat_map.h, fixed_flat_set.h

Questions?

Appendix

StrCat: Another implementation

```
template <typename T>
auto GetLength(T&& elem) {
  using Type = std::decay_t<T>;
  constexpr auto IsCharPtr =
      std::is_same_v<Type, char*> || std::is_same_v<Type, const char*>;
  if constexpr (IsCharPtr) {
    return strlen(elem);
  } else {
    return std::size(std::forward<T>(elem));
  }
}

template <typename T1, typename... Args>
  requires std::is_constructible_v<std::string, T1&&> &&
           (std::is_constructible_v<std::string, Args &&> && ...)
std::string StrCat(T1&& first, Args&&... args) {
  // Determine final string length.
  const auto final_size = GetLength(std::forward<T1>(first)) +
                          (GetLength(std::forward<Args>(args)) + ... + 1u);
  std::string ret;
  ret.reserve(final_size);
  ret += std::forward<T1>(first);
  ((ret += std::forward<Args>(args)), ...);
  return ret;
}
```

```
int main() {
  char arr[] = "hello";
  char* p_arr = arr;
  const char* p_arr2 = " hey2 ";
  const std::string str(" folks!!");
  std::cout << StrCat(p_arr, p_arr2,
        std::string_view{"world!!"},
        std::string{", how are"}, " you", str)
        << '\n';
}
```

```
hello hey2 world!!, how are you folks!!
```

StrCat: Another implementation

```cpp
template <typename>
using StringViewType = std::string_view;

template <typename... Args>
std::string StrCatImpl(StringViewType<Args>... args) {
  // Determine final string length.
  const auto final_size = (args.size() + ... + 1u);
  std::string ret;
  ret.reserve(final_size);
  ((ret += args), ...);
  return ret;
}

template <typename... Args>
auto StrCat(Args&&... args)
    -> decltype(StrCatImpl<Args...>(std::forward<Args>(args)...)) {
  return StrCatImpl<Args...>(std::forward<Args>(args)...);
}
```

```cpp
int main() {
  char arr[] = "hello";
  char* p_arr = arr;
  const char* p_arr2 = " hey2 ";
  const std::string str(" folks!!");
  std::cout << StrCat(p_arr, p_arr2,
       std::string_view{"world!!"},
       std::string{", how are"}, " you", str)
       << '\n';
}
```

```
hello hey2 world!!, how are you folks!!
```

Create Vector function

```
template <typename... Args>
void TupleFunc(std::tuple<Args...>&&);

template <typename T>
concept IsTuple = requires(T t) { TupleFunc(std::move(t)); };

template <typename T, size_t... Index, typename... Args>
void AddToVector(std::vector<T>& vec,
                 std::integer_sequence<size_t, Index...> seq,
                 std::tuple<Args...>&& t) {
  vec.emplace_back(std::get<Index>(t)...);
}

template <typename T, typename... Args>
void AddToVector(std::vector<T>& vec, std::tuple<Args...>&& t) {
  constexpr auto N = sizeof...(Args);
  AddToVector(vec, std::make_index_sequence<N>{}, std::move(t));
}

template <typename T, typename... Args>
  requires(IsTuple<Args> && ...)
std::vector<T> CreateVector(Args&&... args) {
  std::vector<T> vec;
  vec.reserve(sizeof...(args));
  (AddToVector(vec, std::forward<Args>(args)), ...);
  return vec;
}
```

```
int main() {
  const auto a =
      CreateVector<A>(std::forward_as_tuple(1, 2),
                      std::forward_as_tuple(3, 4),
                      std::forward_as_tuple(5, 6));
  return a.size();
}
```

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

-Wlarge-by-value-copy

```cpp
#include <iostream>
#include <string>

struct A {
  int a, b, c, d;
  int e, f, g;
};

struct B {
  std::string a;
  std::string b;
};

void Foo(A) {}

void Foo(B) {}

int main() {
  std::cout << "sizeof(A): " << sizeof(A) << '\n';                   // 28
  std::cout << "sizeof(B): " << sizeof(B) << '\n';                   // 48
  std::cout << "sizeof(std::string): " << sizeof(std::string) << '\n';  // 24
  std::cout << "sizeof(std::string_view): " << sizeof(std::string_view)
            << '\n';  // 16
}
```

When compiled with ***-Wlarge-by-value-copy=24***
This flags sizes > 24.

```
error: '' is a large (28 bytes) pass-by-value
argument; pass it by reference instead ? [-Werror,-
Wlarge-by-value-copy]
 void Foo(A) {}
          ^
```

It only catches PODs, so ***does not*** catch ***Foo(B)***.

It is “not” on by default.

Compile time set

```
template <typename Range, typename Comp = std::less<>>
constexpr bool IsSortedAndUnique(const Range& range) {
  return std::ranges::adjacent_find(range, std::not_fn(Comp{})) ==
         std::ranges::end(range);
}

void TriggerCompileError(std::string_view);

template <typename Key, typename Value, size_t N>
class CompileTimeMap {
 public:
  using PairType = std::pair<Key, Value>;
  constexpr CompileTimeMap(std::array<PairType, N>&& data)
      : data_(std::move(data)) {
    std::ranges::sort(data_);
    if (!IsSortedAndUnique(data_)) {
      TriggerCompileError("Non-unique");
    }
  }
  constexpr CompileTimeMap(PairType (&&arr)[N])
      : data_(std::to_array(std::move(arr))) {
    std::ranges::sort(data_);
    if (!IsSortedAndUnique(data_)) {
      TriggerCompileError("Non-unique");
    }
  }

  constexpr bool Contains(const Key& elem) const {
    return std::ranges::binary_search(data_, elem, std::less<>{},
                                      &PairType::first);
  }

  constexpr std::span<const PairType> AsSpan() const { return data_; }

 private:
  std::array<PairType, N> data_;
};
```

```
int main() {
  static constexpr CompileTimeMap kCompMap(
      {std::pair<int, int>{3, 3}, std::pair<int, int>{1, 2}});
  for (const auto& [key, value] : kCompMap.AsSpan()) {
    std::cout << '(' << key << ", " << value << ") ";
  }
  std::cout << '\n';
  for (const auto i : {1, 2, 4}) {
    const auto found = kCompMap.Contains(i);
    std::cout << i << (found ? " found\n" : " not found\n");
  }
}
```

```
(1, 2) (3, 3)
1 found
2 not found
4 not found
```

Compile time map

```
template <typename Range, typename Comp = std::less<>>
constexpr bool IsSortedAndUnique(const Range& range) {
  return std::ranges::adjacent_find(range, std::not_fn(Comp{})) ==
         std::ranges::end(range);
}

void TriggerCompileError(std::string_view);

template <typename Key, typename Value, size_t N>
class CompileTimeMap {
 public:
  using PairType = std::pair<Key, Value>;
  constexpr CompileTimeMap(std::array<PairType, N>&& data)
      : data_(std::move(data)) {
    std::ranges::sort(data_);
    if (!IsSortedAndUnique(data_)) {
      TriggerCompileError("Non-unique");
    }
  }
  constexpr CompileTimeMap(PairType (&&arr)[N])
      : data_(std::to_array(std::move(arr))) {
    std::ranges::sort(data_);
    if (!IsSortedAndUnique(data_)) {
      TriggerCompileError("Non-unique");
    }
  }

  constexpr bool Contains(const Key& elem) const {
    return std::ranges::binary_search(data_, elem, std::less<>{},
                                      &PairType::first);
  }
```

```
  constexpr std::optional<Value> GetValue(const Key& elem) const {
    const auto [start, end] =
        std::ranges::equal_range(data_, elem, std::less<>{},
                                 &PairType::first);
    if (start == end) {
      // Not found.
      return std::nullopt;
    }
    return start->second;
  }

  constexpr std::span<const PairType> AsSpan() const { return data_; }

 private:
  std::array<PairType, N> data_;
};
```

```
int main() {
  static constexpr CompileTimeMap kCompMap(
      {std::pair<int, int>{3, 3}, std::pair<int, int>{1, 2}});
  for (const auto& [key, value] : kCompMap.AsSpan()) {
    std::cout << '(' << key << ", " << value << ") ";
  }
  std::cout << '\n';
  for (const auto i : {1, 2, 4}) {
    const auto found = kCompMap.Contains(i);
    std::cout << i << (found ? " found\n" : " not found\n");
  }
  for (const auto i : {1, 2, 4, 3}) {
    const auto value_opt = kCompMap.GetValue(i);
    if (!value_opt) {
      std::cout << i << " not found\n";
    } else {
      std::cout << i << " => " << *value_opt << '\n';
    }
  }
}
```

```
(1, 2) (3, 3)
1 found
2 not found
4 not found
1 => 2
2 not found
4 not found
3 => 3
```

Compile time map

```cpp
template <typename Key, typename Value, size_t N>
consteval auto MakeCompMap(std::pair<Key, Value> (&&arr)[N]) {
  return CompileTimeMap(std::move(arr));
}
```

```cpp
void TestIntIntMap() {
  static constexpr auto kCompMap = MakeCompMap<int, int>({{3, 3}, {1, 2}});
  for (const auto& [key, value] : kCompMap.AsSpan()) {
    std::cout << '(' << key << ", " << value << ") ";
  }
  std::cout << '\n';
  for (const auto i : {1, 2, 4}) {
    const auto found = kCompMap.Contains(i);
    std::cout << i << (found ? " found\n" : " not found\n");
  }
  for (const auto i : {1, 2, 4, 3}) {
    const auto value_opt = kCompMap.GetValue(i);
    if (!value_opt) {
      std::cout << i << " not found\n";
    } else {
      std::cout << i << " => " << *value_opt << '\n';
    }
  }
}
```

```cpp
void TestStringIntMap() {
  static constexpr auto kCompMap =
      MakeCompMap<std::string_view, int>({{"three", 3}, {"one", 2}});
  for (const auto& [key, value] : kCompMap.AsSpan()) {
    std::cout << '(' << key << ", " << value << ") ";
  }
  std::cout << '\n';
  for (const auto i : {"four", "five", "three"}) {
    const auto found = kCompMap.Contains(i);
    std::cout << i << (found ? " found\n" : " not found\n");
  }
  for (const auto i : {"four", "five", "three"}) {
    const auto value_opt = kCompMap.GetValue(i);
    if (!value_opt) {
      std::cout << i << " not found\n";
    } else {
      std::cout << i << " => " << *value_opt << '\n';
    }
  }
}
```

```cpp
int main() {
  TestIntIntMap();
  TestStringIntMap();
}
```

```
(1, 2) (3, 3)
1 found
2 not found
4 not found
1 => 2
2 not found
4 not found
3 => 3
(one, 2) (three, 3)
four not found
five not found
three found
four not found
five not found
three => 3
```