Back to Basics:
Designing Classes (part 2 of 2)

KLAUS IGLBERGER

CppCon
The C++ Conference
2021
October 24-29

C++ Trainer/Consultant

Author of the blaze C++ math library

(Co-)Organizer of the Munich C++ user group

Chair of the CppCon B2B and SD tracks

Email: klaus.iglberger@gmx.de

Klaus Iglberger

Content

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Implementation Guidelines

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Data Member Initialization

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Data Member Initialization

Interactive Task: What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{

   int i;             // Uninitialized
   std::string s;     // Default (i.e. empty string)
   int* pi;           // Uninitialized
};

int main()
{
   Widget w;          // Default initialization
}
```

Data Member Initialization

The compiler generated default constructor …

- initializes all data members of class (user-defined) type …
- but not the data members of fundamental type.

```
struct Widget
{

   int i;          // Uninitialized
   std::string s;  // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w;       // Default initialization: Calls
}                  // the default constructor
```

Data Member Initialization

Interactive Task: What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{

   int i;            // Initialized to 0
   std::string s;    // Default (i.e. empty string)
   int* pi;          // Initialized to nullptr
};

int main()
{
   Widget w{};      // Value initialization
}
```

Data Member Initialization

If no default constructor is declared, value initialization …

- zero-initializes the object
- and then default-initializes all non-trivial data members.

```
struct Widget
{
   int i;          // Initialized to 0
   std::string s;  // Default (i.e. empty string)
   int* pi;        // Initialized to nullptr
};

int main()
{
   Widget w{};    // Value initialization: No default
}                 // ctor -> zero+default init
```

Data Member Initialization

Guideline: Prefer to create default objects by means of an empty set of braces (value initialization).

Data Member Initialization

Interactive Task: What is the initial value of the three data members `i`, `s`, and `pi`?

```
struct Widget
{
   Widget() {}     // Explicit default constructor
   int i;          // Uninitialized
   std::string s; // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w{};    // Value initialization
}
```

Data Member Initialization

An empty default constructor …

- initializes all data members of class (user-defined) type …
- but not the data members of fundamental type.

```
struct Widget
{
   Widget() {}     // Explicit default constructor
   int i;          // Uninitialized
   std::string s;  // Default (i.e. empty string)
   int* pi;        // Uninitialized
};

int main()
{
   Widget w{};    // Value initialization: Declared
}                 // default ctor -> calls ctor
```

Data Member Initialization

Guideline: Avoid writing an empty default constructor.

Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
  {
    i  = 42;         // Initialize the int to 42
    s  = "CppCon";   // Initialize the string to “CppCon"
    pi = nullptr;    // Initialize the pointer to nullptr
  }


  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
  {
    i  = 42;         // Assignment, not initialization
    s  = "CppCon";   // Assignment, not initialization
    pi = nullptr;    // Assignment, not initialization
  }


  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
    : s{}  // Initialization happens in the
           // member initializer list
  {
    i  = 42;        // Assignment, not initialization
    s  = "CppCon";  // Assignment, not initialization
    pi = nullptr;   // Assignment, not initialization
  }

  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
    : s{"CppCon"}   // Initialization of the string
                    // in the member initializer list
  {
    i  = 42;        // Assignment, not initialization

    pi = nullptr;   // Assignment, not initialization
  }

  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Via the default constructor, we can properly initialize all data members:

```
struct Widget
{
  Widget()
    : i {42}          // Initializing to 42
    , s {"CppCon"}    // Initializing to "CppCon"
    , pi{}            // Initializing to nullptr
  {}



  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Core Guideline C.47: Define and initialise member variables in the order of member declaration

Core Guideline C.49: Prefer initialization to assignment in constructors.

Data Member Initialization

Let's assume that a colleague adds another constructor…

```
struct Widget
{
   Widget()
     : i {42}          // Initializing to 42
     , s {"CppCon"}    // Initializing to "CppCon"
     , pi{}            // Initializing to nullptr
   {}

   Widget( int j )
     : i {j}           // Initialization to j
   {}



   int i;
   std::string s;
   int* pi;
};
```

Data Member Initialization

Let's assume that a colleague adds another constructor…

```
struct Widget
{
  Widget()
    : i {42}          // Initializing to 42
    , s {"CppCon"}    // Initializing to "CppCon"
    , pi{}            // Initializing to nullptr
  {}

  Widget( int j )
    : i {j}           // Initialization to j
    , s {"CppCon"}    // Initialization to "CppCon"
    , pi{}            // Initialization to nullptr
  {}

  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Let’s assume that a colleague adds another constructor…

```
struct Widget
{
  Widget()
    : i {42}          // Initializing to 42
    , s {"CppCon"}    // Initializing to "CppCon"
    , pi{}            // Initializing to nullptr
  {}

  Widget( int j )
    : i {j}           // Initialization to j
    , s {"CppCon"}    // Initialization to “CppCon" (duplication)
    , pi{}            // Initialization to nullptr (duplication)
  {}

  int i;
  std::string s;
  int* pi;
};
```

Data Member Initialization

Guideline: Avoid duplication to enable you to change everything in one place (the DRY principle).

Guideline: Design classes for easy change.

Data Member Initialization

In order to reduce duplication, we could use delegating constructors …

```
struct Widget
{
  Widget()
    : Widget(42)  // Delegating constructor
  {}



  Widget( int j )
    : i {j}         // Initialization to j
    , s {"CppCon"}  // Initialization to "CppCon" (duplication)
    , pi{}          // Initialization to nullptr (duplication)
  {}

  int i;                  // Note that the lifetime of the object
  std::string s;          // begins with the closing brace of the
  int* pi;                // delegated constructor!
};
```

Data Member Initialization

Core Guideline C.51: Use delegating constructors to represent common actions for all constructors of a class

Data Member Initialization

… or we could use in-class member initializers.

```
struct Widget
{
  Widget()
  {}

  Widget( int j )
    : i {j} // Initializing to j
  {}

  // Data members with in-class initializers
  int i{42};                   // initializing to 42
  std::string s{"CppCon"};     // initializing to "CppCon"
  int* pi{};                   // initialising to nullptr
};
```

In-class member initializers are used if the data member is not explicitly listed in the member initializer list.

Data Member Initialization

… or we could use in-class member initializers.

```
struct Widget
{
  Widget() = default;


  Widget( int j )
    : i {j} // Initializing to j
  {}

  // Data members with in-class initializers
  int i{42};                 // initializing to 42
  std::string s{"CppCon"};   // initializing to "CppCon"
  int* pi{};                 // initialising to nullptr
};
```

In-class member initializers are used if the data member is not explicitly listed in the member initializer list.

Data Member Initialization

Core Guideline C.44: Prefer default constructors to be simple and non-throwing

Core Guideline C.48: Prefer in-class initializers to member initializers in constructors for constant initializers

Guideline: Prefer to initialize pointer members to nullptr with in-class member initializers.

Implicit Conversions

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Implicit Conversions

```
class Widget
{
 public:
   Widget( int ) { std::puts( "Widget(int)" ); }
   // ...
};

void f( Widget );

int main()
{
   f( 42 );    // Calls the Widget ctor, then f
               //   (probably unintentionally)

   return EXIT_SUCCESS;
}
```

Implicit Conversions

```
class Widget
{
 public:
   explicit Widget( int ) { std::puts( "Widget(int)" ); }
   // ...
};

void f( Widget );

int main()
{
   f( 42 );    // Compilation error! No matching
               // function for ‘f(int)’ (as it should be)

   return EXIT_SUCCESS;
}
```

Implicit Conversions

Core Guideline C.46: By default, declare single-argument constructors `explicit`.

Order of Data Members

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Order of Member Data

Task, Step 1: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   bool b1;
   float f;
   bool b2;
};

std::cout << sizeof(Widget) << '\n';    // prints 12
```

Order of Member Data

Task, Step 1: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   bool b1;   char padding1[3];
   float f;   // Needs to be 4-byte aligned on x64
   bool b2;   char padding2[3];
};

std::cout << sizeof(Widget) << '\n';    // prints 12
```

Order of Member Data

Task, Step 2: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   bool b1;
   double d;
   bool b2;
};

std::cout << sizeof(Widget) << '\n';    // prints 24
```

Order of Member Data

Task, Step 2: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   bool b1;   char padding1[7];
   double d; // Needs to be 8-byte aligned on x64
   bool b2;   char padding2[7];
};


std::cout << sizeof(Widget) << '\n';    // prints 24
```

Order of Member Data

Task, Step 3: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   double d;    // Largest first
   bool b1;
   bool b2;
};


std::cout << sizeof(Widget) << '\n';   // prints 16
```

Order of Member Data

Task, Step 3: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   double d;   // Largest first
   bool b1;
   bool b2; char padding[6];
};

std::cout << sizeof(Widget) << '\n';   // prints 16
```

Order of Member Data

Task, Step 4: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   std::string s;  // Assumption: consumes 24 bytes
   bool b1;
   bool b2;
};


std::cout << sizeof(Widget) << '\n';   // prints 32
```

Order of Member Data

Task, Step 4: Assuming the x64 architecture, what is the size of the given struct `Widget`?

```
struct Widget {
   std::string s;  // Assumption: consumes 24 bytes
   bool b1;
   bool b2; char padding[6];
};


std::cout << sizeof(Widget) << '\n';   // prints 32
```

Order of Member Data

Guideline: Consider the alignment of data members when adding member data to a struct or class.

Core Guideline C.47: Define and initialise member variables in the order of member declaration

Const Correctness

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type* begin() noexcept;
   Type* end()   noexcept;
   // ...
};

std::ostream& operator<<( std::ostream& os
                        , FixedVector<int,10> v )
{
   for( int i : v ) { /*...*/ }

   return EXIT_SUCCESS;
}
```

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type* begin() noexcept;
   Type* end()   noexcept;
   // ...
};

std::ostream& operator<<( std::ostream& os
                        , FixedVector<int,10> const& v )
{
   for( int i : v ) { /*...*/ }   // Compilation error!

   return EXIT_SUCCESS;
}
```

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type* begin()  const noexcept;
   Type* end()    const noexcept;
   // ...
};
```

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   // ...
};
```

Huh? A const pointer?

Detour: West Coast vs. East Coast

”const modifies what is on its left. Unless there is nothing on its left, in which case it modifies what’s on its right.”

(Jon Kalb, A Foolish Consistency)

`const Type*` → Commonly known as **West-Coast const**

`Type const*` → Commonly known as **East-Coast const**

Detour: West Coast vs. East Coast

"const modifies what is on its left. Unless there is nothing on its left, in which case it modifies what's on its right."

(Jon Kalb, A Foolish Consistency)

`const Type*` → Commonly known as const West-Coast

`Type const*` → Commonly known as East-Coast const

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   // ...
};

int main()
{
   FixedVector<int,10> v{ /*...*/ };

   std::fill( v.begin(), v.end(), 42 );   // Compilation error!

   return EXIT_SUCCESS;
}
```

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   Type*       begin()        noexcept;
   Type*       end()          noexcept;

   // ...
};
```

Const Correctness

Task: What is wrong with the declaration of the `begin()` and `end()` functions?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   Type*       begin()        noexcept;
   Type*       end()          noexcept;
   Type const* cbegin() const noexcept;
   Type const* cend()   const noexcept;
   // ...
};
```

Const Correctness

```
namespace std {

template< typename T
        , typename Deleter = std::default_delete<T> >
class unique_ptr
{
 public:
   using pointer = T*;  // Simplified!

   pointer get() const noexcept;  // const member function returning
   // ...                          // a pointer to non-const T!
};

} // namespace std

int main()
{
   std::unique_ptr<int> const ptr1;  // Semantically equivalent
   int* const ptr2;

   return EXIT_SUCCESS;
}
```

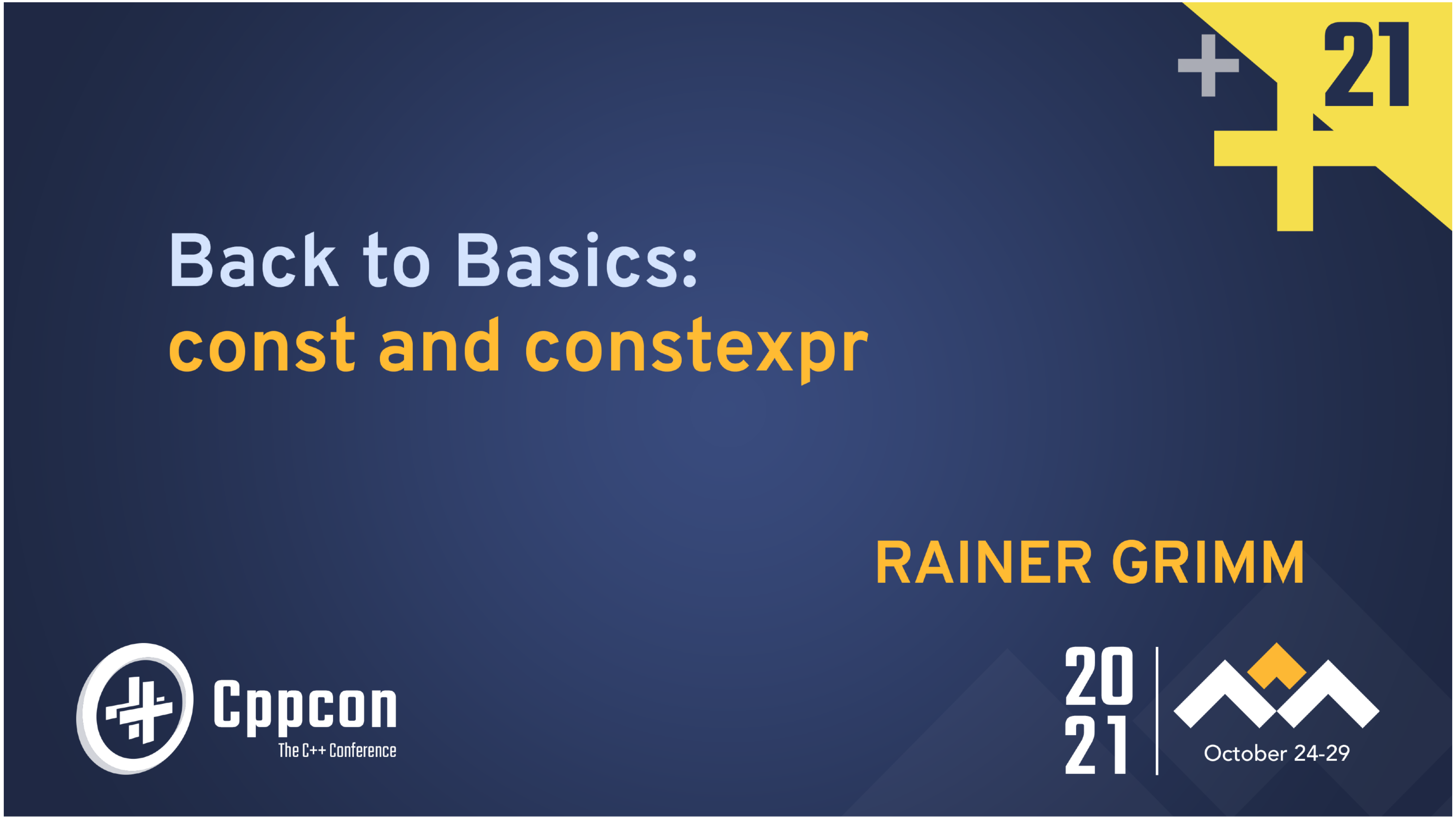
Const Correctness

```
namespace std {

template< typename T
        , typename Deleter = std::default_delete<T> >
class unique_ptr
{
 public:
   using pointer = T*;   // Simplified!

   pointer get() const noexcept;  // const member function returning
   // ...                         // a pointer to non-const T!
};

} // namespace std

int main()
{
   std::unique_ptr<int const> const ptr1;  // Semantically equivalent
   int const* const ptr2;

   return EXIT_SUCCESS;
}
```

Const Correctness

Core Guideline Con.2: By default, make member functions `const`

Guideline: Const correctness is part of the semantics of your class.



Tuesday, October 26th, 10:30am MDT

Encapsulating Design Decisions

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Encapsulating Design Decisions

Task: You decide that you want to represent iterators by means of class types. Why is that a problem?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   Type*       begin()        noexcept;
   Type*       end()          noexcept;
   Type const* cbegin() const noexcept;
   Type const* cend()   const noexcept;
   // ...
};
```

Encapsulating Design Decisions

Task: You decide that you want to represent iterators by means of class types. Why is that a problem?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   Type*       begin()        noexcept;
   Type*       end()          noexcept;
   Type const* cbegin() const noexcept;
   Type const* cend()   const noexcept;
   // ...
};
```

Encapsulating Design Decisions

Task: You decide that you want to represent iterators by means of class types. Why is that a problem?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   using iterator = Type*;
   using const_iterator = const Type*;

   Type const* begin()  const noexcept;
   Type const* end()    const noexcept;
   Type*       begin()        noexcept;
   Type*       end()          noexcept;
   Type const* cbegin() const noexcept;
   Type const* cend()   const noexcept;
   // ...
};
```

Encapsulating Design Decisions

Task: You decide that you want to represent iterators by means of class types. Why is that a problem?

```
template< typename Type, size_t Capacity >
class FixedVector final
{
 public:
   // ...
   using iterator = Type*;
   using const_iterator = const Type*;

   const_iterator begin()  const noexcept;
   const_iterator end()    const noexcept;
   iterator       begin()        noexcept;
   iterator       end()          noexcept;
   const_iterator cbegin() const noexcept;
   const_iterator cend()   const noexcept;
   // ...
};
```

Encapsulating Design Decisions

```
namespace std {

template< typename T
        , typename Allocator = std::allocator<T> >
class vector
{
 public:
   constexpr T*       data()       noexcept;   // data() is expected to
   constexpr T const* data() const noexcept;   // return a pointer to the
   // ...                                      // first element
};

} // namespace std
```

Encapsulating Design Decisions

Guideline: Encapsulate design decisions (i.e. variation points).

Guideline: Design classes for easy change.

Qualified/Modified Member Data

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Qualified/Modified Member Data

Task: What is the problem of the given struct `Widget`?

```
struct Widget
{
   int const i;
   double& d;

   // Widget& operator=( Widget const& );  // implicitly deleted
   // Widget& operator=( Widget&& );       // not declared
};
```

Assignment to const data members or references doesn't work, so the compiler cannot generate the two assignment operators!

Qualified/Modified Member Data

Reference members can be stored as pointers …

```
struct Widget
{
 public:
   Widget( double& d ) : pd_( &d ) {}

   double&       get()       noexcept { return *pd_; }
   double const& get() const noexcept { return *pd_; }

 private:
   double* pd_;
};
```

Qualified/Modified Member Data

… or as `std::reference_wrapper`.

```
#include <functional>

struct Widget
{
 public:
   Widget( double& d ) : d_( d ) {}

   double&       get()       noexcept { return d_; }
   double const& get() const noexcept { return d_; }

 private:
   std::reference_wrapper<double> d_;
};
```

Qualified/Modified Member Data

Core Guideline C.12: Don't make data members `const` or references

Guideline: Remember that a class with const or reference data member cannot be copy/move assigned by default.

Guideline: Strive for symmetry between the two copy operations.

Guideline: Strive for symmetry between the two move operations.

Visibility vs. Accessibility

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Visibility vs. Accessibility

Task: Which of the following two functions is called in the subsequent function call?

```
class Widget
{
 public:
   void doSomething( int );      // (1)
 private:
   void doSomething( double );   // (2)
};

Widget w{};
w.doSomething( 1.0 );
```

The compiler tries to call function (2), but quits the compilation process with an error about an **access violation**: function (2) is declared private!

Visibility vs. Accessibility

Task: Which of the following two functions is called in the subsequent function call?

```
glass Widget
{
 public:
   void doSomething( int );      // (1)
 private:
   void doSomething( double );   // (2)
};

Widget w{};
w.doSomething( 1.0 );
```

Visibility vs. Accessibility

Task: Which of the following two functions is called in the subsequent function call?

```
class Widget
{
 public:
   void doSomething( int );      // (1)
 private:
   void doSomething( double );   // (2)
};

Widget w{};
w.doSomething( 1U );
```

This results in an **ambiguous function call**. The compiler still sees both functions and cannot decide which conversion to perform!

Visibility vs. Accessibility

Remember the four steps of the compiler to resolve a function call:

1. **Name lookup**: Select all (visible) candidate functions with a certain name within the current scope. If none is found, proceed into the next surrounding scope.

2. **Overload resolution**: Find the best match among the selected candidate functions. If necessary, apply the necessary argument conversions.

3. **Access labels**: Check if the best match is accessible from the given call site.

4. **=delete**: Check if the best match has been explicitly deleted.

Content

Back to Basics: Class Design (Part 1)

- The Challenge of Class Design
- Design Guidelines
  - Design for Readability
  - Design for Change and Extension
  - Design for Testability
- Implementation Guidelines
  - Resource Management

Back to Basics: Class Design (Part 2)

- Implementation Guidelines
  - Data Member Initialization
  - Implicit Conversions
  - Order of Data Members
  - Const Correctness
  - Encapsulating Design Decisions
  - Qualified/Modified Member Data
  - Visibility vs. Accessibility

Summary

Guideline: Separate concerns!

Guideline: Design classes for easy change.

Guideline: Design classes for easy extensions.

Guideline: Design classes to be testable.

Back to Basics:
Designing Classes (part 2 of 2)

KLAUS IGLBERGER

Cppcon
The C++ Conference

2021

October 24-29