Writing Modern C++ Code

CODE MONSTERS

Marc Grégoire
Software Architect
marc.gregoire@nuonsoft.com
http://www.nuonsoft.com/blog/

Microsoft Extended Experts Team

Microsoft Most Valuable Professional

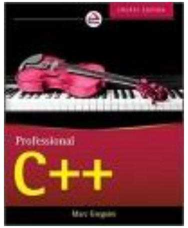
December 10th 2019

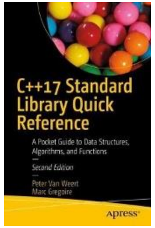
Marc Grégoire

- Belgium
- Software architect for Nikon Metrology

- Microsoft VC++ MVP
- Microsoft Extended Experts Team member

- Author of [Professional C++, 2^{nd}, 3^{rd} and 4^{th} Edition]
- Co-author of [C++ Standard Library Quick Reference] & [C++17 Standard Library Quick Reference]





- Founder of the [Belgian C++ Users Group] (BeCPP)

Modern C++

Modern C++ is

C++11 and later

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Uniform Initialization

- Use brace initialization for any type
- Ex: Old way:

```
std::vector<int> vec;
vec.push_back(1);
vec.push_back(2);
vec.push_back(3);
```

- New way:

```
std::vector<int> vec = { 1,2,3 };
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Auto Type Deduction

CODE MONSTERS

- Compiler can automatically deduce types of variables
- Ex:

```cpp
auto myInt = 123;

// Old:
std::vector<int>::const_iterator i = vec.begin();
// Modern:
auto i = vec.begin();
```

Auto Type Deduction

- Compiler can automatically deduce return types
- Ex:

```cpp
auto GetHello() {
  return "Hello";
}

int main() {
  auto result = GetHello();
}
```

Auto Type Deduction

- Benefits
  - Reduces verbosity, allowing important code to stand out
  - Avoids type mismatches
  - Increases genericity, by allowing templates to be written that care less about the types of intermediate expressions
  - Deals with undocumented or unspeakable types, like lambdas

Modern C++

CODE MONSTERS

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Range-Based for Loops

- Loops over all elements of a container
- Easier to write and read, expresses intend more clearly
- Instead of:

```
std::vector<int> vec = { 1,2,3,4,5,6 };
for (std::vector<int>::iterator iter = vec.begin();
     iter != vec.end(); ++iter) {
  *iter *= 2;
}
```

- In Modern C++:

```
std::vector<int> vec = { 1,2,3,4,5,6 };
for (auto& i : vec) {
  i *= 2;
}
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Real Null Pointer Type (nullptr)

- Problem with old `NULL` :
  - It is implicitly converted to an integer because:

```
#define NULL 0
```

  - It might not do what you expect it to do:

```
void foo(char* p) { cout << "char* version" << endl; }
void foo(int i) { cout << "int version" << endl; }
int main() {
  foo(0);          // Calls int version
  foo(NULL);       // Also calls int version ☹
  foo(nullptr); // Properly calls char* version ☺
}
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

In-Class Member Initialization

- Only certain members could be initialized in-class pre c++11
  - Others had to be initialized in the constructor
- C++11 supports in-class member initialization, removing the need for a constructor

```
class MyObject
{
private:
  int m_someInt = 42;
  std::string m_aString = "Hello World!";
  std::vector<std::string> m_aVector = { "11", "22", "33" };
};
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Nested Namespaces

- Instead of:

```
namespace Nuonsoft {
  namespace Platform {
    namespace Interfaces {
      namespace UI {
        class ICommandWindow { };
}}}}
```

- In Modern C++:

```
namespace Nuonsoft::Platform::Interfaces::UI
{
  class ICommandWindow { };
}
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Structured Bindings

- Deconstructs **pair**, **tuple**, …
- Instead of:

```
std::set<int> mySet;
auto result = mySet.insert(42);
if (result.second) { /* insert succeeded. */ }
```

- In Modern C++:

```
auto[location, success] = mySet.insert(42);
if (success) { /* insert succeeded. */ }
```

- Benefit: no need for multiple output parameters, just return a tuple and deconstruct it on the calling side

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

CTAD

- **C**lass **T**emplate **A**rgument **D**eduction
- Previously, only deduction of template types for function templates, now also for class templates

```
std::vector<int> v2 = { 1,2,3 };
// In Modern C++:
std::vector v1 = { 1,2,3 };

std::pair<std::string, double> p2{ "Hello"s, 42.24 };
// or:
auto p3 = std::make_pair("Hello"s, 42.24);
// In Modern C++:
std::pair p1{ "Hello"s, 42.24 };
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

String Views

<string_view>

- Usually, strings are passed by **const std::string&**
- But can cause unnecessary copying of strings
  - For example: passing a string literal to a **const std::string&** creates a temporary **std::string**
- A **string_view** is read-only view on any string
- Cheap to copy, only contains a pointer and a length
- Always use **std::string_view** instead of **const std::string&**
- Pass by value is ok

String Views

<string_view>

CODE MONSTERS

- **std::string_view** is almost a drop-in replacement for **const std::string&**
  - No **c_str()**, only **data()**, because a **string_view** is not necessarily null terminated
  - No implicit conversion from a **string_view** to a **string** to avoid accidental copying
  - Implicit conversion from **string** to **string_view**
  - Extra members: **remove_prefix(n)** and **remove_suffix(n)**

String Views

<string_view>

- Example:

```
void ProcessString(std::string_view myString)
{
}

ProcessString("Hello World");

std::string str = "Hello!";
ProcessString(str);
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

std::optional

<optional>

CODE MONSTERS

- std::optional for optional parameters, return values, or data members

```
void ProcessData(std::string_view data1,
                 std::optional<std::string_view> moreData)
{
}

std::optional<int> Generate()
{
  if (ok)
    return 42;
  return {}; // or return std::nullopt;
}
```

std::optional

<optional>

CODE MONSTERS

- Access data from **std::optional**:

```
void ProcessData(std::string_view data1,
                 std::optional<std::string_view> moreData)
{
  if (moreData)
  {
    // Access data through moreData.value()
    // Or *moreData (undefined behavior if optional is empty)
  }
  // Or, use moreData.value_or("Default Data"s)
}
```

Throws std::bad_optional_access if the optional is empty.

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Lambda Expressions

- Lambda expressions can be seen as small inline anonymous functions
- Syntax

```
[capture_block](parameters) mutable exception_specification ->
  return_type { body }
```

  - capture block: how to capture variables from enclosing scope
  - parameters (optional): parameter list, just like a function
  - mutable (optional): marks the function call operator non-const
  - exception_specification (optional): = throw list
  - return_type (optional): the return type; if omitted, compiler deduces return type

Lambda Expressions

- Basic example:

```
int main() {
  [] { cout << "Hello from Lambda" << endl; }();
}
```

Lambda Expressions

- Powerful in combination with Standard Library algorithms
- Ex:

```
bool gt5(const int& i) { return i > 5; }
...
std::vector<int> vec{ 1,2,3,4,5,6,7,8,9 };
int c = std::count_if(vec.begin(), vec.end(), gt5);

// With lambda expressions, no need for separate function:
int c = std::count_if(vec.begin(), vec.end(),
                      [](int i) { return i > 5; });
```

Lambda Expressions

CODE MONSTERS

- `auto` can be used to name lambdas, allowing them to be reused

```
int main() {
  auto doubler = [](const int i) {
    return i * 2;
  };
  std::vector<int> v1, v2;
  // ... Fill vectors
  // Transform elements from vectors:
  std::transform(v1.begin(), v1.end(), v1.begin(), doubler);
  std::transform(v2.begin(), v2.end(), v2.begin(), doubler);
}
```

Lambda Expressions

- Can capture variables from enclosing scope
- Ex:

```
int value = 3;
int c = std::count_if(vec.begin(), vec.end(),
                      [=](int i) { return i > value; });
```

- Capture block
  - `[ ]` captures nothing
  - `[=]` captures all variables by value
  - `[&]` captures all variables by reference

  - `[&x]` captures only `x` by reference and nothing else
  - `[x]` captures only `x` by value and nothing else
  - `[=, &x, &y]` captures by value by default, except variables `x` and `y`, which are captured by reference
  - `[&, x]` captures by reference by default, except variable `x`, which is captured by value

Lambda Expressions

- Compiler can automatically deduce parameter types for lambda expressions
- Ex:

```
auto doubler = [](int i) {
  return i * 2;
};

// With auto parameter
auto doubler = [](auto i) {
  return i * 2;
};
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Parallel Algorithms

<execution> <algorithm>

CODE MONSTERS

- Around 70 algorithms can be executed in parallel
- Just specify an execution policy, <execution>
  - Policies: std::execution::seq, par, par_unseq
- E.g.:

```
std::vector in{ 139, 41, 151, 137, 73 };
std::sort(                              begin(in), end(in));
```

- You have to take care of data races!

Parallel Algorithms

<execution> <algorithm>

CODE MONSTERS

```
std::vector<int> in{ /* ... */ };
std::vector<int> out;

std::for_each(std::execution::par, cbegin(in), cend(in), [&out](int i) {
  int j = DoSomething(i);
  out.push_back(j);
});
```

Not thread-safe!

```
std::mutex m;
std::for_each(std::execution::par, cbegin(in), cend(in), [&out, &m](int i) {
  int j = DoSomething(i);
  std::scoped_lock lock(m);
  out.push_back(j);
});
```

Parallel Algorithms

<execution> <algorithm>

- When an exception is thrown in a parallel algorithm
  - → std::terminate() is called!

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Things To Unlearn

- If you know C++ pre-C++11 → **unlearn** a few things related to memory management
- Avoid low-level pointer and raw memory manipulation → use higher level constructs
  - Smart pointers, containers, RAII, ...
- Do not use `new` / `new []` / `delete` / `delete[]`, use **smart pointers**:
  - Exceptions safe
  - Leak free
  - Less error prone
  - Deterministic, *unlike garbage collectors (!)*

Things To Unlearn

- Never do something as follows (C-style coding):

```
FILE* f = fopen("data.ext", "w");
// ...
fclose(f);
```

  - Not exception safe!
  - Error prone!

  - Instead, use concepts like RAII, discussed later

Modern C++

CODE MONSTERS

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Arrays

- **Don't** use plain-old-dumb C-style arrays using
  - ~~new[]~~
  - ~~delete[]~~
- Instead, use containers like
  - std::array
  - std::vector
  - …

Raw Pointers

- Raw pointers in C++
  - Do not use them if there is ownership involved
    - Use smart pointers
  - It's ok to use them for pure observers
    - Only if you can guarantee that the lifetime of the object pointed to is longer than the lifetime of the observer

Smart Pointers

- Use `shared_ptr` or `unique_ptr` from `<memory>`:
  - `shared_ptr`: reference counted
  - `unique_ptr`: not reference counted, non-copyable, but movable
  - Safe to be stored in containers

Smart Pointers

- Use `std::make_unique()` / `make_shared()`
  - Less typing in combination with auto type deduction
    - Without :

```
unique_ptr<int> up(new int(42));  // int written twice ☹
```

    - With:

```
auto up = make_unique<int>(42);   // int written once ☺
```

  - A bit more performant in certain cases

Smart Pointers

- Never use `std::auto_ptr`
  - Deprecated since C++11
  - **Removed** (!) from C++17

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Old C++ Versus New C++

- Suppose you have the following Circle class

```
class Circle
{
public:
  Circle(double radius) : m_radius(radius) {}
  double GetRadius() const { return m_radius; }
  bool operator==(const Circle& c) { return m_radius == c.m_radius; }

private:
  double m_radius;
};

ostream& operator<<(ostream& os, const Circle& circle)
{
  os << "circle with radius " << circle.GetRadius();
  return os;
}
```

Old C++ Versus New C++

```
vector<Circle*> LoadCirclesRaw()
{
  vector<Circle*> circles;
  circles.push_back(new Circle(11));
  circles.push_back(new Circle(42));
  circles.push_back(new Circle(33));
  return circles;
}
```

Old

```
vector<unique_ptr<Circle>> LoadCircles()
{
  vector<unique_ptr<Circle>> circles;
  circles.push_back(make_unique<Circle>(11));
  circles.push_back(make_unique<Circle>(42));
  circles.push_back(make_unique<Circle>(33));
  return circles;
}
```

New

Old C++ Versus New C++

CODE

auto type deduction

T* → unique_ptr<T>
new → make_unique
vector<T*> → vector<unique_ptr<T>>

Old

```
Circle* p = new Circle(42);
vector<Circle*> vw = LoadCirclesRaw();
for (vector<Circle*>::iterator i =
     vw.begin(); i != vw.end(); ++i)
{
  if (*i && **i == *p)
    cout << **i << " is a match\n";
}
// ...
for (vector<Circle*>::iterator i =
     vw.begin(); i != vw.end(); ++i)
{
  delete *i;
}
vw.clear();
delete p;
```

New

```
auto p = make_unique<Circle>(42);
for (auto& circle : LoadCircles())
{
  if (circle && *circle == *p)
    cout << *circle << " is a match\n";
}
```

no need for “circles” variable
no need for “delete”
automatic lifetime management
exception-safe

range-based
for loops

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Avoid Delete

- Write your code in such a way that there is never a need to use `delete` or `delete[]`

Avoid Delete

- Don’t write code as follows:

```
void foo()
{
  MyObject* p = new MyObject();

  // ...

  delete p;
}
```

  - Not exception safe!

Avoid Delete

- Instead, use shared_ptr or unique_ptr:

```
void foo()
{
  auto p1 = make_unique<MyObject>();
  // ...
}
```

- Or, even better, use the stack:

```
void foo()
{
  MyObject obj;
  // ...
}
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Automatic Lifetime

- **Automatic Lifetime** = Efficient + Exception Safe

```
class widget {
private:
  gadget g;
public:
  void draw();
};
```

Lifetime automatically tied to enclosing object
No leak, exception safe

```
void f() {
  widget w;
  // ...
  w.draw();
  // ...
}
```

Lifetime automatically tied to enclosing scope
Constructs w, including the w.g gadget member

Automatic destruction and deallocation of w and w.g

Automatic exception safety, as if “finally { w.g.dispose(); w.dispose(); }”

The Heap and Smart Pointers

```
class gadget;
class widget {
private:
  shared_ptr<gadget> g;
};
class gadget {
private:
  weak_ptr<widget> w;
};
```

shared ownership
keeps gadget alive with
auto lifetime management
no leak, exception safe

use weak_ptr to break
reference-count cycles

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

RAII

- RAII = **R**esource **A**cquisition **I**s **I**nitialization
  - A wrapper class:
    - Constructor → acquires a resource
    - Destructor → automatically releases the resource

Deterministic ☺

  - Often you can use `std::unique_ptr` or `shared_ptr`, as simple RAII objects

RAII

- For example, instead of:

```
FILE* f = fopen("data.ext", "w");
// ...
fclose(f);
```

- Use shared_ptr:

```
shared_ptr<FILE> filePtr(fopen("data.ext", "w"), fclose);
```

- Or unique_ptr:

```
unique_ptr<FILE, decltype(&fclose)> p(fopen("data.ext", "w"), fclose);
```

RAII

- Or write your own RAII object

```
class File
{
public:
  // Constructor acquires resource
  File(FILE* file) : m_file(file) {}

  // Destructor automatically releases resource
  ~File()
  {
    if (m_file)
    {
      fclose(m_file);
      m_file = nullptr;
    }
  }

  // Conversion operator to FILE*
  operator FILE* () const { return m_file; }

private:
  FILE* m_file;
};
```

Creating a File instance:

```
File myFile(fopen("data.ext", "w"));
```

Using a File instance:

```
fputc('a', myFile);
```

Thanks to the FILE* conversion operator, you can use a File instance just as you would use a FILE*.

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Garbage Collection in C++?

"C++ is the best language for garbage collection principally because it creates ***less garbage****."*

— Bjarne Stroustrup

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Modules

- Advantages
  - No header files
  - Separation into interface files and implementation files is possible but not needed
  - Modules explicitly state what should be exported (e.g. classes, functions, …)
  - No need for include guards
  - No need to invent unique names, same name in multiple modules will not clash
  - Modules are processed only once → faster build times
  - Preprocessor macros have no effect on modules
  - Order of module imports is not important

Modules

- Create a module:

```
// codemonsters.cpp
export module codemonsters;

namespace CodeMonsters {
  auto GetWelcomeHelper() { return "Welcome to CodeMonsters 2019!"; }
  export auto GetWelcome() { return GetWelcomeHelper(); }
}
```

- Consume a module:

```
// main.cpp
import codemonsters;

int main() {
  std::cout << CodeMonsters::GetWelcome();
}
```

Modules

- C++20 doesn't specify if and how to modularize the Standard Library
- Visual Studio makes it available as follows:
  - **`std.regex`** → <regex>
  - **`std.filesystem`** → <filesystem>
  - **`std.memory`** → <memory>
  - **`std.threading`** → <atomic>, <condition_variable>, <future>, <mutex>, <shared_mutex>, and <thread>
  - **`std.core`** → everything else in the C++ Standard Library

Modules

- You can “import” header files, e.g.:
  - `import <iostream>`
  - Implicitly turns the **`iostream`** header into a module
  - Improves build throughput, as **`iostream`** will then be processed only once
  - Comparable to precompiled header files (PCH)

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Ranges

- What's a range?
  - An object referring to a sequence/range of elements
  - Similar to a begin/end iterator pair, but not replace them
- Why ranges?
  - Provide nicer and easier to read syntax:

```
vector data{ 11, 22, 33 };
sort(begin(data), end(data));
sort(data); // with ranges
```

  - Eliminate mismatching begin/end iterators
  - Allows "range adaptors" to lazily transform/filter underlying sequences of elements

Ranges

- Based on two core components:
  - Views: range adaptors: lazily evaluated, non-owning, non-mutating
  - Algorithms: all Standard Library algorithms accepting ranges instead of iterator pairs
  - Views can be chained using pipes → |

Ranges

- Example of chaining views:

```
vector data{ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
auto result = data | views::remove_if([](int i) { return i % 2 == 1; })
                   | views::transform([](int i) { return to_string(i); });
// result == {"2","4","6","8","10"};
```

  - Note: all lazily executed: nothing is done until you iterate over `result`

Ranges

- Example of a filtering and transforming chain of range adaptors:

```
int total = accumulate(
    view::ints(1) |
    view::transform([](int i) {return i * i; }) |
    view::take(10),
0);
```

  - `view::ints(1)` lazily generates an infinite sequence of integers
  - this is lazily squared
  - And finally we only take the first 10 elements of the infinite sequence and accumulate these

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Concepts

- Requirements attached to class- and function templates to constraint template arguments
- Predicates evaluated at compile time
- Can contain multiple statements

Concepts

CODE MONSTERS

- Example of a concept definition:

```
template<typename T>
concept Incrementable = requires(T x) { x++; ++x; };
```

- Using this concept:

```
template<Incrementable T>
void Foo(T t);

template<typename T> requires Incrementable<T>
void Foo(T t);

template<typename T>
void Foo(T t) requires Incrementable<T>;

void Foo(Incrementable auto t);
```

Concepts

- A concept to require a **`size()`** method returning a **`size_t`**

```
template <typename T>
concept HasSize = requires (T x) {
  { x.size() } -> std::convertible_to<std::size_t>;
};
```

Concepts

- Combining concepts:

```
template<typename T> requires Incrementable<T> && Decrementable<T>
void Foo(T t);
```

- Or:

```
template<typename T>
concept Incr_Decrementable = Incrementable<T> && Decrementable<T>;

void Foo(Incr_Decrementable auto t);
```

- The Standard defines a whole collection of standard concepts:
  - `same`, `derived_from`, `convertible_to`, `integral`, `constructible`, …
  - `sortable`, `mergeable`, `permutable`, …

Concepts

- Concepts help with compiler error message
- Easier to read template error messages: e.g.:

```
Error: cannot call Foo() with Bar.
Note:  concept Incrementable<Bar> was not satisfied.
```

Modern C++

- Uniform Initialization
- Auto Type Deduction
- Range-Based for Loops
- Real Null Pointer Type
- In-Class Member Initialization
- Nested Namespaces
- Structured Bindings
- CTAD
- String Views
- std::optional
- Lambda Expressions
- Parallel Algorithms
- Memory Management
  - Things To Unlearn
  - Pointers
  - Old C++ Versus New C++
  - Avoid delete
  - Automatic Lifetime (stack & heap)
  - RAII
  - Garbage Collection in C++?
- C++20
  - Modules
  - Ranges
  - Concepts

Questions

?