Modern C++: When Efficiency Matters

Training Material

CppCon Academy, 2024-09-21

Andreas Fertig

Write unique code!

All programs, procedures and electronic circuits contained in this book have been created to the best of our knowledge and belief and have been tested with care. Nevertheless, errors cannot be completely ruled out. For this reason, the program material contained in this book is not associated with any obligation or guarantee of any kind. The author therefore assumes no responsibility and will not accept any liability, consequential or otherwise, arising in any way from the use of this program material or parts thereof.

Version: v1.1

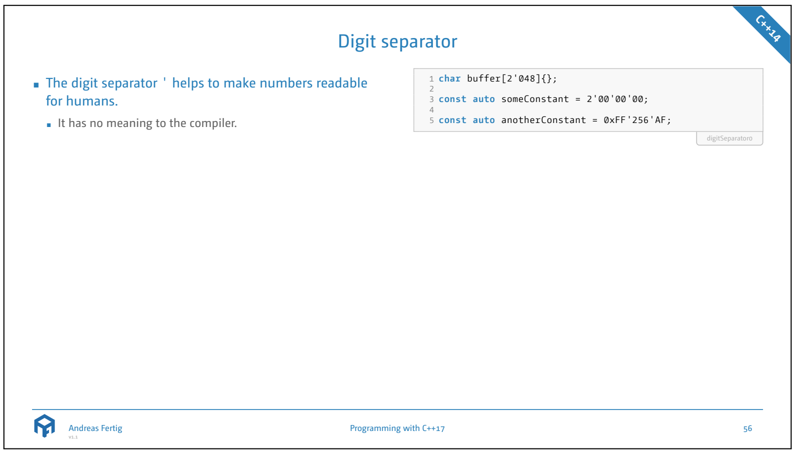
Planning, typesetting and cover design: Andreas Fertig
Cover art and illustrations: Franziska Panter https://franziskapanter.com
Production and publishing: Andreas Fertig

Style and conventions

The following shows the execution of a program. I used the Linux way here and skipped supplying the desired output name, resulting in a.out as the program name.

```
$ ./a.out
Hello, C++!
```

- <string> stands for a header file with the name string
- [[xyz]] marks a C++ attribute with the name xyz.



All code listings have their filename attached at the bottom right (in the slide above digitSeparator0). The corresponding file is part of materials.zip which is explained in the next section.

At the top right corner of a slide you can see a blue banner which contains the C++ standard the feature on the slide was introduced. Features from C++11 and before have no banner.

materials.zip

The materials.zip you received contains, among the handout.pdf the files and structure as shown below.

The CMakeLists.txt in code-samples contains the build information for the exercises and solutions folder. You can use them to build, run, and debug the exercises.

More optional is the CMakeLists.txt in slide-code. This folder contains the code from all the slides you will see in this class. With the cmake-file building and running them should be easy.

- materials.zip
  - handout.pdf
  - code-samples
    - CMakeLists.txt
    - initial-check.cpp
    - exercises
      - ex*.cpp
    - solutions
      - sol*.cpp
    - slide-code
      - CMakeLists.txt
      - *.cpp

Schedule

The timezone is America/Denver (UTC-0600).

Block 1: **09:00 - 10:30**

15 min break

Block 2: **10:45 - 12:15**

60 min break

Block 3: **13:15 - 14:45**

15 min break

Block 4: **15:00 - 16:30**

Sample code disclaimer

The source code examples in this material can be used without any warranty.

Please keep in mind that some of this code may be untested.

https://AndreasFertig.com
post@AndreasFertig.com

Overview

1 Smaller Language Features 8
2 constexpr 31
3 Lambdas 39
4 Speedy Templates 52
5 Move semantics 74
6 The costs of abstractions 91
7 Using the Standard Template Library efficiently 113
8 Miscellaneous 123

My motto

Write unique code.

https://AndreasFertig.com
post@AndreasFertig.com

What is efficient?

- What qualifies as efficient is … well it depends.
  - Do we write software that runs on a battery-powered device?
  - Do we have to deal with large amounts of data?
  - Is timing a factor?
  - Is the device equipped with minimal hardware (e.g. slow CPU, only a few kB of RAM/ROM)?
  - Is the compile speed of our code a factor?
  - Are there restrictions in place, like no heap allocations?
  - Do we refactor our code frequently?

If there are any questions

- Your questions, comments, topics, etc., have priority.
- This course aims to address the topics that interest you.
  - Regardless of whether the topics are on the slides or not.
  - If you see or hear something unfamiliar, please ask!
- Ask the standard. The official standard costs. Alternative:
  - Drafts are free of charge.
  - https://wg21.link/std
  - PDF: [1] or GitHub: [2]
- Information close to the standard:
  - https://cppreference.com
- Try & verify:
  - Online compiler
  - https://compiler-explorer.com
  - https://wandbox.org
  - https://cppinsights.io
  - …

1. Smaller Language Features

Uniform initialization

- Initialization forms:

```
std::string u;                          (A) default
std::string v = std::string();          (B) value
std::string w("C++");                   (C) direct
std::string x = "C++";                  (D) copy
char  y[4]     = {'C', '+', '+'};       (E) aggregate
char& z        = y[0];                  (F) reference
```

bracedInit0

- Unified syntax:

```
std::string u{};                        (A) default
std::string v{};                        (B) value
std::string w{"C++"};                   (C) direct
std::string x{"C++"};                   (D) copy
char        y[4]{'C', '+', '+'};        (E) aggregate
char&       z{y[0]};                    (F) reference
```

bracedInit1

- Can be used everywhere to initialize variables.
- With {}, we can achieve a *default* or *zero initialization*.
- Braced (uniform) initialization guarantees that there will be no *narrowing* (more about that later).

https://AndreasFertig.com
post@AndreasFertig.com

Uniform initialization

- We can express the difference between a copy-constructor and a copy-assignment:

```cpp
Apple a;
Apple b = a;
b       = a;
```

uniformInit0

- By omitting the =, it becomes more evident that it is an initialization, not an assignment.

```cpp
Apple a{};
Apple b{a};
b = a;
```

uniformInit1

- The *most vexing problem* [1] can be avoided:

```cpp
class Lifeguard {
public:
  Lifeguard() { puts("Lifeguard()"); }
  Lifeguard(int) { puts("Lifeguard(int)"); }
  Lifeguard(const Lifeguard&)
  {
    puts("Lifeguard(Lifeguard&)");
  }
};

void Use()
{
  Lifeguard a(3);
  Lifeguard b();
  Lifeguard c{3};
  Lifeguard d{};
}
```

uniformInit2

```
$ ./a.out
Lifeguard(int)
Lifeguard(int)
Lifeguard()
```

[1] If it can be interpreted as a function prototype, it will be.

Uniform initialization

```cpp
// Classic C++

const int arr[]{3, 4, 27, 22, 9};

std::vector<int> v;
for(int i = 0; i < 5; ++i) { v.push_back(arr[i]); }

std::set<int> s;
for(int i = 0; i < 5; ++i) { s.insert(arr[i]); }

std::map<int, std::string> m;
m[0] = "null";
m[1] = "first";
m[2] = "second";

std::vector<int> v2;
v2.push_back(20);
v2.push_back(30);
v2.push_back(40);
v2.push_back(50);
```

uniformInit3

```cpp
// Modern C++



const std::vector<int> v{3, 4, 27, 22, 9};


const std::set<int> s{3, 4, 27, 22, 9};


const std::map<int, std::string> m{{0, "null"},
                                   {1, "first"},
                                   {2, "second"}};


const std::vector<int> v2{20, 30, 40, 50};
```

uniformInit4

https://AndreasFertig.com
post@AndreasFertig.com

Uniform initialization - `std::initializer_list`

- Be careful if you use `std::initializer_list` as a parameter in a constructor of a custom class.
- The order that the compiler uses for braced initialization forms is as follows:
  a) `initializer_list`
  b) regular constructor
  c) aggregate initialization

```cpp
class Lifeguard {
public:
  explicit Lifeguard(int) { puts("Lifeguard(int)"); }

  Lifeguard(std::initializer_list<int>)
  {
    puts("Lifeguard(std::initializer_list<int>)");
  }
};

void Use()
{
  Lifeguard f{2};
  Lifeguard f2{2, 3};

  Lifeguard f3(2);
  Lifeguard f4({2, 3});
}
```

uniformInits

```
$ ./a.out
Lifeguard(std::initializer_list<int>)
Lifeguard(std::initializer_list<int>)
Lifeguard(int)
Lifeguard(std::initializer_list<int>)
```


Exercise

Compile the file `initial-check.cpp`. The output should look like this:

```
$ ./a.out
Supported:
 - C++11:  [OK]
 - C++14:  [OK]
 - C++17:  [OK]
 - C++20:  [OK]
 - C++23:  [FAILED]

Overall: READY
```


https://AndreasFertig.com
post@AndreasFertig.com

Exercise

a) Apply uniform initialization to `exUniformInit.cpp` .
- Solution: `solUniformInit.cpp`

b) Use `exUniformInit2.cpp` to implement a function `List` which takes a `std::initializer_list<int>`. Print the values in `List`.
- Solution: `solUniformInit2.cpp`

c) Use `exInitializerList2.cpp` to write a class with a default and a copy-constructor. For each constructor, print a message to identify the constructor if it is called. Execute the following initializations and look at the output:

```cpp
UInit a;
UInit b(a);
UInit c{a};
```

Does this match your expectations?
- Solution: `solInitializerList2.cpp`

d) Extend the class with a constructor that takes a `std::initializer_list<UInit>`. Add a message to this constructor as well. Now, execute the program again. Does the output match your expectations?
- Solution: `solInitializerList3.cpp`

range-based for loops

- Users need to know fewer (internal) details about a class.
  - begin, end
- This makes refactoring easier.

```cpp
const std::vector<int> numbers{2, 3, 5, 7};

for(auto it{numbers.begin()}; it != numbers.end();
    ++it) {
  std::cout << *it << '\n';
}
```

foreacho

range-based for loops

- Users need to know fewer (internal) details about a class.
  - begin, end
- This makes refactoring easier.

```cpp
const std::vector<int> numbers{2, 3, 5, 7};

for(auto it{numbers.begin()}; it != numbers.end();
    ++it) {
  std::cout << *it << '\n';
}
```

foreach0

```cpp
const std::vector<int> numbers{2, 3, 5, 7};

for(const auto &  e : numbers) {
  std::cout << e << '\n';
}
```

foreach5

range-based for loops

- Classes can be made range-based for loop ready:
  - A class must provide the two functions, begin and end.
  - Or the two functions begin, and end must exist as free functions.
- Source code becomes less and more readable on the using side.
- It contains a slight chance for optimization, as the data are provided uniformly now.
- It is a good way to prevent buffer overflows or at least fix them globally.

```cpp
template<class T, size_t SIZE>
class MyArrayWrapper {
  T       data[SIZE]{};
  size_t size{};

public:
  MyArrayWrapper(std::initializer_list<T> l)
    : size{std::min(l.size(), SIZE)}
  {
    std::copy_n(l.begin(), size, data);
  }

  T* begin() { return &data[0]; }
  T* end() { return &data[size]; }
};

void Use()
{
  MyArrayWrapper<int, 10> arr{2, 3, 4, 5};

  for(const int& i : arr) { printf("%d\n", i); }
}
```

forLoop0

https://AndreasFertig.com
post@AndreasFertig.com

Exercise

a) In exRangeBasedForLoop.cpp , create a container (for example, std::vector) with the numbers $1 - 10$ and print the container's contents afterward.

b) Square each element in the container in a second step and save the result at the container's same position.

c) Use std::for_each to square the elements and print the container again.

- Solution: solRangeBasedForLoop.cpp

d) exSimpleList.cpp implements a rudimentary single-linked list. Extend the list implementation so that it can be used with a range-based for-loop.

- Solution: solSimpleList.cpp

How noexcept works

- C++11 brings the keyword: **noexcept**.
- Functions marked with **noexcept** will throw no exceptions!
  - Instead, they call directly std::terminate.
  - How does this work?

```
void Fun() noexcept(true) { int i = 3; }

void Fun2() noexcept(false) { int i = 3; }
```

noexcept4

How noexcept works

- C++11 brings the keyword: **noexcept**.
- **noexcept** brings some optimization opportunities.
  - The initialization A can be skipped because the destructor cannot observe it.

```cpp
void Fun(int);

void Tick() noexcept;

struct Apple {
  int i;
  ~Apple() { Fun(i); }
};

void c(int i)
{
  Apple obj{i};   // A Can be skipped

  Tick();

  obj.i = 4;
}
```

noexcept2

How noexcept works

- C++11 brings the keyword: **noexcept**.
- **noexcept** brings some optimization opportunities.
  - In g is no unwinding happening which allows the omission of the frame information for the callee.

```cpp
struct B {
  ~B();
};

void f();

void g() noexcept
{
  B b1{};

  f();

  B b2{};
}
```

noexcept3

https://AndreasFertig.com
post@AndreasFertig.com

if & switch with initialization

C++17

- **if** and **switch** can now declare variables similar to **for** loops and initialize them.
  - Variables can thus be declared in the smallest range.
  - Variables are valid until the end of the complete **if**.
  - Caution: Variables *without* names are destroyed directly!
- Overview:

```
for(   init; condition; expression){}
for(   init; range-decl : expression) {} // C++20
if(    init; condition) {}
switch(init; condition) {}
```

```cpp
void ChangeScreen(Screen& newScreen)
{
  if(std::lock_guard lock{gMutex};
     screen == &newScreen) {
    return;
  } else {
    screen = &newScreen;  // still in locked scope
  }

  // not part of locked scope
  SendUpdateNotificationEvent();
}
```

ifSwitchInito

Default member initialization

- With C++11, class members can be initialized inline with a default.
  - An initialization in the constructor is no longer necessary.
  - If a member is also initialized in the constructor, this initialization wins.

```cpp
class DInit {
  int              i{5};
  std::vector<int> v{2, 3, 4};
  std::string      s{"Hello"};

public:
  DInit() = default;

  DInit(const std::vector<int>& _v)
  : v{_v}
  {}

  DInit(int _i)
  : i{_i}
  {}

  DInit(const std::string& _s)
  : s{_s}
  {}
};
```

defaultMemberInito

Exercise

a) Simplify the constructors in `exClassMemberDefaultInit.cpp` by using default member initialization.

- Solution: `solClassMemberDefaultInit.cpp`

Constructor inheritance

- `using` can be used to inherit all constructors from a direct base class.
  - The default-, copy- and move-constructor is not inherited!
  - Members of the inheriting class are not initialized!

```cpp
class TextFormatter {
  std::string mValue{};

public:
  explicit TextFormatter(std::string val)
  : mValue{val}
  {}

  virtual ~TextFormatter() = default;

  virtual bool InsertNewLine() const { return true; }
};

class RawFormatter : public TextFormatter {
  int mData{};

public:
  A Get ctor's of base - class
  using TextFormatter::TextFormatter;

  bool InsertNewLine() const override { return false; }
};

RawFormatter braces{"Hello"};
```

usingAndConstructors0

https://AndreasFertig.com
post@AndreasFertig.com

Exercise

a) Derive **`public`** from a class `Base` and use constructor inheritance to bring the constructor `Base(int)` into the derived class.

b) In addition, derive **`protected`** and **`private`** from the base class. How does this change the visibility of the inherited constructor?

c) Is there a difference if you move the **`using`** statement into a different section?

- Solution: `solCtorInheritance.cpp`

`static` or `inline`

- What is the difference of **`static`** vs. **`inline`**

`static` Generate the function in every translation unit and don't share it.

`inline` suppress the one definition rule (ODR) for this function, such that each translation unit can provide its own copy of the functions' definition. The compiler either inlines the calls or ensures that the multiple definitions get merged.

- https://andreasfertig.blog/2023/03/static-inline-or-an-unnamed-namespace-whats-the-difference/

```cpp
#ifndef _VS_INLINE_H
#define _VS_INLINE_H

static int StaticFun() { return 42; }

inline int InlineFun() { return 42; }

#endif /* _VS_INLINE_H */
```

staticVsInline0

```cpp
#include "staticVsInline0.h"

int Something() { return InlineFun(); }
```

staticVsInline0

```cpp
#include "staticVsInline0.h"

static int Nonsense() { return 5; }

int main() { return InlineFun(); }
```

staticVsInline0

C++17

CTAD

- class template argument deduction (CTAD) helps the compiler automatically perform even more instantiations for class templates.
- CTAD saves us from naming types more than once.
- Deduction guides are preferred by the compiler when things are specialized.

```cpp
int main()
{
  // A Before C++17 <int> was mandatory
  std::vector<int> x{2, 3, 4};

  // B With CTAD in C++17 the compiler detects the type
  std::vector y{2, 3, 4};

  x = y;
}
```

ctado

C++17

CTAD or make_NNN function?

- C++17 added CTAD to the language.
  - This makes `make_NNN` functions like `make_pair` obsolete.
  - How does CTAD perform?

```cpp
template<typename F, typename S>
struct pair {
  F first;
  S second;
};

template<typename F, typename S>
inline auto make_pair(F&& f, S&& s)
{
  // Production code should use std::remove_cvref_t
  return pair<F, S>{std::forward<F>(f),
                    std::forward<S>(s)};
}

// Production code should use std::remove_cvref_t
template<typename F, typename S>
pair(F&& f, S&& s) -> pair<F, S>;
```

ctadOrMakeFunctions0

Things to remember

- Use the `initializer_list` to initialize objects.
- Avoid class operators or special member functions with an `initializer_list` as a parameter.
- In the case of constructor inheritance, the members of the inheriting class are *not* initialized.
- Prefer CTAD over `make_NNN` functions.

2. constexpr

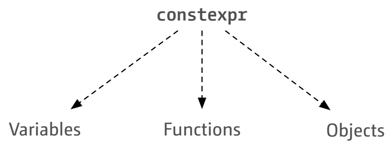
https://AndreasFertig.com
post@AndreasFertig.com

Compile-time vs. runtime

- Compile-time
  - Typically introduced by **`constexpr`**.
  - The compiler does the heavy lifting *once*.
  - Users can profit from faster code.
  - With things calculated at compile-time, the binary size can be smaller, possibly resulting in more features for the users.
  - Not having to calculate the same value repeatedly can reduce energy consumption, and mobile users will thank you for that.
  - Doing things at compile-time increases compile-times.
  - Values calculated at compile-time can be checked at compile-time. Issues there never leave the factory.
- Run-time
  - The default mode.
  - Your compilation process is fast, which can be valuable if you compile a large code-base.

`constexpr`

- New since C++11.
- Functions or variables can be marked as **`constexpr`**.
- There are special rules for **`constexpr`**:
  - Variable initializers must be known at compile time.
- The following applies to **`constexpr`** functions:
  - Function *can* be evaluated at compile-time, *if* all input values are known.
- Since C++17 applies:
  - Lambdas are implicit **`constexpr`** *if* they meet the requirements of **`constexpr`**.
  - Compiler-generated constructors are also **`constexpr`**.



constexpr - Functions

- Functions marked as **constexpr** come with some limitations:
  - Only literal types are allowed as parameters and return types.
  - Before C++20 a member function must not be **virtual**.
  - Before C++14 only one **return** statement was allowed.
  - Catching exceptions is not allowed. Throwing is allowed since C++14.
- In the body of the function, some things are not allowed:
  - **goto** [1]
  - **try-catch** block[2]
  - inline assembler[2]
  - Uninitialized variables[2]
- **constexpr** functions and static class members are *implicitly* **inline**.
  - Definition and implementation can not be divided into different files.
- non-static **constexpr** member functions:
  - Only in C++11, **constexpr** member functions are implicitly **const**.

[1] Allowed since C++23
[2] Allowed since C++20

constexpr - Evolution

		11	14	17	20	23
1	`void` as return-type	-	✓	✓	✓	✓
2	More than just a single `return`	-	✓	✓	✓	✓
3	Using `throw`[1]	-	✓	✓	✓	✓
4	`try`/`catch`-Block	-	-	-	✓	✓
5	`constexpr` member-function implicitly `const`	✓	-	-	-	-
6	`inline` for `static` members with `constexpr`	-	-	✓	✓	✓
7	Lambda can be implicitly `constexpr`	-	-	✓	✓	✓
8	`new` / `delete` in `constexpr` functions	-	-	-	✓	✓
9	`constexpr virtual` member functions	-	-	-	✓	✓
10	inline asm[1]	-	-	-	✓	✓
11	Uninitialized variable[1]	-	-	-	✓	✓
12	`static` variable in `constexpr` function	-	-	-	-	✓
13	`goto` in in a `constexpr` function[1]	-	-	-	-	✓
14	Cast to `void*`	-	-	-	-	✓
15	Conditions for `constexpr` functions are only check in `constexpr` context	-	-	-	-	✓

[1] May not be called on the `constexpr` path.

https://AndreasFertig.com
post@AndreasFertig.com

`constexpr` - Example

- *Can* be evaluated by the compiler already at compile time.
  - This allows calculations to be moved from runtime to compile time.
  - Can help to avoid holding more storage than required.

```cpp
template<size_t N>
 auto make_fixed_string(const char (&str)[N])
{
  return FixedString<N>{str};
}

 static const FixedString<50> x{
  "Hello, embedded World!"};
 static const auto y{
  make_fixed_string("Hello, some other planet!")};
```

constexprFixedString0

```cpp
template<size_t N>
class FixedString {
  size_t mLength{};
  char   mData[N]{};

public:
  FixedString() = default;
   FixedString(const char* str)
  : mLength{std::char_traits<char>::length(str)}
  {
    std::copy_n(str, size(), mData);
  }

   size_t size() const { return mLength; }

  const char* data() const
  {  // std::string_view would be better
    return mData;
  }
};
```

constexprFixedString0

Exercise

a) Have a look at `exConstexprApply.cpp` . Compile the code and check its assembly output. Also, ensure that you use pure C++, without C extensions (requires `–Wpedantic`).

You can find the compiler-option in the source code. The dash in the PDF is a UTF-8 character and cannot be copied!

b) Apply **`constexpr`** and change datatypes if necessary to improve the code such that its assembly lines go down.

- Solution: `solConstexprApply.cpp`

Things to remember

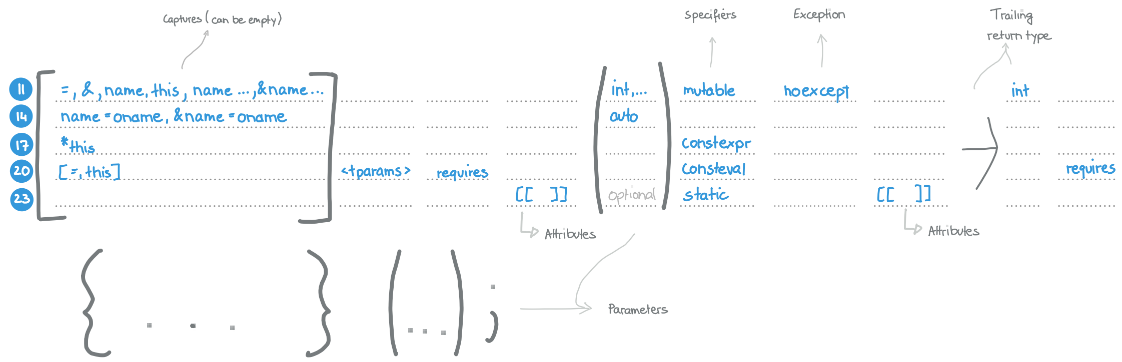
- A `constexpr` function *can* be evaluated at compile-time, *if* all input values are known.

3. Lambdas

https://AndreasFertig.com
post@AndreasFertig.com

Lambda expressions

- Also called anonymous functions.
- Allows defining and executing functions within the local scope.
- Functionality can thus be clearly limited and delineated.



- What is the implementation under the hood?

Lambda captures

- The different ways to capture a variable into a lambda.

	Capture	Description	11	14	17	20
1	`[]`	Empty lambda	✓	✓	✓	✓
2	`[apple]`	Copy `apple`	✓	✓	✓	✓
3	`[&apple]`	`apple` as reference	✓	✓	✓	✓
4	`[=]`	Copy all variables used in the lambda body	✓	✓	✓	✓[1]
5	`[&]`	All variables used in the lambda body as references	✓	✓	✓	✓
6	`[=,&apple]`	All variables used in the lambda body as copy, but `apple` by copy	✓	✓	✓	✓
7	`[this]`	Data and members of the surrounding class as references	✓	✓	✓	✓
8	`[*this]`	Data and members of the surrounding class as deep copy			✓	✓
9	`[=, *this]`	Copy all variables used in the lambda body, `this` as deep copy			✓	✓
10	`[=, this]`	Copy all variables used in the lambda body, `this` by reference				✓
11	`[fruit = apple]`	Create `fruit` as new variable (copy) initialized by `apple`		✓	✓	✓
12	`[&fruit = apple]`	Create `fruit` as new variable (reference) initialized by `apple`		✓	✓	✓
13	`[...y = pack]`	Create y as new pack, initialized by a pack				✓

[1] C++20: Deprecated when used inside a class.

https://AndreasFertig.com
post@AndreasFertig.com

Lambdas applied

- Lambdas can be used in different areas:
  - stored in a variable.
  - as the return type of a function.
  - as a parameter for a function.

```cpp
auto lambdaVariable = [](int x, int y) {
  return x + y;
};

auto LambdaAsReturnObject()
{
  return [](int x, int y) { return x + y; };
}

template<typename T>
void LambdaAsArgument(T&& lambda)
{
  lambda();
}

void Use()
{
  const int a = lambdaVariable(2, 3);

  const auto lr = LambdaAsReturnObject();
  const int  b  = lr(2, 3);

  LambdaAsArgument([] { puts("Hello, world!"); });
}
```

lambdaVariants0

C++14

Generic lambdas

- Lambdas can determine the type of their arguments, like templates.
  - It allows for more generic lambdas.
  - A mixture of `auto` and named type is possible.
- What is the implementation of lambdas under the hood?

```cpp
auto lambdaVariable = [](auto x, auto y) {
  return x + y;
};

int main()
{
  const double res = lambdaVariable(2.0, 3.0);

  return lambdaVariable(2, 3);
}
```

genericLambda0

https://AndreasFertig.com
post@AndreasFertig.com

Lambdas

- Beware of unused lambdas.
  - Lambdas, which capture the variables by copy, are *always initialized*.
  - Regardless of whether they are *executed*.

```cpp
int main()
{
  std::string fun{};

  auto a = [=] { printf("%s\n", fun.c_str()); };

  auto b = [=] {};

  auto c = [fun] { printf("%s\n", fun.c_str()); };

  auto d = [fun] {};

  auto e = [&fun] { printf("%s\n", fun.c_str()); };

  auto f = [&fun] {};
}
```

lambda7

Lambdas applied

- Where / how can lambdas be useful?
  - If additional functionality is required before and / or after a code fragment.

```cpp
template<typename T>
void DoLocked(T&& action)
{
  std::lock_guard _{gMutex};

  action();
}

void Use()
{
  DoLocked([] { puts("Hello"); });
}
```

lambdaUse0

https://AndreasFertig.com
post@AndreasFertig.com

Lambdas applied

- Where / how can lambdas be useful?
  - To achieve more constness within functions. This technique is called IIFE.

```cpp
void Fun(int e)
{
  const std::string name{[&] {
    switch(e) {
      case 0: return "void"s;
      case 1: return globalName;
      case 2: return "bool"s;
      default: return ""s;
    }
  }()};

  printf("name: %s\n", name.c_str());
}
```

lambdaUse1

Lambdas applied

- Where / how lambdas can be useful?
  - Clean up / release resources.

```cpp
size_t ReadData(span<char> buffer)
{
  int fd = Open(/*some well known file*/);

  if(-1 == fd) { return 0; }

  const auto len =
    read(fd, buffer.data(), buffer.size());

  if(-1 == len) { return 0; }

  ftruncate(fd, len);

  close(fd);

  return gsl::narrow_cast<size_t>(len);
}
```

final1

https://AndreasFertig.com
post@AndreasFertig.com

Lambdas applied

- Where / how lambdas can be useful?
  - Clean up / release resources.

```
size_t ReadData(span<char> buffer)
{
  int             fd = Open(/*some well known file*/);
   FinalAction cleanup{[&] {
    if(-1 != fd) { close(fd); }
  }};


    if(-1 == fd)
  {
    return 0;
  }

  const auto len =
    read(fd, buffer.data(), buffer.size());

  if(-1 == len) { return 0; }

  ftruncate(fd, len);

  return gsl::narrow_cast<size_t>(len);
}
```

final5

Lambdas applied

- Where / how lambdas can be useful?
  - Clean up / release resources.

```
template<typename T>
class FinalAction {
  T mAction;

public:
  explicit FinalAction(T&& action) noexcept
  : mAction{std::forward<T>(action)}
  {}

  ~FinalAction() noexcept { mAction(); }
};
```

final5

```
size_t ReadData(span&lt;char&gt; buffer)
{
  int             fd = Open(/*some well known file*/);
   FinalAction cleanup{[&] {
    if(-1 != fd) { close(fd); }
  }};


    if(-1 == fd)
  {
    return 0;
  }

  const auto len =
    read(fd, buffer.data(), buffer.size());

  if(-1 == len) { return 0; }

  ftruncate(fd, len);

  return gsl::narrow_cast&lt;size_t&gt;(len);
}
```

final5

https://AndreasFertig.com
post@AndreasFertig.com

Exercise

a) `exLambdaUB.cpp` contains undefined behavior. Find and correct it.
   - Solution: `solLambdaUB.cpp`

b) The implementation of `Use` in `exLambdaCapture.cpp` is missing. Use the `std::for_each` algorithm and double all values in a `std::vector` which are below a certain threshold, passed to `Use`. Print the results to verify the operation.
   - Solution: `solLambdaCapture.cpp`

Things to remember

- Do not return a lambda that uses local variables as a reference.

https://AndreasFertig.com
post@AndreasFertig.com

4. Speedy Templates

Variadic templates

- Syntax:

(A) `typename|class... Ts` generates a type template parameter pack with an optional name.

(B) `Args... ts` a function argument parameter pack with an optional name.

(C) `sizeof...(ts)` determine the number of arguments passed.

(D) `ts...` in the body of a function to unpack the arguments.

```cpp
template<typename T,
         typename... Ts  // A Variadic template
         >
constexpr auto
min(const T& x,
    const T& y,
    const Ts&... vals)  // B Parameter pack
{
  const auto m = x < y ? x : y;
  if constexpr(
    // C size of a pack
    sizeof...(vals) > 0) {

    // D Expand the pack
    return min(m, vals...);

  } else {
    return m;
  }
}

static_assert(min(3, 2) == 2);
static_assert(min(3, 2, 3, 4, 5) == 2);
```

variadicTemplateMultiMino

Variadic templates

- With C++11, there are *variadic templates*:
  - Variadic templates are templates that take any number of parameters.
  - Already known by variadic macros or variadic functions.

```
A Helper functions to convert everything into a std::string
auto Normalize(const std::string& t) { return t; }
auto Normalize(const QString& t) { return t.toStdString(); }
auto Normalize(const char* t) { return std::string_view{t}; }

B Catch all others and apply to_string
template<class T> auto Normalize(const T& t) { return std::to_string(t); }

template<typename T, typename... Ts> auto _StrCat(std::string& ret, const T& targ, const Ts&... vals)
{
  ret += Normalize(targ);
  if constexpr(sizeof...(vals) > 0) {
    _StrCat(ret, vals...);  C Do, as long as the pack has elements
  }
}

template<typename T, typename... Ts> auto StrCat(const T& targ, const Ts&... vals)
{
  std::string ret{Normalize(targ)};

  _StrCat(ret, vals...);  D Start the recursion to expand the pack

  return ret;
}
```

strCat0

Exercise

a) `exVariadicTemplateSum.cpp` : Write a function template that accepts any number of parameters and returns the sum of the values.
   - Solution: `solVariadicTemplateSum.cpp`

b) Make sure that the add function from `solVariadicTemplateSum` can only be used with positive integral data types. A *helpful* error message should appear at compile time if the rule is violated.
   - Solution: `solTypeTraitsAssert.cpp`

c) Calculate the sum of values always at compile time using variadic templates.
   - Solution: `solVariadicTemplateSumConstexpr.cpp`

C++17

Fold Expressions

- Used to unpack a parameter pack using an operation.
  - Saves the recursion.
- Syntax:
  - unary { right fold: (pack *op* ...) / left fold: (... *op* pack)
  - binary { right fold: (pack *op* ... *op* init) / left fold: (init *op* ... *op* pack)
- Note:
  - All *op* must be the same operation.
  - op: $+, -, *, /, \%, \hat{}, \&, |, =, <, >, <<, >>, +=, -=, *=, /=, \%=, \hat{}=, \&=, |=, <<=, >>=, ==, !=, <=, >=, \&\&, ||, ,, .*, ->*$
  - Parentheses around the expression are required to make it a fold expression.

```cpp
template<typename... Ts>
constexpr auto avg(const Ts&... vals)
{
  return (vals + ...) / sizeof...(vals);
}

static_assert(avg(2, 3, 4) == 3);
```

foldExpression0

```cpp
template<typename T, typename... Args>
void push_back(std::vector<T>& v, Args&&... vals)
{
  v.reserve(v.size() + sizeof...(vals));

  (v.push_back(std::forward<Args>(vals)), ...);
}

void Use()
{
  std::vector v{2, 3, 4};
  const int   z = 5;

  push_back(v, z, 6, 7);
}
```

foldExpression1

C++17

Variable templates

- Variables can now also become templates.
  - With them, we can define constants like π or `true_type`
- This makes some template metaprogramming (TMP) code more readable.
  - An alias template, as in B, is only an alias.

```cpp
A Helper to store a value at compile-time
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
};

B Aliases for clean TMP
using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

C Base is_pointer template
template<class T>
struct is_pointer : false_type {};

D is_pointer specialization for T*
template<class T>
struct is_pointer<T*> : true_type {};

E Test it
static_assert(is_pointer<int*>::value);
static_assert(not is_pointer<int>::value);
```

typeTraitsIsPointer0

Variable templates

C++17

- Variables can now also become templates.
  - With them, we can define constants like π or `true_type`
- This makes some TMP code more readable.
  - An alias template, as in B, is only an alias.
  - With this new version, B defines a new variable.
  - The two together make TMP much more readable in many places.

```cpp
// A As seen before
template<class T, T v>
struct integral_constant {
  static constexpr T value = v;
};

using true_type  = integral_constant<bool, true>;
using false_type = integral_constant<bool, false>;

template<class T>
struct is_pointer : false_type {};

template<class T>
struct is_pointer<T*> : true_type {};

// B A variable template to access ::value
template<typename T>
constexpr inline auto is_pointer_v =
  is_pointer<T>::value;

// C is_pointer_v looks cleaner than ::value
static_assert(is_pointer_v<int*>);
static_assert(not is_pointer_v<int>);
```

integralConstanto

Exercise

a) Write a function template in `exVariadicTemplateSumSameType.cpp` that accepts any number of parameters of the same type and returns the values' sum. You may use a variable template `are_same_v` to store the value of a type-trait from the Standard Template Library (STL).
   - Solution: `solVariadicTemplateSumSameType.cpp`

https://AndreasFertig.com
post@AndreasFertig.com

Only use rvalue references in templates when needed

- RValue references allow move semantics which can lead to faster code.
- In templates, we may end up with instantiations for `&` and `const &`.
- If the values that get passed in are consumed in the template (e.g. for formatting), there is no benefit of rvalue references.
  - They only lead to another template instantiation.
  - Try to use `const &` as a parameter to save template instantiations

Use `constexpr if` instead of `enable_if`

- A simple and probably easy-to-read code example for us humans.
  - It is clear to us that there are two `process` functions.
  - For the compiler, the `enable_if` isn't that clear.
  - It must instantiate both functions.

```cpp
template<typename T>
std::enable_if_t<std::is_integral_v<T>>
process(T&& value)
{
  puts("integral");
}

template<typename T>
std::enable_if_t<not std::is_integral_v<T>>
process(T&& value)
{
  puts("not integral");
}
```

useConstexprIfInsteadOfEnableIfo

https://AndreasFertig.com
post@AndreasFertig.com

Use constexpr if instead of enable_if

- A simple and probably easy-to-read code example for us humans.
  - It is clear to us that there are two `process` functions.
  - For the compiler, the `enable_if` isn't that clear.
  - It must instantiate both functions.
- Thanks for **`constexpr if`** we can reduce this effort and also improve readability.
  - Functions that belong together are now together.
  - Distinctions become clear.

```cpp
template<typename T>
void process(T&& value)
{
  if constexpr(std::is_integral_v<T>) {
    puts("integral");
  } else if constexpr(not std::is_integral_v<T>) {
    puts("not integral");
  }
}
```

useConstexprIfInsteadOfEnableIf1

C++17

Recursion vs. Fold Expressions

- Recursion together with **`constexpr if`** is already better than `enable_if`.
- However, the compiler needs to create N functions where N is the number of parameters.

```cpp
template<typename T, typename... Ts>
void Print(const T& targ, const Ts&... vals)
{
  std::cout << ' ' << targ;

  if constexpr(sizeof...(vals) > 0) { Print(vals...); }
}

int main() { Print("Hello", "C++", 20); }
```

comparisonRecursionVsFoldExpression0

Recursion vs. Fold Expressions

C++17

- With fold expressions, the compiler needs to create only 1 function.

```cpp
template<typename... Ts>
void Print(const Ts&... vals)
{
  (..., (std::cout << ' ' << vals));
}

int main() { Print("Hello", "C++", 20); }
```

comparisonRecursionVsFoldExpression1

Recursion vs. Fold Expressions

C++17

- With fold expressions, the compiler needs to create only 1 function.
- Even when the compiler needs to create a lambda inside for readability, fold expressions are faster for a larger N.

```cpp
template<typename... Ts>
void Print(const Ts&... vals)
{
  auto coutSpaceAndArg = [](const auto& arg) {
    std::cout << ' ' << arg;
  };

  (..., coutSpaceAndArg(vals));
}

int main() { Print("Hello", "C++", 20); }
```

comparisonRecursionVsFoldExpression2

Improve compile times with variable templates

C++14

- Historically, we often use **`struct`**s in TMP.
  - The compiler doesn't know we want to use such a type only for compile-time information.
  - It has to generate a full class, with all bells and whistles.

```
template<int N>
struct test {
  static constexpr int value = N;
};

int x0 = test<0>::value;
int x1 = test<1>::value;
```

variableTmplOverStructs0

Improve compile times with variable templates

C++14

- Historically, we often use **`struct`**s in TMP.
  - The compiler doesn't know we want to use such a type only for compile-time information.
  - It has to generate a full class, with all bells and whistles.
- Since C++14, we can use variable templates instead.
  - Here, the compiler doesn't need to generate a class.
  - It only needs to create a cheap **`bool`**.
  - You can improve compile times by using variable templates in such cases.
  - ... and readability as well.

```
template<int N>
constexpr inline int test = N;

int x0 = test<0>;
int x1 = test<1>;
```

variableTmplOverStructs1

`extern template`

- Use **`extern template`** to tell the compile to not fully instantiate a template at this point because it is provided somewhere else.
  - That way, only the data members get instantiated as the compiler needs to know the class size.
  - All member functions are not instantiated at this point.

```cpp
template<typename T>
struct S {
  T Fun() { return data; }

  T data;
};

extern template struct S<int>;
```

externTemplate0

Omit the body if possible

- In this example, `impl` is used to gather a datatype stored in `type`.
  - We force the compiler to instantiate `impl` and create its body.
  - This costs compile time.

```cpp
// This class contains some information of a type.
template<typename>
class trait {};

// Helper template mapping an index to a type.
template<template <int> class TypeMap, int N>
struct get_type_traits;

template<int> struct type_map;
template<> struct type_map<0> { using type = int; };
template<> struct type_map<1> { using type = float; };

template <template <int> class TypeMap, int N>
struct get_type_traits {
private:
  template<int... I>
  static auto impl(std::integer_sequence<int, I...>) {
    return std::make_tuple(
            trait<typename TypeMap<I>::type>{}...);
  }

public:
  using type = decltype(
          impl(std::make_integer_sequence<int, N>{}));
};

// Should be 'std::tuple<trait<int>, trait<float>>'.
using type_traits = get_type_traits<type_map, 2>::type;
```

templatesTypeOnly0

Omit the body if possible

- In this example, `impl` is used to gather a datatype stored in `type`.
  - We force the compiler to instantiate `impl` and create its body.
  - This costs compile time.
- In such a case, move away from **`auto`** as the return type.
  - By putting the calculated type directly as a return type, we can omit the function body.
  - This spares the compiler from creating a function body we never use.

```cpp
// This class contains some information of a type.
template<typename>
class trait {};

// Helper template mapping an index to a type.
template<template <int> class TypeMap, int N>
struct get_type_traits;

template<int> struct type_map;
template<> struct type_map<0> { using type = int; };
template<> struct type_map<1> { using type = float; };

template <template <int> class TypeMap, int N>
struct get_type_traits {
private:
    template<int... I>
    static std::tuple<trait<
        typename TypeMap<I>::type>...>
    impl(std::integer_sequence<int, I...>);

public:
  using type = decltype(
          impl(std::make_integer_sequence<int, N>{}));
};

// Should be 'std::tuple<trait<int>, trait<float>>'.
using type_traits = get_type_traits<type_map, 2>::type;
```

templatesTypeOnly1

C++20

Concepts subsumption - Compile times

- Concepts have the ability to subsume.
  - This requires an SAT solver.
  - Subsume checking can be expensive.

```cpp
template<typename T>
concept has_leisure_time = requires(T t)
{ t.hasLeisureTime(); };

template<typename T>
concept has_fun = requires(T t) { t.hasFun(); };

template<typename T>
concept has_hobby = requires(T t) { t.hasHobby(); };

struct A {
  void hasLeisureTime() {}
  void hasFun() {}
};

struct B {
  void hasLeisureTime() {}
  void hasFun() {}
  void hasHobby() {}
};

template<typename T>
requires(has_leisure_time<T> and has_fun<T> and
         not has_hobby<T>)
void Fun(T) { puts("fun only"); }

template<typename T>
requires(has_leisure_time<T> and has_fun<T> and
           has_hobby<T>)
void Fun(T) { puts("fun and hobby"); }
```

conceptsSubsumeVsConstexprIfo

C++20

Concepts subsumption - Compile times

- Concepts have the ability to subsume.
  - This requires an SAT solver.
  - Subsume checking can be expensive.
  - Check whether `constexpr if` is a viable alternative to speed up the compile times of your project.

```cpp
template<typename T>
concept has_leisure_time = requires(T t)
{ t.hasLeisureTime(); };

template<typename T>
concept has_fun = requires(T t) { t.hasFun(); };

template<typename T>
concept has_hobby = requires(T t) { t.hasHobby(); };

struct A {
  void hasLeisureTime() {}
  void hasFun() {}
};

struct B {
  void hasLeisureTime() {}
  void hasFun() {}
  void hasHobby() {}
};

void Fun(has_leisure_time auto t) {
  constexpr bool hasHobby = requires { t.hasHobby(); };
  constexpr bool hasFun   = requires { t.hasFun(); };

  if constexpr(hasFun and not hasHobby) {
    puts("fun only");
  } else if constexpr(hasFun and hasHobby) {
    puts("fun and hobby");
  }
}
```

conceptsSubsumeVsConstexprIf1

Things to remember

- Prefer fold expressions over recursive variadic templates.
- Use variable templates to make TMP more readable.
- Use `const &` as a parameter over rvalue references in a template as long as you don't take profit from move semantics.
- Prefer `constexpr if` over `enable_if` to improve readability and compile-times.
- Prefer variable templates over struct templates for compile-time information storage.
- Use `extern template` whenever you want to prevent a full instantiation at this point and you know a template is instantiated somewhere else.
- In case only a datatype is needed, omit the function body and use only the return type.
- Use concept subsumption only when necessary. Prefer `constexpr if` together with concepts.

5. Move semantics

The value categories

lvalue **rvalue**

```
Object base{};
Object obj2 = base;
Object obj3 = GetValue();
Object obj4 = std::move(base);
Object obj5 = GetOtherValue();
Object obj6 = 5;
```


https://AndreasFertig.com
post@AndreasFertig.com

The value categories

lvalue

xvalue

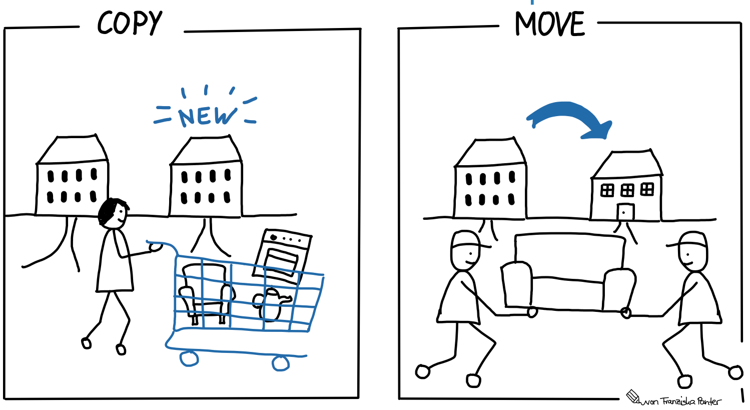
```
Object base{};
Object obj2 = base;
Object obj3 = GetValue();         // Object&& GetValue();
Object obj4 = std::move(base);
Object obj5 = GetOtherValue();    // Object GetOtherValue();
Object obj6 = 5;
```

glvalue := lvalue or xvalue

prvalue

Andreas Fertig v1.1 Modern C++: When Efficiency Matters 76

Move semantics: move or duplicate




```
void Copy(char** dst, char** src, size_t size)
{
  *dst = new char[size];
  memcpy(*dst, *src, size);
}
```

moveVsCopy0

```
void Move(char** dst, char** src)
{
  *dst = *src;
  *src = nullptr;
}
```

moveVsCopy1

Andreas Fertig v1.1 Modern C++: When Efficiency Matters 77

std::move

- std::move doesn't move! It is an rvalue cast.
- Explicit moving of a resource can be forced with std::move.
  - Useful if it is absolutely clear that a move is allowed.
  - The source object of std::move is then in an unknown but valid state.
  - You must only call member functions that have no precondition. Otherwise, it is *undefined behavior*!
  - Safe are the following two operations: assign, delete.
- Mostly: std::move should be used rarely. The default behavior is usually already correct. [3]
- Move semantics has a higher priority than copy semantics!

```cpp
struct DynInt {
  int* mData{};

  DynInt(int v) : mData{new int{v}} {}

  ~DynInt() { delete mData; }

  DynInt(const DynInt& rhs)
  : mData{new int{*rhs.mData}} {}

  DynInt(DynInt&& rhs)
  : mData{rhs.mData}  // Use std::exchange
  { rhs.mData = nullptr; }
};

struct Holder {
  DynInt mData;

  Holder(const DynInt& rhs)
  : mData{rhs}
  { puts("Holder copy ctor"); }

  Holder(DynInt&& rhs)
  : mData{static_cast<DynInt&&>(rhs)}
  { puts("Holder move ctor"); }
};

void Fun(Holder);

void Use() { Fun(DynInt{5}); }
```

moveSemantics0

Move semantics: move or duplicate

- A construct as it is often used.
- Item 7: *Declare destructors virtual in polymorphic base classes.* [4]

```cpp
class Base
{
public:
    virtual ~Base() {}

    virtual void Fun() = 0;
};
```

moveAndVirtual0

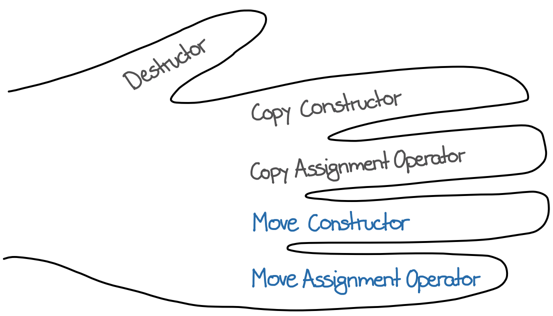
Special member functions and their dependencies

		compiler implicitly declares					
				copy		move	
user declares		**default ctor**	**dtor**	**ctor**	**assignment**	**ctor**	**assignment**
	Nothing	defaulted	defaulted	defaulted	defaulted	defaulted	defaulted
	Any ctor	not declared	defaulted	defaulted	defaulted	defaulted	defaulted
	default ctor	*user declared*	defaulted	defaulted	defaulted	defaulted	defaulted
	dtor	defaulted	*user declared*	defaulted	defaulted	not declared	not declared
copy	**ctor**	not declared	defaulted	*user declared*	defaulted	not declared	not declared
copy	**assignment**	not declared	defaulted	defaulted	*user declared*	not declared	not declared
move	**ctor**	not declared	defaulted	deleted	deleted	*user declared*	not declared
move	**assignment**	not declared	defaulted	deleted	deleted	not declared	*user declared*

Source: [5]

Move semantics: The rule of ...

The rule of ~~three~~ five




```cpp
class Base
{
public:
    virtual ~Base() = default;
    Base(const Base& ) = default;
    Base(      Base&&) = default;

    Base& operator=(const Base& ) = default;
    Base& operator=(      Base&&) = default;

    virtual void Fun() = 0;
};
```

moveAndVirtual1

Exercise

In the program, `exMoveSemantics.cpp` , a data type `BigArray` with 1 billion elements is moved into a `std::vector`.

a) Compile the program and measure the runtime.
   Note: On Windows, you might need to compile the program with 64-bit or decrease the size.
   For cmake: `cmake -DCMAKE_GENERATOR_PLATFORM=x64`
b) Extend the class `BigArray` by move semantics and rerun the program. What is the difference in speed?
   - Solution: `solMoveSemantics.cpp`

The STL, move, and custom object

- Here, we have a `std::vector` and a custom object.
  - What is the output?

```cpp
struct Lifeguard {
  Lifeguard() { puts("ctor"); }
  Lifeguard(const Lifeguard&) { puts("copy ctor"); }
  Lifeguard(Lifeguard&&) { puts("move ctor"); }
  Lifeguard& operator=(const Lifeguard&)
  {
    puts("copy assign");
    return *this;
  }
  Lifeguard& operator=(Lifeguard&&)
  {
    puts("move assign");
    return *this;
  }
};

void Use()
{
  std::vector<Lifeguard> v{};

  v.push_back(Lifeguard{});

  puts("second element");
  v.push_back(Lifeguard{});
}
```

stdVectorAndNoexcepto

The STL, move, and custom object

- Here, we have a `std::vector` and a custom object.
  - What is the output?
- If we mark the move constructor **`noexcept`**, things change.

```cpp
struct Lifeguard {
  Lifeguard() { puts("ctor"); }
  Lifeguard(const Lifeguard&) { puts("copy ctor"); }
  Lifeguard(Lifeguard&&)  noexcept
  {
    puts("move ctor");
  }
  Lifeguard& operator=(const Lifeguard&)
  {
    puts("copy assign");
    return *this;
  }
  Lifeguard& operator=(Lifeguard&&)  noexcept
  {
    puts("move assign");
    return *this;
  }
};

void Use()
{
  std::vector<Lifeguard> v{};

  v.push_back(Lifeguard{});

  puts("second element");
  v.push_back(Lifeguard{});
}
```

stdVectorAndNoexcept1

lvalues and rvalues

- lvalue reference (&):
  - Named objects (variables, parameters, ...).
- rvalue reference (&&)
  - Whenever it is an untitled object.
  - Temporary objects.
  - Nameless objects.
  - Objects whose address we can not determine.
- Forwarding (universal) references:
  - Forwarding references bind lvalues *and* rvalues.
  - The distinction when what is what is hard.
- It is a forwarding reference if

a) The form exactly matches T&&.

and

b) type deduction is required.

```cpp
Apple&& apple1 = Fun();  // rvalue ref
auto&&  apple2 = Fun(); // forwarding ref

void Fun(Apple&& f);     // rvalue ref

template<typename T>
void Fun(Class<T>&& f);  // rvalue ref

template<typename T>
void Fun(T&& f);         // forwarding ref
```

urefo

Perfect forwarding

- The goal of perfect forwarding:
  - To pass on an object while preserving its properties.
  - Mostly found in template code.

```cpp
template<typename T>
void f(T&& t)
{
  fun(t);
}

template<typename T>
void g(T&& t)
{
  fun(std::forward<T>(t));
}

template<typename T>
void h(T&& t)
{
  fun(std::move(t));
}
```

correctForwardo

Utilizing move semantics even more: ref-qualifiers

```cpp
class string {
  size_t                 mLen{};
  std::unique_ptr<char[]> mData{};

  void Concat(const char* s);

public:
  string(const char* data);
  string(const string& rhs);
  string& operator=(const string& rhs);
  string(string&& rhs);
  string& operator=(string&& rhs);
  char*   c_str() const { return mData.get(); }

  string& append(const char* s)
  {
    Concat(s);
    return *this;
  }
};
```

appendAndRefQualifierso

```cpp
string s{"Hello"};
s.append(", world!");

std::cout << s.c_str();
```

appendAndRefQualifierso

Utilizing move semantics even more: ref-qualifiers

```cpp
class string {
  size_t                  mLen{};
  std::unique_ptr<char[]> mData{};

  void Concat(const char* s);

public:
  string(const char* data);
  string(const string& rhs);
  string& operator=(const string& rhs);
  string(string&& rhs);
  string& operator=(string&& rhs);
  char*   c_str() const { return mData.get(); }

  string& append(const char* s)
  {
    Concat(s);
    return *this;
  }
};
```

appendAndRefQualifiers0

```cpp
string s{"Hello"};
s.append(", world!");

std::cout << s.c_str();

string s2 = string{"Hello"}.append(", world!");

std::cout << s.c_str();
```

appendAndRefQualifiers1

Andreas Fertig v1.1 — Modern C++: When Efficiency Matters — 88

Utilizing move semantics even more: ref-qualifiers

```cpp
class string {
  size_t                  mLen{};
  std::unique_ptr<char[]> mData{};

  void Concat(const char* s);

public:
  string(const char* data);
  string(const string& rhs);
  string& operator=(const string& rhs);
  string(string&& rhs);
  string& operator=(string&& rhs);
  char*   c_str() const { return mData.get(); }

  string& append(const char* s) &
  {
    Concat(s);
    return *this;
  }

  string&& append(const char* s) &&
  {
    Concat(s);
    return  std::move(*this) ;
  }
};
```

appendAndRefQualifiers1

```cpp
string s{"Hello"};
s.append(", world!");

std::cout << s.c_str();

string s2 = string{"Hello"}.append(", world!");

std::cout << s.c_str();
```

appendAndRefQualifiers1

Andreas Fertig v1.1 — Modern C++: When Efficiency Matters — 89

Things to remember

- Move semantics has a higher priority than copy semantics.
- Remember to apply `std::move` to *any* parameter within a move constructor or assignment operator, regardless of whether it is *currently* moveable.
- Remember the rule of five.
- Mark your move constructor and assignment operator **`noexcept`** to get the full performance of the STL.

6. The costs of abstractions

https://AndreasFertig.com
post@AndreasFertig.com

static

- `static` is probably best known.
- There are several types of `static`. This is block-local `static`.

```
Singleton& Singleton::Instance()
{
  static Singleton singleton;

  return singleton;
}
```

statico

static

- `static` is probably best known.
- There are several types of `static`. This is block-local `static`.

```
Singleton& Singleton::Instance()
{
  static Singleton singleton;

  return singleton;
}
```

statico

```
Singleton& Singleton::Instance()
{
  static bool __compiler_computed;
  static char singleton[sizeof(Singleton)];

  if(!__compiler_computed) {
    new(&singleton) Singleton;
    __compiler_computed = true;
  }

  return *reinterpret_cast<Singleton*>(&singleton);
}
```

singleton1

Conceptually what the compiler generates.

https://AndreasFertig.com
post@AndreasFertig.com

static - Ab C++11

“[...] If the initialization exits by throwing an exception, the initialization is not complete, so it will be tried again the next time control enters the declaration. If control enters the declaration concurrently while the variable is being initialized, the concurrent execution shall wait for completion of the initialization. If control re-enters the declaration recursively while the [...]”

— N3337 [131] § 6.7 p4 [6]

static - Since C++11

- static is probably best known.
- There are several types of static, this is block-local static.

```cpp
Singleton& Singleton::Instance()
{
  static Singleton singleton;

  return singleton;
}
```

statico

```cpp
Singleton& Singleton::Instance()
{
  static  int   __compiler_computed;
  static char           singleton[sizeof(Singleton)];

  if(!__compiler_computed) {
    if( __cxa_guard_acquire(__compiler_computed)
        ) {
      new(&singleton) Singleton;
      __compiler_computed = true;
       __cxa_guard_release(__compiler_computed);

    }
  }

  return *reinterpret_cast<Singleton*>(&singleton);
}
```

singleton4

Conceptual what the compiler generates.

Smart pointer

- With C++11, there are smart pointers (now really smart ...)
  - They follow the **R**esource **A**cquisition **I**s **I**nitialization (RAII) idiom and manage the resources.
  - Effective remedy against memory leaks.
  - Like a garbage collection for C++.
- They are available in 3 variants for different use cases:

`std::unique_ptr` or Highlander pointer, according to the principle, there can only be one. The pointer can not be copied but moved.

`std::shared_ptr` contains a reference count and releases the memory if the last user leaves.

`std::weak_ptr` helps against *dangling pointers*.

`std::unique_ptr`

- What do you think about this piece of code?

```cpp
class Apple {
  int mX{};

public:
  explicit Apple(int x)
  : mX{x}
  {
    printf("ctor %d\n", mX);
  }

  ~Apple() { printf("dtor: %d\n", mX); }

  void Print() const { printf("%d\n", mX); }
};

// A The ownership of f is unclear
void Fun(Apple* f)
{
  f->Print();
  delete f;
}

void Use()
{
  auto* f = new Apple{37};

  Fun(f);
}
```

uniquePtr0

std::unique_ptr

- A unique pointer can not be copied. You have to *move* it.
- It has minimal administrative overhead.
- There is a specialization for arrays.
  - Its delete function can be determined, but no std::make_unique will work.

```cpp
class Apple {
  int mX{};

public:
  explicit Apple(int x)
  : mX{x}
  {
    printf("ctor %d\n", mX);
  }

  ~Apple() { printf("dtor: %d\n", mX); }

  void Print() const { printf("%d\n", mX); }
};

void Fun(std::unique_ptr<Apple> f) { f->Print(); }

void Use()
{
  auto f = std::make_unique<Apple>(37);

  Fun(std::move(f));
}
```

uniquePtr1

std::unique_ptr

- A unique pointer can not be copied. You have to *move* it.
- It has minimal administrative overhead.
- There is a specialization for arrays.
  - Its delete function can be determined, but no std::make_unique will work.

	Function	Explanation
1	.get()	Access the resource as a pointer.
2	.swap(other)	Swaps the contents of two unique pointers.
3	.reset(other)	Replaces the resource with another and releases the old resource.
4	std::make_unique<T>(...)	Create a unique pointer (only from C++14).

https://AndreasFertig.com
post@AndreasFertig.com

Exercise

a) Familiarize yourself with `std::unique_ptr`. Create a `unique_ptr` with a class `Apple` receiving an **`int`**. Invoke a method of class `Apple` on `std::unique_ptr`. Use `reset()` to delete the `std::unique_ptr` or to assign a new value to it.
   - Solution: `solUniquePtr.cpp`

b) Look at `exUniquePtrWithCustomDeleter.cpp`, which uses a custom deleter. Figure out the size that the `unique_ptr` uses.
   - Solution: `solUniquePtrWithCustomDeleter.cpp`

c) Should you have found out in the step before that `unique_ptr` with a custom deleter requires more space, figure out a way to reduce the size back to the one of a pointer.
   - Solution: `solUniquePtrWithCustomDeleterEfficient.cpp`

d) Have a look at `exUniquePtrCosts.cpp`, which uses a raw-pointer application programming interface (API). Change it such that it uses `std::unique_ptr` for clearness and safety. After that, compare the generated code for the raw-pointer and the `std::unique_ptr` version. What do you observe? Source: [7]
   - Solution: `solUniquePtrCosts.cpp`

`std::shared_ptr`

- With `std::shared_ptr`, you can share a resource that is automatically released.
  - A `shared_ptr` internally stores a pointer to the resource and a reference counter.
  - The reference counter is thread-safe through an atomic operation.
  - This makes it slightly more complicated than the `std::unique_ptr`.
  - A `shared_ptr` should not be passed around bluntly but consciously.

```cpp
class Apple {
  int mX;

public:
  Apple(int x)
  : mX{x}
  {
    printf("ctor %d\n", mX);
  }

  ~Apple() { printf("dtor: %d\n", mX); }

  void Print() const { printf("%d\n", mX); }
};

// A The ownership of f is unclear
void Fun(Apple* f) { f->Print(); }

void Use()
{
  auto f = new Apple{37};

  Fun(f);

  f->Print();
  delete f;  // B Cleanup required
}
```

sharedPtr3

std::shared_ptr

- With std::shared_ptr, you can share a resource that is automatically released.
  - A shared_ptr internally stores a pointer to the resource and a reference counter.
  - The reference counter is thread-safe through an atomic operation.
  - This makes it slightly more complicated than the std::unique_ptr.
  - A shared_ptr should not be passed around bluntly but consciously.

```
class Apple {
  int mX;

public:
  Apple(int x)
  : mX{x}
  {
    printf("ctor %d\n", mX);
  }

  ~Apple() { printf("dtor: %d\n", mX); }

  void Print() const { printf("%d\n", mX); }
};

void Fun(std::shared_ptr<Apple> f)
{
  printf("count: %ld\n", f.use_count());
  f->Print();
}

void Use()
{
  auto f = std::make_shared<Apple>(37);
  printf("count: %ld\n", f.use_count());

  Fun(f);
  printf("count: %ld\n", f.use_count());

  f->Print();
}
```

sharedPtr4

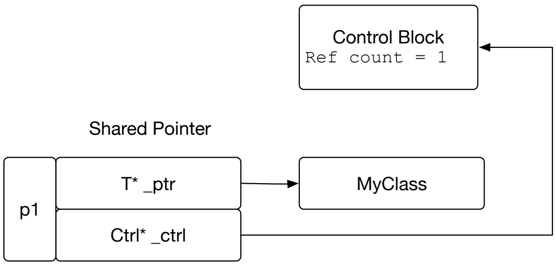
std::shared_ptr

- With std::shared_ptr, you can share a resource that is automatically released.

Function	Explanation
.get()	Access the resource as a pointer.
.swap(other)	Swaps the contents of two shared pointers.
.reset(other)	Replaces the resource with another and releases the old resource.
.use_count()	How many owners are currently available.
std::make_shared<T>(...)	Create a shared pointer safely.

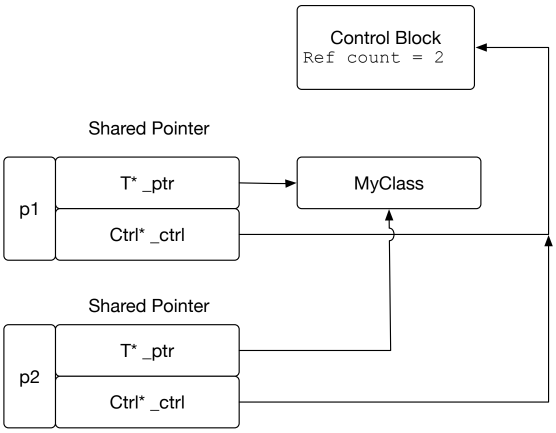
std::shared_ptr - Backstage

- std::shared_ptr is a great tool for memory management.
- But abstractions come with a cost.
  - A shared_ptr uses a control block to store the reference count.
  - This control block is also stored as a pointer in a shared_ptr.
  - Each increment / decrement is protected with an atomic lock!
  - The first shared_ptr for a raw-pointer needs to allocate this control block.



std::shared_ptr - Backstage

- std::shared_ptr is a great tool for memory management.
- But abstractions come with a cost.
  - A shared_ptr uses a control block to store the reference count.
  - This control block is also stored as a pointer in a shared_ptr.
  - Each increment / decrement is protected with an atomic lock!
  - The first shared_ptr for a raw-pointer needs to allocate this control block.
  - All following shared_ptr siblings can just use the preallocated control block.



https://AndreasFertig.com
post@AndreasFertig.com

std::weak_ptr

- With `std::weak_ptr`, cyclic references can be broken up.

```cpp
struct Person;

struct Team {
  // potentially and array/vector
  std::shared_ptr<Person> members{};

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

struct Person {
  std::shared_ptr<Team> team{};

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

void Fun()
{
  auto teamWoods = std::make_shared<Team>();
  auto alex      = std::make_shared<Person>();

  teamWoods->members = alex;
  alex->team         = teamWoods;
}

void Use()
{
  Fun();
  puts("finished");
}
```

weakPtro

std::weak_ptr

- With `std::weak_ptr`, cyclic references can be broken up.
  - Even a `std::shared_ptr` has its limits.
  - In the case of A → B → A, the `shared_ptr` can not release the resource.

```
$ ./a.out
finished
```

```cpp
struct Person;

struct Team {
  // potentially and array/vector
  std::shared_ptr<Person> members{};

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

struct Person {
  std::shared_ptr<Team> team{};

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

void Fun()
{
  auto teamWoods = std::make_shared<Team>();
  auto alex      = std::make_shared<Person>();

  teamWoods->members = alex;
  alex->team         = teamWoods;
}

void Use()
{
  Fun();
  puts("finished");
}
```

weakPtro

std::weak_ptr

- With std::weak_ptr, cyclic references can be broken up.
  - Even a std::shared_ptr has its limits.
  - In the case of A → B → A, the shared_ptr can not release the resource.
  - The std::weak_ptr helps here.

```
$ ./a.out
~Team
~Person
finished
```

```cpp
struct Person;

struct Team {
  // potentially and array/vector
  std::shared_ptr<Person> members{};

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

struct Person {
    std::weak_ptr<Team>  team{};

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

void Fun()
{
  auto teamWoods = std::make_shared<Team>();
  auto alex      = std::make_shared<Person>();

  teamWoods->members = alex;
  alex->team         = teamWoods;
}

void Use()
{
  Fun();
  puts("finished");
}
```

weakPtr1

std::weak_ptr

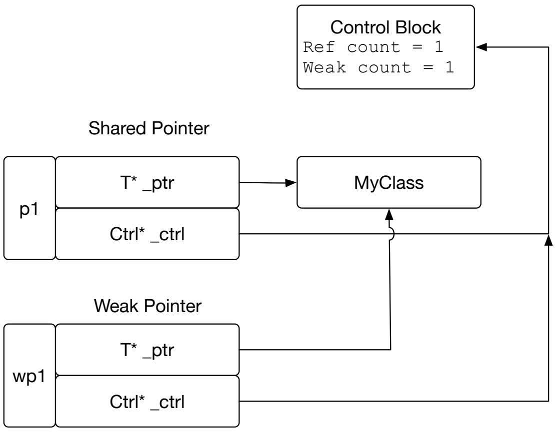
- With std::weak_ptr, cyclic references can be broken up.

Function	Explanation
.expired()	Check if the resource has already been released.
.swap(other)	Swaps the contents of two weak pointers.
.reset()	Releases the resource.
.use_count()	How many owners are currently available.
.lock()	Create a std::shared_ptr from the resource.

https://AndreasFertig.com
post@AndreasFertig.com

std::shared_ptr & std::weak_ptr - Backstage

- How does the former picture of std::shared_ptr change when we also have a std::weak_ptr?
  - A std::weak_ptr is always constructed from a std::shared_ptr.
  - The availability of std::weak_ptr increases the control block.
  - In addition to the ref count of shared pointers, another field counts the weak pointers.



Exercise

a) Use exSharedPtrSize.cpp , create shared_ptr for **int** and TenInts by using **new** and std::make_shared. Use mydelete to create a shared_ptr with a custom deleter. Observe the number of allocations and the allocated size for each combination. What is your conclusion?
   - Solution: solSharedPtrSize.cpp

https://AndreasFertig.com
post@AndreasFertig.com

Things to remember

- Take advantage of C++11 `std::unique_ptr` instead of raw pointers.
- Prefer `std::make_unique` and `std::make_shared` instead of direct, smart pointer initialization.
- To reduce the cost of a `std::unique_ptr` with a custom deleter, prefer passing a function object or lambda instead of a function pointer as a deleter.
- `std::unique_ptr` gives us exception guarantees. We can control them by marking the receiving functions as **`noexcept`**.
- Prefer `unique_ptr` over `shared_ptr` if you're unsure.
- For efficiency reasons, prefer `std::make_shared` to create a `std::shared_ptr`. It saves allocations and allocated space.

7. Using the STL efficiently

https://AndreasFertig.com
post@AndreasFertig.com

General STL guidelines

- Prefer the container's member functions to algorithms with the same name.
- The `at` member function does a out-of-bounds check and throws in case the access is out-of-bounds.
- If you need a variable length container, prefer `std::vector` by default.
- If you need a fixed-size container, prefer `std::array`.

`std::vector`: `push_back` or `emplace_back`

- Using `emplace_back` is tempting.
  - It creates the element in-place, saving a potential copy for temporaries even with move operations.
  - For non-temporary variables, there is no difference to

```cpp
// A Avoid seeing the realloc's
std::vector<Lifeguard> v{};
v.reserve(5);

puts("- push_back");

// B Using push_back with a temporary object.
v.push_back(Lifeguard{3});

puts("- emplace_back");

// C Using emplace_back with a temporary object.
v.emplace_back(Lifeguard{3});

puts("- emplace_back");

// D Using emplace_back to create a new object.
v.emplace_back(3);

puts("-------");
```

pushBackVsEmplaceBack2

```cpp
struct Lifeguard {
  Lifeguard() { puts("Lifeguard()"); }
  ~Lifeguard() { puts("~Lifeguard()"); }

  Lifeguard(int) { puts("Lifeguard(int)"); }

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

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

pushBackVsEmplaceBack2

`std::vector`: `push_back` or `emplace_back`

- What do you think about this code?

```
v.push_back(2 << 4);
v.emplace_back(2 << 4);
```

pushBackVsEmplaceBack1

Pre-reserve a `std::vector` if you know the final size

- What happens here?

```
auto GetUserInput(const int numValues)
{
  std::vector<int> v{};

  for(int i = 0; i < numValues; ++i) {
    v.push_back(AskForUserInput());
  }

  return v;
}
```

reserveIfPossible0

https://AndreasFertig.com
post@AndreasFertig.com

Pre-reserve a `std::vector` if you know the final size

- What happens here?
  - The vector will allocate its internal memory based on a strategy. Often it allocates twice the current memory. Then all the existing elements must be copied or moved.

```cpp
auto GetUserInput(const int numValues)
{
  std::vector<int> v{};

  for(int i = 0; i < numValues; ++i) {
    v.push_back(AskForUserInput());
  }

  return v;
}
```

reserveIfPossible0

Andreas Fertig v1.1 | Modern C++: When Efficiency Matters | 118

Pre-reserve a `std::vector` if you know the final size

- What happens here?
  - The vector will allocate its internal memory based on a strategy. Often it allocates twice the current memory. Then all the existing elements must be copied or moved.
- To spare some dynamic allocations and copies of your data, use `reserve` for a `std::vector` if you know the final size.

```cpp
auto GetUserInput(const int numValues)
{
  std::vector<int> v{};
  v.reserve(numValues);

  for(int i = 0; i < numValues; ++i) {
    v.push_back(AskForUserInput());
  }

  return v;
}
```

reserveIfPossible1

Andreas Fertig v1.1 | Modern C++: When Efficiency Matters | 119

STL container rules of thumb

- Prefer sequential containers when accessing elements by position.
  - Use `std::vector` as default.
  - Use `std::array` if you know the size up-front or if you can specify a maximum size.
  - Avoid `std::list` especially if you need to access objects in random order [8] [9].
  - `std::vector` is great when your systems uses caching, `std::list` isn't.
  - If to operation is to add and remove elements at both ends use `std::deque`, or check whether `std::vector` is still better.
  - Use `std::list` if the elements which are modified in the middle of a container, or check whether `std::vector` is still better.
  - If you think a list is the proper data structure, see whether a `std::forward_list` is sufficient. It saves a little bit of memory.
- Prefer associative containers when you need to access elements by a key.

Exercise

a) Measure the performance of the program `exMemoryAccess.cpp` . Did you expect these numbers?

b) The ordered associative containers such as `std::map` do not have a cache-line aware layout. Implement a cache line aware fast variant of a `std::map` echoing the operations you find in `exFlatMap.cpp` .
   - Solution: `solFlatMap.cpp`

Things to remember

- Use push_back when you have an *existing* temporary object that you want to move into your std::vector. Or, more general, use push_back when you want to move an existing object into your std::vector.
- Use emplace_back when you create a *new* temporary object. Instead of creating that temporary object, pass the object's constructor arguments directly to emplace_back.
- Pre-reserve a std::vector if you know the final size.

8. Miscellaneous

https://AndreasFertig.com
post@AndreasFertig.com

Further Information

- Detect the standard of the compiler:
  - Predefined compiler macros, like `__cplusplus = 201703L`, can be found in the standard at: [cpp.predefined].
  - Alternative: [10]
- A list of C++ standards and related drafts:
  - C++-03: N1638 (a little after 03, but one that is for free)
  - C++-11: N3337
  - C++-14: N4296
  - C++-17: N4640
  - C++-20: N4860
- Code formatting helper:
  - clang-tidy [11]: Contains functionality like modernize.
  - clang-format [12]: Automatically convert the source code to a specific format. Helps with style guides.
- Conferences
  - Meeting C++, Germany, https://meetingcpp.com, https://www.youtube.com/user/MeetingCPP
  - CppCon, USA, https://cppcon.org, https://www.youtube.com/user/CppCon
  - emBO++, Germany, http://embo.io
  - ACCU, UK, https://conference.accu.org, https://www.youtube.com/channel/UCJhay24LTpO1s4bIZxuIqKw

Further Information

- ADC++, Germany, http://www.adcpp.de
- code::dive, Poland, https://codedive.pl, https://www.youtube.com/channel/UCU0Rt8VHO5-YNQXwIjkf-1g

- Pod-/Screencast
  - C++ Weekly, https://www.youtube.com/playlist?list=PLs3KjaCtOwSZ2tbuV1hx8Xz-rFZTan2J1
  - CppCast, http://cppcast.com
- Books
  - A Tour of C++ [13]
  - Embracing Modern C++ Safely [14]
  - Beautiful C++ [15]
  - Effective Modern C++ [16]
  - C++ Templates: The Complete Guide [17]
  - C++17 in Detail [18]
- Blogs
  - https://fluentcpp.com
  - https://akrzemi1.wordpress.com

https://AndreasFertig.com
post@AndreasFertig.com

Used Compilers & Typography

Used Compilers

- Compilers used to compile (most of) the examples.
  - GCC 14.1.0
  - Clang 18.1.0

Typography

- Main font:
  - Camingo Dos Pro by Jan Fromm (https://janfromm.de/)
- Code font:
  - CamingoCode by Jan Fromm licensed under Creative Commons CC BY-ND, Version 3.0 http://creativecommons.org/licenses/by-nd/3.0/

Acronyms

API application programming interface

CTAD class template argument deduction

IIFE immediately invoked function expression

ODR one definition rule

RAII **R**esource **A**cquisition **I**s **I**nitialization

STL Standard Template Library

TMP template metaprogramming

https://AndreasFertig.com
post@AndreasFertig.com

References

[1] Köppe T., "Working Draft, Standard for Programming Language C++", *N4892*, June 2021. wg21.link/std

[2] "C++ Standard Draft Papers". https://github.com/cplusplus/draft/tree/master/papers

[3] Fertig A., "Why you should use std::move only rarely". https://andreasfertig.blog/2022/02/why-you-should-use-stdmove-only-rarely/

[4] Meyers S., *Effective C++: 55 Specific Ways to Improve Your Programs and Designs*, 3rd ed., ser. Addison-Wesley professional computing series. Upper Saddle River, NJ [u.a.]: Addison-Wesley, 2010.

[5] Hinnant H., "Everything You Ever Wanted To Know About Move Semantics (and then some)", *ACCU*, Apr. 2014. https://accu.org/content/conf2014/Howard_Hinnant_Accu_2014.pdf

[6] Toit S. D., "Working Draft, Standard for Programming Language C++", *N3337*, Jan. 2012. http://wg21.link/n3337

[7] Carruth C., "There are no zero-cost abstractions", *CppCon*, 2019. https://youtu.be/rHIkrotSwcc?t=1050

[8] Stroustrup B., "Why you should avoid linked lists". https://youtu.be/YQs6IC-vgmo

[9] Carruth C., "Cppcon 2014: "efficiency with algorithms, performance with data structures"". https://www.youtube.com/watch?v=fHNmRkzxHWs

[10] Reese B. and Honermann T., "Pre-defined Compiler Macros". https://sourceforge.net/p/predef/wiki/Standards/

References

[11] "clang-tidy". http://clang.llvm.org/extra/clang-tidy/

[12] "clang-format". https://clang.llvm.org/docs/ClangFormat.html

[13] Stroustrup B., *A Tour of C++*, ser. C++ In-Depth Series. Pearson Education, 2018.

[14] Lakos J., Romeo V., Khlebnikov R. and Meredith A., *Embracing Modern C++ Safely*. Addison Wesley Professional, 2021.

[15] Davidson J. and Gregory K., *Beautiful C++: 30 Core Guidelines for Writing Clean, Safe, and Fast Code*. Addison Wesley Publishing Company Incorporated, 2021.

[16] Meyers S., *Effective Modern C++: 42 Specific Ways to Improve Your Use of C++11 and C++14*. O'Reilly Media, 2014.

[17] Vandevoorde D., Josuttis N. and Gregor D., *C++ Templates: The Complete Guide*. Addison-Wesley, 2017.

[18] Filipek B., *C++17 in Detail: Learn the Exciting Features of the New C++ Standard!* Amazon Digital Services LLC - KDP Print US, 2019.

Images:
77: Franziska Panter
81: Franziska Panter
130: Franziska Panter

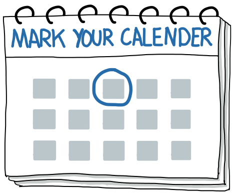
https://AndreasFertig.com
post@AndreasFertig.com

Upcoming Events

Talks

- *Fast and small C++ - When efficiency matters*, Meeting C++, November 16
- *Fast and small C++ - When efficiency matters*, code::dive, November 25
- *Effizientes C++ - Tips und Tricks aus dem Alltag*, ESE Kongress, December 04

For my upcoming talks you can check https://andreasfertig.com/talks/.
For my courses you can check https://andreasfertig.com/courses/.
Like to always be informed? Subscribe to my newsletter: https://andreasfertig.com/newsletter/.



About Andreas Fertig


Photo: Kristijan Matic www.kristijanmatic.de

Andreas Fertig, CEO of Unique Code GmbH, is an experienced trainer and consultant for C++ for standards 11 to 23.
Andreas is involved in the C++ standardization committee, developing the new standards. At international conferences, he presents how code can be written better. He publishes specialist articles, e.g., for iX magazine, and has published several textbooks on C++.
With C++ Insights (https://cppinsights.io), Andreas has created an internationally recognized tool that enables users to look behind the scenes of C++ and thus understand constructs even better.
Before training and consulting, he worked for Philips Medizin Systeme GmbH for ten years as a C++ software developer and architect focusing on embedded systems.
You can find Andreas online at andreasfertig.com.

https://AndreasFertig.com
post@AndreasFertig.com