Agenda: Modern C++

Focusing

- C++11/14/17 Extensions

Copyright (C) 2016: [Creative Commons BY-SA]

- by: Dipl.-Ing. Martin Weitzel
- for: MicroConsult GmbH Munich

- Example: modern_cplusplus/demo.cpp
- Example: preprocessor_macro/demo.cpp
- Example: pxt.h
- Example: show_type/demo.cpp
- Example: decltype/demo.cpp
- Example: const_usage/demo.cpp
- Example: const_object/demo.cpp
- Example: constexpr/demo.cpp
- Example: constexpr_ctor/demo.cpp
- Example: raw_string_literal/demo.cpp
- Example: enumeration/demo.cpp
- Example: value_semantics/demo.cpp
- Example: reference_basics/demo.cpp
- Example: copy_move_basics/demo.cpp
- Example: copy_move_defaults/demo.cpp
- Example: move_only_types/demo.cpp
- Example: perfect_forwarding/demo.cpp
- Example: auto_variables/demo.cpp
- Example: auto_typed_functions/demo.cpp
- Example: init_syntax/demo.cpp
- Example: initializer_lists/demo.cpp
- Example: range_for/demo.cpp

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

- Example: callable_lambda/demo.cpp
- Example: callable_operation/demo.cpp
- Example: member_init/demo.cpp
- Example: override_final/demo.cpp
- Example: template_variable/demo.cpp
- Example: unique_pointer/demo.cpp
- Example: shared_pointer/demo.cpp
- Example: shared_from_this/demo.cpp
- Example: weak_pointer/demo.cpp
- Example: chrono_library/demo.cpp
- Example: chrono_duration/demo.cpp
- Example: chrono_clocks/demo.cpp
- Example: chrono_timepoints/demo.cpp
- Example: regular_expressions/demo.cpp
- Example: random_numbers/demo.cpp
- Example: tuple_demo.cpp
- Example: string_related/demo.cpp
- Example: noexcept_usage/demo.cpp

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Notice to the Reader

- This document supplies the **Guiding Thread** only
  - It is not meant to be read stand alone
  - The bulk of teaching material is in the live demos
    - created and augmented throughout this course, while
    - **YOU** (the listeners) control for each topic
      - the depth of coverage and
      - the time spent on it
- In the *Electronic Version* feel free to follow the reference links
- The *Printed Version* is provided for annotations in hand-writing

---

If you nevertheless want to use this document for self-study, not only to follow the links to the compilable code, **also vary and extend it**.

If you need suggestions what to try (add or modify) you will find some in this presentation and a lot more usually in the example code itself.

There are two basic forms:

- **Conditional code `#if … #else … #endif`**
  - Usually both versions work, demonstrating alternative ways to solve the particular problem at hand
  - Sometimes not all versions are compatible with C++11 but may require C++14 or even C++14 features

- **Commented-out lines – sometimes with alternatives**
  - Sometimes these are shown because they **do not compile** and should indicate that a particular solution that may even seem attractive at first glance is **not** the way to go.
  - Furthermore, if removing the comment often some other line (in close proximity, typically the previous or next one) needs to be commented.

Understanding the code by trying variations is the crucial step to actually internalise the topics covered!

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Ways to Learn "Modern C++"

- Assuming you are well versed in "Classic C++98"
  - Get the idea from an example
  - Look-up the reference documentation
    - Grasp what you (seem to) understand at from a cursory look
    - **Test your understanding by tweaking the example!**
    - Maybe go back to documentation …
      - … but do not loop endlessly

Most things in C++ are designed to work exactly as you would expect it …

… except those that work differently.

Linked Reference Documentation

- This presentation frequently links to http://en.cppreference.com
  - It is provided freely
  - It is "vitally alive" and very recent
  - It clearly marks additions of C++11, C++14, and C++17
  - It provides compilable examples
- A close competitor also to recommend is http://www.cplusplus.com

---

What if `cppreference.com` goes out of service some day, temporarily or permanently?

Well, then the links in this document will not work anymore.

But: There exist mirrors "out there" in the world-wide-web with static copies and because this presentation is based on Remark it is plain (markdown) text you can load it into any decent editor and change the link targets systematically. (You can even make a static copy of the reference site yourself, only the live examples cannot be compiled then.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: modern_cplusplus/demo.cpp

```
#if 1
constexpr int faculty(int n) {
    return (n == 0) ? 1 : n*faculty(n-1);
}
#else
using ULL = unsigned long long;
std::function<ULL (ULL)> faculty =
    [](ULL n) -> ULL { return n ? n*faculty(n-1) : 1uLL; };
#endif

auto foo(bool tf) /* -> const int* */ {
    static constexpr int x{faculty(6)};
    return tf ? &x : nullptr;
    //if (tf) return &x; else return nullptr;
}
```

http://coliru.stacked-crooked.com/a/8257d868776840dd

- Alternatives in examples are provided to be tried!
  - Switch between `#if` – `#else` – `#endif` sections.
  - Remove comments (like around `/* -> const int* */`.
  - Move comments (like between last two lines in `foo`).
- Add annotations and feel free to extend.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): modern_cplusplus/demo.cpp

```
#define PX(x)\
    do { using namespace std;\
         cout << #x " --> " << (x) << endl;\
    } while (false)

#include <boost/type_index.hpp>

#define PT(t)\
    do { using namespace std; namespace bm = boost::typeindex;\
         cout << #t " --> " << bm::type_id_with_cvr<t>() << endl;\
    } while (false)

int main() {
    PX(faculty(6));
    PT(decltype(foo(true)));
}
```

- Useful helpers for writing test `main` programs:
  - Simple expression printer macro (`PT`)
  - Boost.Type_index based type printer macro (`PT`)

Live Examples (Using Coliru)

- Coliru is provided freely - do not overly stress it!
  - It is an Online Compile Service
  - Very up-to-date with respect to modern C++
  - Includes recent version of Boost Libraries
- **Coliru is provided freely - do not overly stress it!**

Consider to copy the code from the (coliru) example and try it on a locally installed compiler.

Advantage: you may not be trapped by "Modern C++" features not (yet) supported by your production use compiler.

Turn to Coliru (or another online compile service) only in case of problems with your local compiler.

If you come to the conclusion Coliru (or some other free internet service) is a **real big boon**, consider making a donation.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Section Overview

- Some Useful Stuff
- `const` vs. `constexpr`

Some Useful Stuff

- Why - **not** - Use The Preprocessor
- Expressions have **Value** and **Type**
- Printing Types at Run-Time
- C++98 and C++11 Support (for the former)
- The Preprocessor (once more)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

The Preprocessor

- Basic Features
  - Including Files
  - Expanding Macros
  - Conditional Compilation

http://en.cppreference.com/w/c/preprocessor

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Preprocessor: `#include` Principles

- Insert contents of a file
  - Usually looked up in search path
  - May actually skip content (Guard-Technique!)
- File existence can be tested in C++1z
  - `__has_include()`

http://en.cppreference.com/w/cpp/preprocessor/include

Preprocessor: `#include` Problems

- `#include` processing may be time consuming …
    - … and even cause subtle problems …
    - … especially when unnecessarily duplicated
- Counter measures are:
    - External to Compiler: Include guards
    - Internal in Compiler: Processing optimisations

Optimisations and other techniques to improve the "user experience" are in general not covered by the C++ standardisation but instead considered a *Quality of Implementation* issue (QOI).

Preprocessor: `#define` Principles

- Allows textual substitutions
  - At early translation phase
  - Little regard to C++ syntax
- Caller may provide arguments

http://en.cppreference.com/w/cpp/preprocessor/replace

Example: preprocessor_macro/demo.cpp

```cpp
#include <cmath>
#include <iostream>

#define PI 3.1415
#define sq(x) x*x

int main() {
    double r = 3.0, h = 4.0;
    std::cout << r * std::sqrt(sq(r) + sq(h)) * PI << std::endl;
}
```

- Describe what this code does.
- Point-out potential problems.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Preprocessor: `#define` Problems

- `#define`-s are
  - not properly scoped
  - may cause code duplication
- `#define`-s expand arguments textually
  - Countermeasure: extensive parenthesizing
  - **Strictly to avoid: side effects**
- Sloppy use may cause glitches hard to find

Example (cont.): preprocessor_macro/demo.cpp

```cpp
#include <cmath>
#include <iostream>

#define PI 3.1415
#define sq(x) x*x

namespace my {
    inline double sq(double x) { return x*x; }
}

int main() {
    using namespace my;
    std::cout << sq(PI) << std::endl;
    // -- more will be demonstrated live
}
```

- Demonstrate current problems of **macro** version of `sq`
- Add potential improvements
- Point out limits of solution

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Preprocessor: `#if` (etc.) Principles

- Allows to enable or disable parts of a source code
  - Conditions limited to:
    - Defined preprocessor macro
    - Literal constant expressions
  - May be nested but not across files

http://en.cppreference.com/w/cpp/preprocessor/conditional

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Preprocessor: `#if` (etc.) Problems

- Conditions evaluated early in translation, e.g.
  - no knowledge about types
  - conditions need to be trivial
- If used in nested forms
  - Code quickly gets unmaintainable
  - More so when crossing runtime control structure

```
    if (x > 0) { // x is positive
        …
#ifdef HANDLE_BELOW_ZERO
    } else if (x < 0) { // x is negative
        …
#endif
    } else { // x is zero or negative
        …
    }
```

Preprocessor: `#error` Principles and Problems

- Terminates translation with error message
  - Obviously needs to be combined with `#if` (etc.)
- Conditions evaluated early in translation, e.g.
  - no knowledge about types
  - conditions need to be trivial

http://en.cppreference.com/w/cpp/preprocessor/error

Consider replacing with `static_assert`

http://en.cppreference.com/w/cpp/language/static_assert

Optional Example: static_assert/demo.cpp

```
#if __cplusplus < 201402L
#error "This code makes use of C++14 features"
#endif

namespace my {
    class first {};
    class other : first {};
}

#include <type_traits>

int main() {
    static_assert(std::is_convertible<my::other, my::first>{}(),
                  "my::other must be convertible in my::first");
}
```

http://coliru.stacked-crooked.com/a/cf6cfccb737c9c1b

The above makes use of language features not covered so far and is meant for demonstration purposes only. (Expert question: Which C++14 feature is actually used and how could it have been avoided?)

Preprocessor: Classic Uses

- Including Files
- Include Guards

- Alternative `#pragma once`:
  - Widely Supported
  - Not Standardized

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: pxt.h

```
// file: pxt.h
#ifndef PXT_H
#define PXT_H

#include <iostream>

#define PX(expr)\
    ((void)(std::cout << (expr) << std::endl))

#endif
```

```
// file: preprocessor_macro/demo.cpp
#include "../pxt.h"
int main() {
    PX(6*7);
}
```

http://coliru.stacked-crooked.com/a/3fc20b0a0765e6fb

- Improve macro to also show the expression textually.
- Also include file name and line number where called.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Preprocessor: Deprecated Uses

- Macros to centralize literals
  - Prefer `const` / `constexpr`
- Macros to avoid subroutines
  - Prefer `inline`
- Macros to parametrize types
  - Prefer `template`-s
- Complex conditional sections
  - Rely on "optimised-out" Run-Time `if`
  - Consider Template Meta-Programming

Example (cont.): preprocessor_macro/demo.cpp

```
…
namespace my {
    const double PI = 2*std::acos(0.0);
    inline double sq(double x) { return x*x; }
}
…
int main() {
    using namespace my;
    PX(1/PI);
    …
    int z = 9;
    PX(++z);
    …
}
```

http://coliru.stacked-crooked.com/a/eadcb46a3f2391fa

- Come up with more test cases.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): preprocessor_macro/demo.cpp

```
…
namespace my {
    …
    const/*expr*/ double PI = 2*std::acos(0.0);
    #if 0
    inline constexpr double sq(double x) { return x*x; }
    #else
    template<typename T>
    inline constexpr T sq(T x) { return x*x; }
    #endif
    …
}
…
```

http://coliru.stacked-crooked.com/a/006fb9bec1154bf8

Covered later:

- Difference between `const` and `constexpr`.
- Fully type-generic version of `sq`.

Preprocessor: Macros Still Useful

- Adorned use of macro arguments
  - Stringizing (`#arg`)
  - Token Pasting (`arg ## whatever`)

Example: pxt.h

```
#define PX(expr)\
    ((void)(std::cout << __FILE__  << ':' << __LINE__\
                      << "\t" #expr " --> "\
                      << (expr) << std::endl))
```

http://coliru.stacked-crooked.com/a/429608d619fa9559

Some more examples are shown along this chapter.

Types vs. Values

- Each expression has distinct
  - **Type** – known at Compile-Time
  - **Value** – known at Runtime

Show Expression Value

- At Run-Time `operator>>` works for
  - **All** basic types
  - **Some** more complex types
    - e.g. pointers (shows address in hex)
    - **not** for native arrays
- Can be overloaded for user-defined classes
- Values often **not** known at Compile-Time

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: show_value/demo.cpp

- Overload output operator for class `my::point`.
- Advanced: overload output operator for native array.

Show (Expression) Type

- At Run-Time
  - … ?
- At Compile-Time
  - May occurs in error messages
- Types are **always** known at Compile-Time

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: show_type/demo.cpp

```
#define PX(value)\
    std::cout << value << std::endl
#define PT(type)\
    std::cout << ?…? type ?…? << std::endl

int main() {
    int v = 6*7;
    PX(v);    // --> 42
    PT(int);  // --> int
}
```

- How could that be achieved?

Type Aliases

- Alternative syntax for `typedef`
  - After keyword `using` alias name comes first …
  - … then an equals (=), then the aliased type
- Improves readability
  - left to right, as with variables
  - otherwise same semantics as `typedef`
- In addition: may be templated

http://en.cppreference.com/w/cpp/language/type_alias

Example (cont.): show_type/demo.cpp

```
int main() {
    using iptr = int*;
    typedef double *dptr;
    PT(iptr); // --> int*
    PT(dptr); // --> double*
}
```

`typeid()` and `decltype()`

- C++98 introduced `typeid()`
  - Has `name()` member (among other uses)
    - Returns unique string for each distinct type
    - Name of type may be encoded (not C++ syntax)

http://en.cppreference.com/w/cpp/language/typeid

- C++11 added `decltype()`
  - Takes type for expression
  - and then ???

http://en.cppreference.com/w/cpp/language/decltype

Example (cont.): show_type/demo.cpp

```
#include <iostream>
#include <typeinfo>

#define PT(type)\
    std::cout << typeid(type).name() << std::endl

int main() {
   using iptr = int*;
   iptr p = nullptr;
   PT(int);
   PT(iptr);
   PT(decltype(p));
   PT(decltype(*p));
   PT(decltype(nullptr));
}
```

http://coliru.stacked-crooked.com/a/f444d6912e498e68

- Can you guess the name mangling?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Show Type in C++ Syntax

- Maybe your Compiler does (lucky you!) …
- … if not:
  - Read in error message
    - idea "stolen" from Scott Meyers
    - ugly … but works (for desperadoes!)
  - Turn error message into string
    - starts out simple …
    - … needs much diligence to complete

Example (cont.): show_type/demo.cpp

```
template<typename T> struct showtype;

template<> struct showtype<int> {
    static const char *str() { return "int"; }
};
template<> struct showtype<int*> {
    static const char *str() { return "int*"; }
};

#define PT(type) std::cout << #type << " --> "\
                           << showtype<type>::str() << std::endl;

int main() {
   int* p = nullptr;
   PT(int);
   PT(decltype(p));
// PT(decltype(*p));
}
```

http://coliru.stacked-crooked.com/a/62bc674eba5a575c

- Why is `decltype(p)` an error?
- What to add so that it can be printed?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Build Your Own Type Printer

- The Preprocessor is your friend
- Or use Boost.Type_index

Example (cont.): show_type/demo.cpp

```
…
template<> struct showtype<void>
{static std::string str() {return "void";}};
template<> struct showtype<bool>
{static std::string str() {return "bool";}};
…
template<> struct showtype<unsigned long long>
{static std::string str() {return "unsigned long long";}};
…
template<typename T> struct showtype<const T>
{static std::string str() {return "const " + showtype<T>::str();}};
…
template<typename T> struct showtype<T*>
{static std::string str() {return showtype<T>::str() + "*";}};
template<typename T> struct showtype<T&>
{static std::string str() {return showtype<T>::str() + "&";}};
…
```

http://coliru.stacked-crooked.com/a/4907e3cbb1ed8c81

- Be sure to understand the systematic approach.
- How could the preprocessor help?

`decltype()` Peculiarities

- Use for variable names
- Use for expressions
  - Parenthesizing?
  - Non-Modifying Operations

Example: decltype/demo.cpp

```
#include <iostream>
#include <boost/type_index.hpp>

#define PT(type)\
    std::cout << #type << " --> "\
              << boost::typeindex::type_id_with_cvr<type>()
              << std::endl;

int main() {
                            PT(decltype(42))
    const int ci = 42;  PT(decltype(ci))
    int i = ci;         PT(decltype(i))
                            PT(decltype(2*i))
    int &ri = i;        PT(decltype(ri))
                            PT(decltype(2*ri))
    const int *p = &i;  PT(decltype(p))
                            PT(decltype(*p))
                            PT(decltype(*p + 0))
}
```

http://coliru.stacked-crooked.com/a/549408ad7b6130c9

- Come up with more examples of your own.
- Especially try to explore grey areas.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

The Preprocessor (again)

- C++11: Variadic Macro Arguments
  - Defining with `...`
  - Accessing with `__VA_ARGS__`
- Untypical use: solve the comma problem

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: pxt.h

```
#include <iostream>
#include <boost/type_index.hpp>

#define PT(...)\
    std::cout << #__VA_ARGS__ << " --> "\
              << boost::typeindex::type_id_with_cvr<__VA_ARGS__>()\
              << std::endl;

#include <string>
#include <map>

int main() {
    PT(std::string)
    PT(std::map<std::string, int>)
}
```

http://coliru.stacked-crooked.com/a/9a58d5a76693422e

- Come up with more examples to test the type printer.
- How does it handle classes defined by the program itself?
- How does it compare to `typeid( … ).name()` in this respect?

`const` vs. `constexpr`

- `const` has always there
  - Actually introduced by C++
  - Adopted by C89 (ANSI/ISO-C)
- C++11 added `constexpr`
  - For data (Compile-Time initialisation)
  - For functions (Compile-Time callable)
- C++14 improved `constexpr` functions

`const` is for write protection

- No code generated on behalf of `const`
- Modifying operations turned into Compile Errors
- May or may not hold
  - different values
  - on different instantiations
- May or may not
  - load initial value with executable
  - contrary to requiring Run-Time initialisation

http://en.cppreference.com/w/cpp/language/cv

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`const` Qualified Variables

- Guaranteed to be initialised
  - exactly once (at the start of their life-time)
  - never change (until the end of their life-time)

Effect of `const_cast` is implementation defined!

http://en.cppreference.com/w/cpp/language/const_cast

Example: const_usage/demo.cpp

```
#include <iostream>

#define PX(...)\
    std::cout << __FUNCTION__ << ':' << __LINE__ << '\t'\
              << #__VA_ARGS__ " --> "\
              << (__VA_ARGS__) << std::endl;

int main() {
    int i = 10;                 PX(i)
    ++i;                        PX(i)
    const int ci = 20;          PX(ci)
//  ++ci;                       PX(ci)
    int *ip = &i;               PX(++*ip)
                                PX(i)
    const int& cri = i;         PX(cri)
    ++i;                        PX(cri)
}
```

http://coliru.stacked-crooked.com/a/b6a7a847e3fdab1e

- Add more examples demonstrating effects of `const`.
- Especially understand effects of `const` on pointers.

Optional Example: const_cast/demo.cpp

```
#include <iostream>

#define PX(...)\
    std::cout << __FUNCTION__ << ':' << __LINE__ << '\t'\
              << #__VA_ARGS__ " --> "\
              << (__VA_ARGS__) << std::endl;

const int v = 101;

int main() {
                                PX(::v)
//                              PX(++::v)
    const int v = 4711;         PX(v)
    ++const_cast<int&>(v);      PX(v)
    ++*const_cast<int*>(&v);    PX(v)
    const int* ip = &v;         PX(*ip)
    ++*const_cast<int*>(ip);    PX(*ip)
}
```

http://coliru.stacked-crooked.com/a/f8f7f0933d68b758

- Add more examples demonstrating effects of `const_cast`.

`const` Qualified Objects

- Require use of `const` member functions
  - `const` goes after argument list
  - before implementation code block or semicolon
- Such member functions
  - may not modify data members (Compile Error)
  - except when marked as `mutable`

Typical use for caching of values expensive to calculate.

Example: const_object/demo.cpp

```
class point {
    double xc, yc;
public:
    point(double xc_, double yc_) : xc(xc_), yc(yc_) {}
    double x() const { return xc; }
    double y() const { return yc; }
    void x(double xc_) { xc = xc_; }
    void y(double yc_) { yc = yc_; }
    point &shift_x(double xd) { xc += xd; return *this; }
    point &shift_y(double yd) { yc += yd; return *this; }
};
std::ostream& operator<<(std::ostream& os, const point& pt) {
    return os << "point{" << pt.x() << ", " << pt.y() << "}";
}
void shiftxy(point &pt, double xd, double yd) {
    pt.shift_x(xd).shift_y(yd);
}
```

http://coliru.stacked-crooked.com/a/4dcb87353c1e1a84

- Demonstrate operations with modifiable `point` object.
- Demonstrate restriction for non-modifiable `point` objects.

Optional Example (cont.): const_object/demo.cpp

```
class point {
    double xc, yc;
    mutable double radius, angle;
    mutable bool mod;
    void sync_ra() const {
        radius = std::sqrt(xc*xc + yc*yc);
        angle = std::atan2(xc, yc);
        mod = false;
    }
public:
    point(double xc_, double yc_) : xc(xc_), yc(yc_), mod(true) {}
    …
    double r() const { if (mod) sync_ra(); return radius; }
    double a() const { if (mod) sync_ra(); return angle; }
    void x(double xc_) { xc = xc_; mod = true; }
    void y(double yc_) { yc = yc_; mod = true; }
    void shift_x(double xd) { xc += xd; mod = true; }
    void shift_y(double yd) { yc += yd; mod = true; }
}
```

http://coliru.stacked-crooked.com/a/4dcb87353c1e1a84

- Demonstrate how `const_cast` can replace `mutable`.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`constexpr` Enforces Compile-Time

- `constexpr` for data
  - guarantees initialisation during compilation
  - turns modification attempts into Compile Error
- `constexpr` for functions
  - `constexpr` goes all on the left
  - requires visible implementation (like `inline`)

http://en.cppreference.com/w/cpp/language/constexpr

---

http://www.informit.com/guides/content.aspx?g=cplusplus&seqNum=315

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: constexpr/demo.cpp

```cpp
// `constexpr` function to find greatest common divisor
//
constexpr unsigned gcd(unsigned m, unsigned n) {
#if 0
    return n ? gcd(n, m % n) : m; // as introduced in C++11
#else
    if (n == 0)             // restrictions lifted with C++14
        return m;
    else
        return gcd(n, m % n);
#endif
}

int main() {
   const int x = 27;                PX(x)
   constexpr auto y = gcd(12, 6);   PX(y)
   constexpr auto z = gcd(x, y);    PX(z)
}
```

http://coliru.stacked-crooked.com/a/bc181edb47b6e6f9

- Try more variations of `const` and `constexpr`

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`constexpr` Data Items

- Must be initialised
  - Literal constants
  - other `constexpr` data values
  - calls to `constexpr` functions
  - or a combination thereof.
- Initial value (can be) loaded from executable file

Example (cont.): constexpr/demo.cpp

```
#include <cmath>

const/*expr*/ double PI = 2*std::acos(0.0);

// constexpr
double circle_area(double r) { return PI*r*r; }

// constexpr
double sphere_surface(double r) { return 4*PI*r*r; }

int main() {
    double radius = 3.0; PX(radius)
                         PX(circle_area(radius))
                         PX(sphere_surface(radius))
}
```

http://coliru.stacked-crooked.com/a/1192d833caacb675

- Demonstrate Compile-Time and Run-Time initialisation.
- In the C++ standard `<cmath>` functions are **not** `constexpr`.
- GCC extends the standard by making `<cmath>` functions `constexpr`.

`constexpr` Functions

- Are callable at Compile-Time …
  - … given that all arguments can be determined
- A Run-Time version may be generated for calls …
  - … with arguments **not** `constexpr` themselves

Restrictions on `constexpr` functions relaxed in C++14.

https://isocpp.org/files/papers/N3652.html

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`const` vs. `constexpr`

- Some usages of plain `const`
  - also qualify as compile-time constants
  - hence may be used to
    - initialize `constexpr` data
    - call `constexpr` functions

http://en.cppreference.com/w/cpp/language/constant_expression

---

To avoid confusion, with C++11/14 rather prefer `constexpr` over `const`.

`constexpr` Constructors

- Constructors may also be `constexpr`
  - given the class is "simple enough" (POD) …
  - … objects may be created at compile time and
- Such objects may participate in (other) …
  - `constexpr` data initializations
  - `constexpr` function calls

Example: constexpr_ctor/demo.cpp

```
struct point {
    const double x, y;
    constexpr point(double x_, double y_) : x(x_), y(y_) {}
};
constexpr double square(double v) {
    return v*v;
}
/*constexpr*/ double operator-(const point lhs, const point rhs) {
    return std::sqrt(square(lhs.x-rhs.x) + square(lhs.y-rhs.y));
}
std::ostream& operator<<(std::ostream& lhs, const point& rhs) {
    return lhs << "point{" << rhs.x << ", " << rhs.y << "}";
}

int main() {
    constexpr point a{0, 3};         PX(a)
    constexpr point b{4, 0};         PX(b)
    const/*expr*/ double dist{a-b}; PX(dist)
}
```

http://coliru.stacked-crooked.com/a/babf9e34714a33fb

- Discuss use and applicability of `const` and `constexpr` above.
- Protect member data (`x` and `x`) and provide accessors.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

New Kinds of Literals

- Binary Literal Notation
- Structuring Numeric Literals (C++14)
- Character / String Literal Encoding
- User Defined Literals

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Binary Literal Notation (C++14)

- Prefix `0b` followed by `0` and `1`
  - `0b11` (same as `03`, `3`, or `0x3`)
  - `0b1001` (same as `011`, `9`, or `0x9`)
  - `0b1110` (same as `016`, `14`, or `0xE`)
  - `0b11011110101011011011111011101111` (same as ?)

http://en.cppreference.com/w/cpp/language/integer_literal

Structuring Numeric Literals (C++14)

- Use Apostroph (`'` = ASCII/UTF code point 39)
  - `1'000'000'000`
  - `3'735'928'559`
  - `0xDE'AD'BE'EF`
  - `0b1101'1110'1010'1101'1011'1110'1110'1111`
  - `0b110'111101'01011'01'1011111'01'110'1111`

http://en.cppreference.com/w/cpp/language/integer_literal#Syntax

(see remark in box at end of section)

Deep syntax change (downto lexer)

Effects IDEs and Syntax-Highlighters!

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: numeric_literals/demo.cpp

```
int main() {
    PX(0b11)    PX(03)  PX(3)   PX(0x3)
    PX(0b1001)  PX(011) PX(9)   PX(0x9)
    PX(0b1110)  PX(016) PX(14)  PX(0xE)

    PX(0b11011110101011011011111011101111)
    PX(3'735'928'559)
    PX(0xDE'AD'BE'EF)
    PX(0b1101'1110'1010'1101'1011'1110'1110'1111)
    PX(0b110'111101'01011'01'1011111'01'110'1111)
}
```

http://coliru.stacked-crooked.com/a/896dd0ad3580020b

Note: `BOOST_BINARY` uses the preprocessor to provide binary constants with optional structuring by spaces.

http://www.boost.org/doc/libs/release/libs/utility/utility.htm#BOOST_BINARY

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Character / String Literal Encoding (C++11)

- What makes up a Character?
  - User perceived character *not same as*
  - Glyph *not same as*
  - Code-Point *not same as*
  - Code-Unit
- Understand the Terminology!
  - Encoding not same as
  - Character Set

Very Quick Tour into Terminology

- Glyphs may or may not represent
  - exactly **one** user perceived character (like y)
  - a part of a user perceived character (e.g. diacritic marks)
  - more then one user perceived characters (e.g. ligatures)
- Character Sets
  - Define available glyphs
  - Assign code points (or otherwise name) glyphs
- Encodings
  - Map code points to code units

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Traditional Code Unit Sizes

- 7 Bit = 128 possible values
  - traditional ASCII (excess bit sometimes used for parity)
- 8 Bit = 256 possible values
  - smallest addressable unit in most of today's computers
  - typically chosen for `char` in C/C++
- 16 Bit = 65536 possible values
  - internal character type in Windows / `char` type in Java
  - may be chosen for `wchar_t` in C/C++
- 32 Bit = 4'294'967'296 possible values
  - may also be chosen for `wchar_t` in C/C++

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Traditional Character Sets

- 7-bit ASCII (128 code points)
  - Code Point = Code Unit = User Perceived Character
  - Only 95 printable character
  - Glyphs map 1:1 to user perceived characters
- ISO 8859-x (256 code points)
  - Requires variants to hold all world alphabets
- Unicode (about one million code points)
  - 1:1 mapping to 32 bit (and still some excess bits)
  - 1:1 mapping to 16 bit **only** when limited to BMP

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Traditional Encodings

- In computer "stone age" often 1:1 mapping between
  - Code units and
  - Code points and
  - User perceived character
- Code Pages (e.g. in IBM main-frames and PC-DOS/MS-DOS)
  - usually 1:1 mapping to glyphs or user perceived characters
- Today frequently used encodings
  - UTF-8 = 1 to 4 code units per code point)
  - UTF-16 (sic!) = 1 or 2 code units per code point
  - UTF-32 = 1 code unit per code point

Ligatures and Combining Characters

- Mostly **not** much of a problem when
  - when user input is just …
  - … "handed through" to system calls
- But may cause headaches for reliable string processing
  - because different possible encodings …
  - … for what is perceived by the user
- Normalization to canonical form may help …
  - … but very limited support, even in modern C++
  - maybe consider using the ICU library

Extended Character / String Literals

- C++11 added prefixes
  - to specify various encodings
  - to allow for raw strings
- Encoding-Transformations
  - do **not** generally happen automatically
  - but may be bound to I/O streams

http://en.cppreference.com/w/cpp/locale/wstring_convert

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Extended Forms of Character Literals

- Like in C in C++98 `L` for a character literal
  - changes its type to `wchar_t`
  - which (typically) may be either 16 or 32 bit
- C++11 added the prefixes
  - `u8` – request use of UTF-8 encoding
  - `u` (lower case) – request use of UTF-16 encoding
    - Changes its type to `char16_t`
    - **Compile error if character outside of BMP**
  - `U` (upper case) – request use of UTF-32 encoding
    - Changes its type to `char32_t`

http://en.cppreference.com/w/cpp/language/character_literal

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Extended Forms of String Literals

- Like in C in C++98 `L` for a string literal
  - changes the type **of its elements** to `wchar_t`
- C++11 added the prefixes
  - `u8` – request use of UTF-8 encoding
  - `u` (lower case) – request use of UTF-16 encoding
    - Changes the type **of its elements** to `char16_t`
    - Uses two code units for characters outside BMP
  - `U` (upper case) – request use of UTF-32 encoding
    - Changes the type **of its elements** to `char32_t`

http://en.cppreference.com/w/cpp/language/string_literal

Optional Example: char_string_literal/demo.cpp

```
// assuming sufficient type-printing capability and
// definition of macros `PX` and `PT` as usual

int main() {
    PX(static_cast<long long>('ü'))
    PX(static_cast<long long>(L'ü'))
    PT(decltype('ü'))
    PT(decltype(u8'ü'))
    PT(decltype(u'ü'))
    PT(decltype(U'ü'))
    PT(decltype("ü"))   PT(decltype(*"ü"))
    PT(decltype(u8"ü")) PT(decltype(*u8"ü"))
    PT(decltype(u"ü"))  PT(decltype(*u"ü"))
    PT(decltype(U"ü"))  PT(decltype(*U"ü"))
}
```

http://coliru.stacked-crooked.com/a/c03827eb62cf51cb

- Why do some statements not compile?
- Try more of the above.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Raw String Literals

- Added by C++11
  - Requested by prefix `R`
  - Follows optional encoding prefix
- Every character represents itself
  - Especially backslashes ('\') are **not** evaluated
    - Line concatenation from *Translation Phase 2* is reverted
  - Termination delimiter may be chosen

http://en.cppreference.com/w/cpp/language/translation_phases#Phase_2

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: raw_string_literal/demo.cpp

```
#include <iostream>
int main() {
    std::cout << u8R"!end-of-html!(
    <!DOCTYPE html>
    <html lang="de">
        <head>
            <meta charset="utf-8" />
            <title>TBFE Homepage</title>
        </head>
        <body style="text-align: center;" >
            — Technische Beratung für EDV — <br />
            © 2016: Dipl.-Ing. Martin Weitzel, Roßdorf
        </body>
    </html>
    )!end-of-html!";
}
```

- What **exactly** is the output of the above?
- Show backslashes do **not** cause line continuation.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

User Defined Literal

- Extends (traditionally) hard-coded suffix use
  - like `…L`, `…LL`, `…LU` ...
- Syntax based on operator overloading (`operator""`)
  - Reserves regular suffixes for later use
  - User defined suffixes should
    - start with underscore
    - followed by lower case letter

http://en.cppreference.com/w/cpp/language/user_literal

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: literal_suffix/demo.cpp

```
namespace my {
    struct length {
        const double meter;
        length(double m) : meter(m) {}
    };
}

my::length operator"" _mm(long double parsed_value) {
    return my::length(parsed_value/1e3);
}
my::length operator"" _km(long double parsed_value) {
    return my::length(parsed_value*1e3);
}

int main() {
    my::length x{3.0_m};            PX(x.meter);
    my::length y{715._mm};          PX(y.meter);
//  my::length z = 3_km + 120_m;    PX(z.meter);
}
```

http://coliru.stacked-crooked.com/a/1e47b87f9498ff65

- Support more usage forms

Predefined Literal Suffixes

- C++14 has introduced predefined suffixes:
  - String literal to `std::basic_string` (`s`) …
  - … directly visible via `using namespace::string_literals`
  - Complex types imaginary part (`i`, `if`, `il`) …
  - … directly visible via `using namespace complex::literals`
  - Duration types (`h`, `s`, `min`, `ms`, `us`, `ns`) …
  - … directly visible via `using namespace std::chrono_literals`
- Future versions of standard C++ may add more

http://en.cppreference.com/w/cpp/language/user_literal#Standard_library

---

For full support of type-safe physical units consider *Boost.Units*:

http://www.boost.org/doc/libs/develop/doc/html/boost_units.html

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: predef_suffixes/demo.cpp

```
#include <chrono>

int main() {
    namespace sc = std::chrono;
    using namespace std::chrono_literals;
    const sc::seconds total = 12h + 7min + 15s;
    PX(total.count());
}
```

http://coliru.stacked-crooked.com/a/1209a09c4f1a1db2

The *Durations and Clocks* library (`std::chrono`) will be covered later.

Strongly Typed Enumerations

- Classic Enumerations
  - Originally much like in C
    - though no silent conversion **from** integral
  - extended in C++11
- Scoped Enumerations (C++11)

Classic Enumerations

- `enum`-labels pollute enclosing namespace
- **Silent** conversion into integral
- Only since C++11
  - Representation can be chosen
  - Forward declaration is possible

http://en.cppreference.com/w/cpp/language/enum#Unscoped_enumerations

Scoped Enumerations

- `enum`-labels require scoping prefix
- All conversions must be explicit
- Representation can be chosen
- Forward declaration is possible

http://en.cppreference.com/w/cpp/language/enum#Scoped_enumerations

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: enumeration/demo.cpp

```
// text adjustments Left, Center, Right
enum class Adjust /*: char */ {
    L = 'l', C = 'c', R = 'r',
};

// colors Red, Green, Blue
enum Color : unsigned char {
    R = (1<<0),
    G = (1<<1),
    B = (1<<2),
};

int main() {
    PX(static_cast<char>(Ajust::L)
    …
    PX(R|B)
}
```

http://coliru.stacked-crooked.com/a/9fcd9f0298efdf48

- Demonstrate different scoping.
- Demonstrate different conversion rules.
- Demonstrate base type selection.

Copying vs. Moving

- Value Assignment
- Classic references are aliases
- Rvalue references (C++11) bind to temporaries
- Move-constructor and -assignment
- Move-only types

Value Assignment

- Assignment usually means copying
    - Cheap and easy
        - when directly supported by machine instructions
        - maybe even for character strings (to some degree)
    - Potentially expensive
        - for large objects
        - especially when uniquely owned
- **In initialisation copy elision (RVO or NRVO) may kick in**

http://en.cppreference.com/w/cpp/language/copy_elision

---

http://stupefydeveloper.blogspot.de/2008/10/c-rvo-and-nrvo.html

Example: value_semantics/demo.cpp

```cpp
class point {
    double xc, yc;
public:
    point(double xc_ = 0.0, double yc_ = 0.0)
        : xc(xc_), yc(yc_)
    {}
    std::string to_string() const {
        return "point(xc=" + std::to_string(xc)
                 + ", yc=" + std::to_string(yc) + ")";
    }
};

point foo(double);

int main() {
    point pt1{3.0, 4.0};    PX(pt1.to_string());
    point pt2{pt1};         PX(pt2.to_string());
    point pt3{foo(1.0)};    PX(pt3.to_string());
}
```

http://coliru.stacked-crooked.com/a/72b939757e46d642

- Demonstrate implicit and explicit copying operations.
- Demonstrate RVO.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

References

- Initialising a reference
  - basically creates an alias
    - therefore the initialisation requires no copying
  - usually involves a pointer behind the scenes
    - therefore access via reference causes overhead
- A reference initialised from a reference
  - directly aliases the referenced object
  - hence no "double indirection" overhead

http://en.cppreference.com/w/cpp/language/reference

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: reference_basics/demo.cpp

```
int main() {
    int x = 0;                      PX(x)
    const int cx = 101;             PX(cx)
    int &xr = x;                    PX(xr)
                                    PX(++xr)
                                    PX(x)
    const int &cxr = cx;            PX(cxr)
//                                  PX(++cxr)
    int&& z = 2*x;                  PX(z)
                                    PX(++z)
                                    PX(x)
}
```

http://coliru.stacked-crooked.com/a/b46ba3d4b3e78bd6

- Demonstrate basic behaviour of references.
- Try more combinations (especially in grey araeas).

Classic References and `const`

- Compared to pointers a reference is `const`
  - as it always refers to the same object
  - but the aliased object may or may not be `const`
- `const`-qualified references
  - are mandatory to bind to `const`-qualified objects
  - also bind to modifiable memory locations
  - but **cannot** used to modify there content
- **Only `const`-qualified references bind to temporaries**

http://en.cppreference.com/w/cpp/language/reference#Lvalue_references

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Rvalue References

- Rvalue references (as introduced with C++11)
  - **only** bind to temporaries
  - are typically **not** `const`-qualified

http://en.cppreference.com/w/cpp/language/reference#Rvalue_references

---

The major purpose (99% case) of rvalue references is to supply class specific move operations in addition to or instead of copying.

Overloading Based On References

- Reference-based overloading (of functions) is possible as
    - `const` references *can co-exist with*
    - non-`const`-references *can co-exist with*
    - rvalue references (usually non-`const`)

Example (cont.): reference_basics/demo.cpp

```
void foo(int& ri)            { PX(++ri); }
void foo(const int &cri)     { PX(cri); }
void foo(int&& rri)          { PX(rri); }

int main() {
    int x = 0;
    const int cx = 101;
    foo(x);
    foo(cx);
    foo(2*x);
}
```

http://coliru.stacked-crooked.com/a/15009e8c6e05f61c

- Demonstrate reference based overloading.
- Try more combinations (especially in grey araeas).

Class Specific Operations

- Classes are copyable by default
  - if all of their members are copyable
  - such operations are not explicitly blocked
- Copying potentially takes place during
  - construction (copy constructor)
  - assignment (`operator=`)

http://en.cppreference.com/w/cpp/language/copy_constructor

http://en.cppreference.com/w/cpp/language/move_constructor

Example: copy_move_basics/demo.cpp

```cpp
class point {
    // ...
};
point operator+(const point&, const point&) { return {}; }

void foo(point&)        {}
void foo(const point&)  {}
void foo(point&&)       {}

int main() {
    point pt1;
    const point pt2{pt1};
    foo(pt1);
    foo(pt2);
    foo(pt1 + pt2);
}
```

http://coliru.stacked-crooked.com/a/926b040c36d2a19d

- Demonstrate Copy and Move Operations

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Move Constructor and Assignment

- Since C++11 also
  - move constructors can be added
  - move assignment can be added
- Selection is by overload resolution

http://en.cppreference.com/w/cpp/language/copy_constructor

http://en.cppreference.com/w/cpp/language/copy_assignment

Sometimes effects are hard to demonstrate since RVO or NRVO might take place (if possible use compiler option to disable).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Defaulting and Deleting

- Since C++11 also copy and move constructor
  - can be `default`-ed
  - or `delete`-d
- Same for copy and move assignment

Example: copy_move_defaults/demo.cpp

```
class point {
public:
    point(const point&)             =default;
    point(point&&)                  =default;
    point& operator=(const point&)  =default;
    point& operator=(point&&)       =default;
    point()                         =default;
};
point operator+(const point&, const point&) { return {}; }

void foo(point&)        {}
void foo(const point&)  {}
void foo(point&&)       {}

int main() {
    // ...
}
```

http://coliru.stacked-crooked.com/a/503cb1be438af585

- Demonstrate `default`-ed and `delete`-ed operations.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Move-Only Types

- Such only support move operations
  - which may be `default`-ed or
  - specifically implemented
- Typical library examples are
  - `std::unique_pointer`
  - threads, mutexes, and locks
  - and all kinds of I/O streams

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: move_only_types/demo.cpp

```
class point {
    const char *nm;
    double xc, yc;
public:
    point(const char *name, double xc_ = 0.0, double yc_ = 0.0)
        : nm(std::strcpy(new char[std::strlen(name)+1], name))
        , xc(xc_), yc(yc_)
    {}
    ~point() { delete[] nm; }
    point(point&& init) noexcept
        : nm(init.nm), xc(init.xc), yc(init.yc) {
        init.nm = nullptr;
    }
    point& operator=(point&& rhs) {
        // ...
    }
    point(const point&) =delete;
    point& operator=(const point&) =delete;
};
```

http://coliru.stacked-crooked.com/a/2b9ac3df84185215

- Implement move assignment.
- Demonstrate "move-only" behavior.

Perfect Forwarding

- Goal: select move copy even **inside** a called function
- Works by special rules for template reference arguments
- Most easily used in Cookbook fashion:

```
void foo( … , T&& arg, … )
    …
    … // maybe non-move/non-forward uses of `arg`
    …
    bar( … , std::forward<T>(arg), … );
    … // Important: NO further use of `arg` below!
}
```

- "*You are not expected to understand this*" (the reference link below):

http://en.cppreference.com/w/cpp/language/template_argument_deduction

- If you are eager to grasp the background rather read this:

http://thbecker.net/articles/rvalue_references/section_07.html

Example: perfect_forwarding/demo.cpp

```
class point {
public:
    const char* ctor;
    point()                 : ctor("default c'tor") {}
    point(const point&)     : ctor("copy c'tor") {}
    point(point&&)          : ctor("move c'tor") {}
};
point operator+(const point&, const point&);

template<typename T>
void foo(T&& arg, bool b = false) {
    T loc{std::forward<T>(arg)};  // no furthermore uses of `arg`
}

int main() {
    point pt1;                  foo(pt1);
    const point pt2{pt1};       foo(pt2);
    point pt3{std::move(pt1)}; foo(pt2 + pt3, true);
}
```

http://coliru.stacked-crooked.com/a/ec215758bf9e53f3

- Add printing types and ….`ctor` member.
- Run with variations and explain (or predict) output.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

New Meaning of `auto`

- Old meaning of keyword `auto`:
  - Place variable on stack
  - (still so in C)
- The meaning completely changed in C++11
  - when used as type for a variable
  - when used as function return type

http://en.cppreference.com/w/cpp/language/auto

`auto`-typed Variables

A variable given the type `auto`

- takes the type of its initialising expression
- after stripping `const` and reference
- `auto`-type (on left) may also be adorned,
  - frequently with
    - `*` (pointer)
    - `&` or `&&` (reference)
    - `const` / `volatile` (qualifier)
    - or combinations thereof
  - that way effectively restricting type of initialiser

Example: auto_variables/demo.cpp

```
int main() {
    auto a = 42;            PT(decltype(a))
    auto b = a;             PT(decltype(b))
    const int c = 2*a;      PT(decltype(c))
    auto d = c;             PT(decltype(d))
    auto& e = d;            PT(decltype(e))
    const auto& f = e;      PT(decltype(f))
    auto& g = c;            PT(decltype(g))
    auto h = &a;            PT(decltype(h))
    auto i = &c;            PT(decltype(h))
    auto* j = &a;           PT(decltype(h))
//  auto* k= a;             PT(decltype(k))
    auto z = 'z' /*+1*/;    PT(decltype(z))
}
```

http://coliru.stacked-crooked.com/a/eaef38306c4db828

- Explain (or even predict) the output.
- Also explain Compile-Errors.
- Come up with more examples.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`auto`-typed Functions

- A function given the return type `auto`
  - needs either a trailing return type
  - which follows **after** the argument list
  - separated with `->`
- Since C++14: determined from `return` statement
  - therefore implementation must be supplied
  - in case of more then one return statement …
  - … all must have the same type

Note Pitfall: Typed pointer and `nullptr` do have **different** types!

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: auto_typed_functions/demo.cpp

```
// declaration only (C++11 and C++14)
template<typename T1, typename T2>
auto add(T1&& lhs, T2&& rhs) -> decltype(lhs+rhs);
// decltype(*((std::remove_reference_t<T1>*)0)
//        + *((std::remove_reference_t<T2>*)0)) add(T1&&, T2&&);

// with implementation (C++14 only, C++11 needs trailing `-> …`)
template<typename T1, typename T2>
auto mul(T1&& lhs, T2&& rhs) { return lhs*rhs; }

template<std::size_t Bits>     // Compile-Time fixed size
struct arithmetic { /*...*/ }; // "arbitrary length" arithmetic

template<std::size_t N1, std::size_t N2>
auto operator+(arithmetic<N1>, arithmetic<N2>)
     -> arithmetic<std::max(N1, N2)+1>;

template<std::size_t N1, std::size_t N2>
auto operator*(arithmetic<N1>, arithmetic<N2>)
     -> arithmetic<N1+N2>;
```

http://coliru.stacked-crooked.com/a/0a4e2f27c3d8dbdb

- Add (`decltype`-based) tests to verify expected behaviour.

Uniform Initialisation

- C++11 unifies initialisation syntax
  - The goal is to simplify initilisation
  - In practice, classic forms must still be known …
    - … as there exists many "old" software …
    - … and probably will so for a long time
- Even worse, the new form introduced new pitfalls

Classic Initialisation Syntax

- Classic initialisation uses
  - Equal signs for simple types
  - Parentheses for constructor calls
  - Curly braces for aggregates

http://en.cppreference.com/w/cpp/language/initialization

Example: init_syntax/demo.cpp

```
struct point {
    double x, y;
};

int main() {
    int x = 42;                 PX(x);
    std::string hello("hi!");   PX(hello);
//  std::string empty();        PT(decltype(empty));
    std::string empty;          PX(empty);
    point pt = {3.0, 4.0};      PX(pt.x); PX(pt.y);
    int zero = int();           PX(zero);
    unsigned mask = 0xFF;       PX(mask); PT(decltype(mask));
}
```

http://coliru.stacked-crooked.com/a/08a8e43822fdc323

- What is the problem with the commented-out line?
- Since when is the initialisation syntax for `zero` valid?
- How is `zero` initialised if had class type?
- What is implicitly assumed in case of plain `unsigned`?

Brace Initialisation

- Since C++11 initial values in curly braces
  - are possible in all well-known classic contexts
  - and some more (new and lesser known)

Some ambiguities stem from also introducing initializer-lists.

http://arne-mertz.de/2015/07/new-c-features-uniform-initialization-and-initializer_list/

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): init_syntax/demo.cpp

```
int main() {
    int x{42};                      PX(x);
    std::string hello{"hi!"};       PX(hello);
    std::string empty{};            PX(empty);
    point pt{3.0, 4.0};             PX(pt.x); PX(pt.y);
    int zero{};                     PX(zero);
    auto mask{0xFF};                PX(mask); PT(decltype(mask));
//  auto mask= unsigned long{0xFF}; PX(mask); PT(decltype(mask));
}
```

- What is the type of `mask` in the currently enabled line?
- Why would you prefer the currently commented syntax?

---

https://www.youtube.com/watch?feature=player_detailpage&v=xnqTKD8uD64#t=2336

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Initializer Lists

- `std::initializer_list<T>`
  - allows to hand-over a list of values as a unit
  - access is by iterating in standard ways
- The element type `T` is templated
  - Will be deduced when all contained values have same type
  - Otherwise must be explicitly specified (or Compile Error)

http://coliru.stacked-crooked.com/a/147c36fd4e078fff

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: initializer_lists/demo.cpp

```
int main() {
//  PT(decltype({3, 2, 1, 0, -1}));
    PT(decltype(std::initializer_list<int>{3, 2, 1, 0, -1}));
    PT(decltype(std::initializer_list<float>{3.f, 2.f, 1.f, 0.f}));
    PT(decltype(std::initializer_list<double>{3, 2.5, 1, 0, -1}));
    PT(decltype(std::initializer_list<bool>{true, false, true}));

    std::array<int, 5> a{{3, 2, 1, 0, -1}};
//  std::vector<int> b{std::initializer_list<int>{3, 2, 1, 0, -1}};
    std::vector<int> b{{3, 2, 1, 0, -1}};     PX(b.size());
    std::vector<int> c{{}};                   PX(c.size());
    const std::map<int, bool> prime{
        {1, true}, {2, true}, {3, true}, {4, false}, {5,true}
    };                                        PX(prime.size());
//  std::set<bool> tf = {false, true, false}; PX(tf.size());
```

http://coliru.stacked-crooked.com/a/c1512f80224d37c9

Deducing `auto` for Initializer Lists

- Initializer lists for `auto`-typed variables
  - **are** deducted when elements have a unique type
  - are **ambiguous** if they only hold one element
  - can obviously **not** be deducted if completely empty

Initializer are **never** deduced as templated argument, only the initializer element type can be deduced – example follows later.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): initializer_lists/demo.cpp

```
int main() {
    auto ili = {3, 2, 1, 0, -1};
//  auto ili{3, 2, 1, 0, -1};
//  auto ili = std::initializer_list<int>{3, 2, 1, 0, -1};
    PT(decltype(ili));  PX(ili.size());

    auto ilf = {3.f, 2.f, 1.f, 0.f};
//  auto ilf = {3, 2.f, 1, 0, -1};
    PT(decltype(ilf));  PX(ilf.size());

    auto il3 = {true, false, true};
//  auto il3{true, false, true};
    PT(decltype(il3));  PX(il3.size());

    auto il1 = {true};
//  auto il1{true};
    PT(decltype(il1));  PX(il1.size());

//  auto il0 = {};
}
```

http://coliru.stacked-crooked.com/a/c1512f80224d37c9

- Explore the grey areas indicated above.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Accepting `std::initializer_list` Arguments

- Functions may accept arguments of type `std::initializer_list`
  - The list element type may be spelled out
  - Alternatively the function may templated the type
- Implementation may use container interface as usual
  - Iterating from `begin` to `end`
  - Range-based `for` loop

Range-based `for` loops will be covered in a later chapter.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): initializer_lists/demo.cpp

```
std::string foo(const std::initializer_list<int> &li) {
    std::ostringstream result{};
    result.setf(std::ios::boolalpha);
    result << "initializer_list<int>{";
    for (auto it = li.begin(); it != li.end(); ++it) {
        if (it != li.begin()) result << ", ";
        result << *it;
    }
    result << "}";
    return result.str();
}

int main() {
//  PX(foo(std::initializer_list<int>{3, 2, 1, 0, -1}));
    PX(foo({3, 2, 1, 0, -1}));

    auto li = {'a', 'b', 'c'};
    PT(decltype(li)); //  PX(li);
}
```

http://coliru.stacked-crooked.com/a/c1512f80224d37c9

- Generalise `foo` with respect to accepted `std::initializer_list`
- Advanced: Implement `operator<<` for initializer lists.

Resolving Problems with Initialiser Lists

- Initializer lists also use curly braces as delimiters
  - Ambiguities arise from use with constructors
  - Resolved by doubling braces
- When used to initialise `auto` typed variables
  - initialiser lists are generally deduced in C++11
  - which was changed to be only deduced after `=`

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): initializer_lists/demo.cpp

```
int main() {
    std::vector<int> c1{{3, 2}};    PX(c1.size());
    std::vector<int> c2{3, 2};      PX(c2.size());
    std::vector<int> d1{{3}};       PX(d1.size());
    std::vector<int> d2{3};         PX(d2.size());
    std::vector<int> d3(3);         PX(d3.size());
    std::vector<int> d4({3});       PX(d4.size());
    std::vector<int> e{};           PX(e.size());

    auto li3 = {1, 2, 3};           PT(decltype(li3));
//  auto li3{1, 2, 3};
    auto li1 = {1};                 PT(decltype(li1));
    auto i{1};                      PT(decltype(li));
```

http://coliru.stacked-crooked.com/a/c1512f80224d37c9

- Be sure to understand ambiguities and how they are avoided.

Range-Based Loops

- Syntax has colon in parentheses after `for`
  - Placeholder variable on left
  - Container on right
- Can benefit from `auto` typing the placeholder
  - avoid copying
  - use reference
- Then same or better performance than alternatives
  - (otherwise no one would use it)

http://en.cppreference.com/w/cpp/language/range-for

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Native Arrays

- Works by default, given:
  - Container needs to visible as native array type …
  - … **not** decayed to pointer (to first element)

Note that native arrays do not decay if handed over via reference. (This may be used to preserve array-nes in templates.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: range_for/demo.cpp

```
int main() {
    const int primes[] = {
        2, 3, 5, 7, 11, 13, 17, 19, 23, 29
    };
    for (auto e : primes)
        // ...
}
```

http://coliru.stacked-crooked.com/a/94bcea8326eb854b

STL Containers

- Works by default
  - Placeholder variable has `value_type` of iterator
  - Therefore it's a key/value `std::pair` for maps

Example (cont.): range_for/demo.cpp

```
int main() {
//  using my_container = std::vector<int>;
//  using my_container = std::deque<int>;
//  using my_container = std::list<int>;
    using my_container = std::forward_list<int>;
    my_container primes = { 2, 3, 5, 7, 11, 13, 17, 19, 23, 29 };
    for (auto e : primes)
        std::cout << e << ' ';
    using my_container = std::map<int, bool>;
//  using my_container = std::unordered_map<int, bool>;
    my_container is_prime = {
        {1, false /* primes.utm.edu/notes/faq/one.html */ },
        {2, true}, {3, true}, {4, false}, {5, true},
        // ...
    };
    for (auto e : is_prime) {
        if (e.second)
            std::cout << e.first << ' ';
    }
}
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Initializer Lists

- Works by default
  - Useful for fixed-sized lists …
  - … of values typically precalculated at Compile-Time …
  - … not (necessarily) algorithmically related

Example (cont.): range_for/demo.cpp

```
int main() {
    for (auto e : {2, 3, 5, 7, 11, 13, 17, 19, 23, 29})
        std::cout << e << ' ';
    std::cout << std::endl;
}
```

```
enum color { R, G, B };
auto all_colors = { R, G, B };
// ...
int main() {
    for (auto c : all_colors)
 std::cout << c << ' '
    std::cout << std::endl;
}
```

User-Defined Container Classes

- Work if standard iterator interface is provided

  Member function `begin()` to return (nested) iterator type …

  - … providing (at least)
    - dereference operation (`operator*`)
    - increment operation (`operator++`)
    - comparison (`operator==` and `operator!=`)
  - Member function `end()` for comparison to stop

Example (cont.): range_for/demo.cpp

Non-Member `begin()` and `end()`

- Meant to adapt non-standard containers
  - `begin()` needs to return some (one pass) iterator
  - usable to access an element with `operator*`
  - advanceable with `operator++`
  - at some point equal to what `end()` returns

---

https://isocpp.org/blog/2014/10/stdbegin-stdend

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): range_for/demo.cpp

```
enum rgb {red, green, blue};

rgb next(rgb c) {return static_cast<rgb>(c+1);}

class rgb_it {
    rgb c;
public:
    rgb_it(rgb c_) : c(c_) {}
    rgb operator*() const {return c;}
    rgb_it operator++() {c = next(c); return *this;}
    rgb_it operator++(int) {return rgb_it(next(c));}
};
bool operator==(rgb_it lhs, rgb_it rhs) {return *lhs == *rhs;}
bool operator!=(rgb_it lhs, rgb_it rhs) {return *lhs != *rhs;}

rgb_it begin(rgb) {return {rgb::red};}
rgb_it end(rgb) {return {next(rgb::blue);}

int main() {
    for (auto e : rgb{})
        std::cout << e << std::endl;
}
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Lambdas (Function Literals)

- Most often found in Functional Languages
  - if you want to learn one …

  … you may try Haskell

---

http://learnyouahaskell.com/chapters

Callables

- In C++98 Callable entities are
  - Classical Functions (as in C)
  - Classes overloading `operator()` ...
  - Call syntax for both is identical
    - Append actual arguments in parentheses
    - Maybe retrieve value returned
- C++11 added Lambdas (Function Literals)
  - For a **caller** equivalent to the above

http://en.cppreference.com/w/cpp/language/lambda

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: callable_lambda/demo.cpp

```
bool is_even_fnc(int n) { return (n%2 == 0); }

struct check_even {
    bool operator()(int n) const {return (n%2 == 0);}
};

class check_divisible {
    const int by;
public:
    check_divisible(int by_) : by(by_) {}
    bool operator()(int n) const {return (n%by == 0);}
};

void callable_basics() {
    PX(is_even_fnc(42));        PX(is_even_fnc(3));
    PX(check_even()(42));       PX(check_even()(3));
    PX(check_divisible{2}(42)); PX(check_divisible{2}(3));
    PX(check_divisible{3}(42)); PX(check_divisible{3}(3));
    PX([](int n) { return (n%2 == 0); }(42));
}
```

http://coliru.stacked-crooked.com/a/64d0447ce5e6dc6e

- Try more variations of the above.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Function Literals

- Another name for lambdas
- When handed over to caller …
  - syntax requires `[]` as prefix …
  - … followed by parameter list declaration …
  - … an optional trailing return type …
  - … finally a code block (implementation)
- Optionally inside `[ … `] context can be capture
  - Either by value (i.e. making copies)
  - Or by reference (i.e. handing over pointers)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_lambda/demo.cpp

```
void capture_basics() {
    static int x = 42;  PX(x); PX(&x);
    //      +------------------------- capture list
    //      |  +---------------------- argument list
    //      |  |        +------------- return type
    //      |  |        |           +- implementation
    //      v  v    vvvvv  vvvvvvvvvv     CALL -vv
    PX(     []() -> void* {return &x;}          ());
    PX(     []() {return "whatever";}           ());
    PX(     []() {return x;}                    ());
    PX(     []() {return &x;}                   ());

    int y = 1;  PX(y); PX(&y);
    PX(     [y]() {return y;}                   ());
    PX(     [y]   {return y;}                   ());
    PX(     [y]() mutable {return ++y;}         ());
    PX(     [&y]() {return y;}                  ());
    PX(     [&y]() {return ++y;}                ());
    PX(     [&y]() {++y;}                       ());
}
```

- Try more variations of the above.

Callables Use

- Frequently handed over to STL algorithms, e.g.
  - as predicates
  - for sorting, as ordering criteria
- Tree-based STL containers may also specify ordering

Example (cont.): callable_lambda/demo.cpp

```cpp
template<typename InIt, typename OutIt, typename Pred>
OutIt copy_if(InIt from, InIt upto, OutIt sink, Pred keep) {
    while (from != upto) {
        if (keep(*from))
                *sink++ = *from;
        ++from;
    }
    return sink;
}

void capture_nothing(const std::initializer_list<int>& values) {
    copy_if(values.begin(), values.end(),
            std::ostream_iterator<int>(std::cout, " "),
         // is_even_fnc
         // check_even{}
            [](int n) { return (n%2 == 0); }
           );
}
```

- Change example to use `check_divisible` instead of `check_even`.
- Do the same with a Lambda.
- Let the caller supply the divisor in both cases.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Return Type Deduction

- On the left of a lambda
  - neither a return type …
  - … nor `auto` is specified
- Instead trailing return type syntax may be used
- If not, return type is deduced
  - `void` if there is no return statement
  - from a single (allowed) `return` statement in C++11
  - maybe from multiple `return` statements since C++14

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_lambda/demo.cpp

void capture_basics() { static int x = 42; PX(x); PX(&x);

```
    auto f1 = []() {return &x;}         (); PT(decltype(f1));
    auto f2 = []() {return &x;}           ; PT(decltype(f2));
    auto f3 = []() -> void* {return &x;}(); PT(decltype(f3));
    auto f4 = []() {return (x) ? &x : nullptr;};
//  auto f5 = []() {if (x) return &x; else return nullptr; };
    auto f6 = []() -> const int* {if (x) return &x; else return nullptr;};
}
```

- Predict the output (type of `f1` to `f3` – look closely!)
- Why does `f4` work but `f5` is a Compile error?

Argument Type Deduction (since C++14)

- Arguments of lambdas may have types `auto`
  - emulating templated `operator()` overload

---

https://isocpp.org/wiki/faq/cpp14-language#generic-lambdas

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.): callable_lambda/demo.cpp

```
template<typename T>
class check_positive {
    bool operator()(T n) {return (n > 0);
};

template<typename T>
void generic_lambda(std::initializer_list<T> values) {
    copy_if(values.begin(), values.end(),
            std::ostream_iterator<T>(std::cout, " "),
            [](int n) { return n > 0; }
        // check_positive{}
        // [](T n) { return n > 0; }
        // [](auto n) { return n > 0; }
           );
}
```

- What "strange" behaviour has the example in the form shown.
- Which of the alternatives may only be used in C++14?

Capture Lists

- Used to feed values from the call context
  - May make a copy
  - or use reference access
  - or initialize a local
- Technically like functor data members
  - (much) less boiler plate code

http://en.cppreference.com/w/cpp/language/lambda#Lambda_capture

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Non-Capturing Lambdas

- In non-capturing lambdas the capture list is empty.
  - **It cannot be omitted!**
  - (like an empty argument list can be)
- Non-local entities are still available, i.e.
  - global variables
  - local `static` variables
- In **member functions** of a class …
  - … also `this` can be capture …
  - … giving access to all members

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_lambda/demo.cpp

```
void capture_basics() {
    PX(     []() {return "whatever";}           ());
    static int x = 42;  PX(x); PX(&x);
    PX(     []() -> void* {return &x;}          ());
    PX(     []() {return x;}                    ());
    PX(     []() {return &x;}                   ());
}
```

- Try the above and add variations.
- Add examples to demonstrate capturing `this`.

Lambdas Capturing by Value

- Capturing by value makes copies
  - Therefore typically much "safer", i.e. …
  - … valid for Lambda calls outside of definition context
  - **But: it cannot be used for* move only* types
- The capture list syntax `[=]` may be used as short-hand.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_lambda/demo.cpp

```
void capture_basics() {
    int y = 1;  PX(y); PX(&y);
    PX(     [y]() {return y;}                  ());
    PX(     [y]   {return y;}                  ());
    PX(     [y]() mutable {return ++y;}        ());
    PX(     [y]   mutable {return ++y;}        ());
    PX(     [y]() {return &y;}                 ());
    PX(     [=]() {return &y;}                 ());
}
```

- Add example capturing more than one variable.
- Do native arrays decay to pointers here case?
- Advanced: Try capturing *move only* type.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Lambdas Capturing by Reference

- Capturing by reference uses pointers behind the scenes
    - Therefore the danger of *dangling pointers* arises
    - **Be careful when to call such a lambda, especially**
    - **NEVER** call it after the defining context is discarded
- The capture list syntax `[=]` may be used as short-hand.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_lambda/demo.cpp

```
void capture_basics() {
    int y = 1;  PX(y); PX(&y);
    PX(     [&y]() {return ++y;}                ());
    PX(     [&y]() {return y;}                  ());
    PX(     [&]() {return &y;}                  ());
}
```

- Add example capturing more than one variable.
- Combine value and reference captures.
- Combine with short-hand.
- Advanced: demonstrate danger of dangling pointers.

Lambdas Initialising Locals (C++14)

- Uses of *`var`* = *`init-expr`* in capture list
  - Variable is introduced as local to lambda
  - Comparable to locals in functors
  - Name of *`var`* shadows context of definition …
  - … but only **after** *`init-expr`* has been evaluated

Example (cont.): callable_lambda/demo.cpp

```
void capture_basics() {
    static int x = 42;  PX(x); PX(&x);
    PX(     [z = x]() {return z;}               ());
    PX(     [z = &x]() {return &z;}             ());

    int y = 1;          PX(y); PX(&y);
    PX(     [z = &y]() {return &z;}             ());

    double z = 0;       PX(z); PX(&z);
    PX(     [z = &z]() {return &z;}             ());
```

- Add more examples to explore (potential) grey areas.
- Advanced: how to make *move only* types available in lambda?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Lambdas Capturing Arrays

- Be careful when capturing array variables
  - Understand when names decay to a pointers
  - Be aware of the differences between …
    - … native arrays (like `double data[10]`)
    - … and STL wrappers (like `std::array<double, 10>`)
  - Generally STL containers do not decay
  - `std::move` may be considered on last point of use

Optional Example (cont.): callable_lambda/demo.cpp

```
void capture_init_array() {
//  std::array<int, 10000> data;
    int data[10000];
    [data](){ PT(decltype(data)); }();
    [&data](){ PT(decltype(data)); }();
    [d = data]() { PT(decltype(d)); }();
    [d = data]() mutable { PT(decltype(d)); }();
    [d = &data]() { PT(decltype(d)); }();
    [d = &data]() mutable { PT(decltype(d)); }();
}
```

- Understand the difference between native array and `std::array`.
- Advanced: Demonstrate moving an `std::vector`.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Operations on Callables

- `std::function` abstracts to signature and return value
- `std::bind` transforms one callable into another one

http://en.cppreference.com/w/cpp/utility/functional

In functional programming transformations are also subsumed under the term "higher order functional programming".

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Type Erasure

- Callables generally differ in type
    - Classic function type determined by arguments and result
    - Each Functor is a distinct type
    - Same for each lambda (unspecified internal type)
- Type erasure with std::function means
    - only calling convention is carried along
    - useful to store "call-backs" in variables
    - for arguments competes with fully generic templates

http://en.cppreference.com/w/cpp/utility/functional/function

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: callable_operation/demo.cpp

```
auto is_odd(int n) {return n%2; }
auto is_even(int n) {return !(n%2); }

struct check_odd {auto operator()(int n) {return (n%2 != 0);}};
struct check_even {auto operator()(int n) {return (n%2 == 0);}};

int main() {
    PT(decltype(is_odd)    );
    PT(decltype(is_odd(3)) );                PX(is_odd(3));
    PT(decltype(is_even)   );
    PT(decltype(is_even(3)));                PX(is_even(3));
    PT(         check_odd  );                PX(check_odd{}(3));
    PT(         check_even );                PX(check_even{}(3));
    auto f1 = [](int n) {return n%2;};
    PT(decltype(f1));                        PX(f1(3));
    auto f2 = [](int n) {return !(n%2);};
    PT(decltype(f2));                        PX(f2(3));
}
```

http://coliru.stacked-crooked.com/a/cb9ba4b36734e074

- Make sure to understand the difference between types and values.)
- Why is the type the same for `is_odd` and `is_even`?
- Why is the type different for `check_odd` and `check_even`?

`std::function` Basics

- Definition resembles prototypes
  - `std::function< … >` is a template
  - Inside the square brackets go
    - The return type
    - An argument list in parentheses (only types are required)
    - (`auto` and trailing return type is also possible)

Example (cont.): callable_operation/demo.cpp

```
#include <functional>

int main() {
    std::function<bool(int)> f;
    PT(std::function<double(double, int, bool)>);
    PT(std::function<double(double d, int i, bool b)>);
    PT(std::function<auto (double, int, bool) -> double>);
    PT(std::function<void()>);

    using FVV = std::function<void(void)>;
    PT(std::function<FVV(FVV)>);
    PT(std::function<auto (FVV arg) -> FVV>);

    using std::function;
    PT(function<function<void(void)>(function<void(void)>)>);

}
```

- Learn to read `std::function` declarations like the above.
- Understand how type aliases may be applied to improve readability.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::function` Type Erasure

- Uses standard type erasure idiom, i.e. combination of
  - static polymorphism (via template member)
  - dynamic polymorphism (via virtual member)
- Results in compatibility with
  - Functions (with matching signature)
  - Function Objects (with matching overload of `operator()`)
  - Function Literals (with matching argument and return type)
- (the term "matching" indicates that conversions may be applied)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont): callable_operation/demo.cpp

```
int main() {
    std::function<bool(int)> f;
    PX((f = is_odd)                                 (42));
    PX((f = is_even)                                (42));
    PX((f = check_odd{})                            (42));
    PX((f = check_even{})                           (42));
    PX((f = [](int n) -> bool {return n%2;})        (42));
    PX((f = [](int n) {return n%2;})                (42));
    PX((f = [](int) {return true;})                 (42));
//  PX((f = [](int) {})                             (42));
//  PX((f = []() {return true;})                    (42));
//  PX((f = [] {return true;})                      (42));
    PX((f = [](bool v) -> void* {return 0;})        (42));
}
```

- Understand why type differences are acceptable (sometimes).
- Explain why the commented-out lines do not compile.
- Explain the type conversions in the last line.

Output is not such important in the above examples – i.e. the interesting part is in the assignment expression, not what PX finally prints.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::function` Supported Operations

- Supported operations include
  - Copying / Moving to compatible type
  - Explicit conversion to `bool` (to test validity)
  - but: comparison only with `nullptr`
- (the term "compatible" indicates that conversions may be applied)

Example (cont): callable_operation/demo.cpp

```
int main() {
    std::function<bool(int)> f;         PT(decltype(f));
    std::function<int(short)> f2;       PT(decltype(f2));
                                        PT(decltype(f2 = f));
                                        PT(decltype(f = f2));
    std::function<void(int)> f3{f2};    PT(decltype(f3 = f));
                                     // PT(decltype(f = f3));
                                     // PT(decltype(f3 = f2));

    PX(f ? "nullptr" : "not nullptr");
    PX((bool)(f = nullptr));
//  PX(      (f = nullptr));
    PX(f ? "nullptr" : "not nullptr");
    PX(f == nullptr); PX(nullptr == f);
    PX(f != nullptr); PX(nullptr != f);
//  PX(f == f);
}
```

- Understand that type differences are not carried with assignments.
- Why are only some `std::function`-types are assignable, others not?
- Understand the automatic conversions rules in a `bool` context.
- Why does the last line not compile even though `f` is a `nullptr`?
- Add more examples for grey areas.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::function` vs. Templating

- Using `std::function` as argument …
  - … restricts the argument type
  - Hence Compile Errors messages directly refer to the call
- Using fully generic (templated) arguments …
  - … detects problems only inside the implementation
  - Hence Compile Errors messages may be rather indirect

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont): callable_operation/demo.cpp

```
std::size_t count(const std::initializer_list<int>& values,
                  std::function<bool(int)> pred) {
    auto result = std::size_t{0};
    for (auto v : values)
        if (pred(v)) ++result;
    return result;
}

int main() {
    PX(count({2, 3, 12, 7, 5}, is_odd));
    PX(count({2, 3, 12, 7, 5}, check_odd{}));
    PX(count({2, 3, 12, 7, 5}, [](int n) {return (n%2 != 0);}));
    PX(count({2, 3, 12, 7, 5}, [](int n) -> bool {return n%2;}));
    PX(count({2, 3, 12, 7, 5}, [](int n) {return n%2;}));

    std::function<bool(int)> f = [](int n) {return n%2;};
//  std::function<void(int)> f = [](int) {};
//  std::function<bool()> f = []{return false;};
    PX(count({2, 3, 12, 7, 5}, f));
//  PX(count({2, 3, 12, 7, 5}, [](int) {}));
//  PX(count({2, 3, 12, 7, 5}, [] {return false;}));
}
```

- Why are the commented-out lines Compile Errors?

Example (cont): callable_operation/demo.cpp

```
template<typename Pred>
std::size_t count(const std::initializer_list<int>& values,
                  Pred pred) {
    auto result = std::size_t{0};
    for (auto v : values)
        if (pred(v)) ++result;
    return result;
}
```

- Test the above with the same tests (`main` program) as before.
- Also compare error messages for lines that do not compile.

```
template<typename Pred>
std::size_t count(const std::initializer_list<int>& values,
                  Pred pred) {
    return std::count_if(values.begin(), values.end(), pred);
}
```

- Try the above with the same tests as before.
- Again compare error messages for lines that do not compile.
- Rewrite the above to use an `std::function` argument for `pred`.
- Again compare error messages for lines that do not compile.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Transformations

- `std::bind` transforms a callable to another callable with
  - fewer arguments (some bound to fixed values)
  - with different argument order
  - (or a combination of both)
- Also member functions can be handled
  - Object goes as first argument
  - (as is technically when transferring arguments on the stack)

http://en.cppreference.com/w/cpp/utility/functional/bind

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_operation/demo.cpp

```
#include <functional>

long double powerof(long double value, int exponent) {
    long double result{1};
    while (exponent-- > 0)
        result *= value;
    return result;
}

int main() {
    using std::placeholders::_1;
    using std::placeholders::_2;
    PX(std::bind(powerof, _1, 2)        (4));
    PX(std::bind(powerof, _1, 3)        (4));
    PX(std::bind(powerof, 2, 3)         ());
    PX(std::bind(powerof, _1, _2)       (5, 2));
    PX(std::bind(powerof, _1, _2)       (5, 3));
    PX(std::bind(powerof, _2, _1)       (2, 5));
    PX(std::bind(powerof, _2, _1)       (3, 5));
}
```

- **Understand that the result of `std::bind` is a callable.**
- Be sure to understand the role of *placeholders* (like `_1`, `_2`).
- (Do **not** think about how `std::bind` might be implemented!)

Reference Wrappers

- `std::bind` uses value semantics for its arguments
- Therefore it can/should not be used
  - For arguments to be modified
  - With move only types
  - With types that are expensive to copy
- The solution is provided via reference wrappers

http://en.cppreference.com/w/cpp/utility/functional/reference_wrapper

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::cref` and `std::ref`

- `std::ref` returns an `std::reference_wrapper`
  - holding its argument as `const` (non-modifiable) reference
  - therefore allows to hand *move-only* types to `std::bind`
  - and also improves performances for types expensive to copy
- `std::cref` returns an `std::reference_wrapper`
  - holding its argument as non-`const` (modifiable) reference
  - that way "removing" value semantics from `std::bind` arguments

http://en.cppreference.com/w/cpp/utility/functional/ref

Note that `std::cref` rather communicates what it is intended and gives protection when the intention is violated. It is not strictly necessary as `std::ref` wraps non-modifiable arguments automatically with `const`.

Example (cont.): callable_operation/demo.cpp

```
class xcopy {
    bool b;
    // ... assume more (expensive to copy) members
public:
    xcopy() : b(false) {}
    bool toggle() {return (b = !b);}
    bool get() const {return b;}
};

int main() {
    xcopy cp;  PX(cp.get());

    PX((std::bind(&xcopy::toggle,  std::ref(cp)))   ());
    PX((std::bind(&xcopy::toggle,  std::ref(cp)))   ());
    PX((std::bind(&xcopy::toggle,           cp ))   ());
//  PX((std::bind(&xcopy::toggle, std::cref(cp)))   ());
    PX((std::bind(&xcopy::get,    std::cref(cp)))   ());
    PX((std::bind(&xcopy::get,     std::ref(cp)))   ());
    PX((std::bind(&xcopy::get,              cp ))   ());
}
```

- Understand the benefits of using `std::ref` and `std::cref`.
- Why leads the commented-out line to a Compile Error?
- (Feel free to add more examples, also failing ones.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_operation/demo.cpp

```
class ncopy {
    bool b;
    // assume more members (expensive to copy but cheap to move)
public:
    ncopy() : b(false) {}
    bool toggle() {return (b = !b);}
    operator bool() const {return b;}
    ncopy(const ncopy&)            =delete;
    ncopy& operator=(const ncopy&) =delete;
    ncopy(ncopy&&)                 =default;
    ncopy& operator=(ncopy&&)      =default;
};

int main() {
    ncopy mv;  PX(mv);

//  PX((std::bind(&ncopy::toggle,           mv ))   ());
//  PX((std::bind(&ncopy::toggle,           mv ))   ());
//  PX((std::bind(&ncopy::operator bool,    mv ))   ());
}
```

- Why does all the above fail with a *non-copyable* type?
- How can `std::ref` and `std::cref` solve the problem?

Example (cont.): callable_operation/demo.cpp

```
int main() {
    // assuming classes `xcopy` and `ncopy` and
    // object instances `cp` and `mv` as before

    const auto& ccp = cp;  PX(ccp.get());

//  PX((std::bind(&xcopy::toggle,  std::ref(ccp))) ());
//  PX((std::bind(&xcopy::toggle, std::cref(ccp))) ());
    PX((std::bind(&xcopy::get,     std::ref(ccp))) ());
    PX((std::bind(&xcopy::get,    std::cref(ccp))) ());
    PX((std::bind(&xcopy::get,              ccp )) ());

    const auto& cmv = std::move(mv);  PX(cmv);

//  PX((std::bind(&ncopy::toggle,         std::ref(cmv)))  ());
//  PX((std::bind(&ncopy::toggle,        std::cref(cmv)))  ());
    PX((std::bind(&ncopy::operator bool, std::cref(cmv)))  ());
    PX((std::bind(&ncopy::operator bool,  std::ref(cmv)))  ());
//  PX((std::bind(&ncopy::operator bool,           cmv ))  ());
}
```

- Explain what works and why the commented-out lines fails.

Typical Uses of `std::bind`

- These are the most typical uses of `std::bind`:
  - Reduce or alter the arguments of a given function …
  - … to make it fit for a specific purpose
  - Specialize a (more) generic function …
  - … to be applied to all elements in a container

Note that lambdas are often more easily understood as non-trivial usages of `std::bind`. This topic will be covered later.

Example (cont.): callable_operation/demo.cpp

```cpp
struct xy {
    double x, y;
    std::string to_string() const {
        return "x=" + std::to_string(x)
           + ", y=" + std::to_string(y);
    }
};
class polygon {
    const std::vector<xy> points;
public:
    polygon(std::initializer_list<xy> pts)
        : points(pts.begin(), pts.end()) {
    }
    using pen_t = std::function<void(double x, double y)>;
    void draw(pen_t pen) const {
        if (points.empty()) return;
        for (const auto& pt : points)
            pen(pt.y, pt.x);
        pen(points.front().y, points.front().x);
    }
};
```

- Understand the above classes, as the next examples build on them.

Example (cont.): callable_operation/demo.cpp

```
std::ostream &testpen(std::ostream &lhs, double x, double y) {
    return lhs << "<x:" << x << "|y:" << y << ">";
}

void function_callback() {
    std::ostringstream os;
    using namespace std::placeholders;
    const xy a{1.5, 7.0};   PX(a.to_string());
    const xy b{2.0, 3.0};   PX(b.to_string());
    const xy c{9.1, 2.5};   PX(c.to_string());
    polygon::pen_t adapted_pen =
        std::bind(testpen, std::ref(os), _1, _2);
    PT(decltype(adapted_pen));
    const auto triangle = polygon({a, b, c});
    PX(triangle.draw(adapted_pen), os.str());
}
```

- Which error has slipped into the above code? (Note the output!)
- May the type of `adapted_pen` be changed into `auto`.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_operation/demo.cpp

```cpp
class clazz {
    const char *nm;
public:
    clazz(const char *n) : nm(n) {}
//  clazz(const clazz&) =delete;
//  clazz& operator=(const clazz&) =delete;
    void a(int v) const {os << nm <<  ".a(" << v << ") "; }
    void b(int v) const {os << nm <<  ".b(" << v << ") "; }
    void c(int v) const {os << nm <<  ".c(" << v << ") "; }
    void d(int v) const {os << nm <<  ".d(" << v << ") "; }
    static std::ostringstream os;
    static std::string os_str() {
        const auto result = os.str();
        os.str(std::string{});
        return result;
    }
};
std::ostringstream clazz::os;
```

- Understand the above (demo) class, as the next examples builds on it.
- What needs to be changed to turn it into a *move only* type?

Example (cont.): callable_operation/demo.cpp

```
auto b_for_each_arg(const clazz& obj,
                    std::initializer_list<int> arg_list) {
    using std::placeholders::_1;
    std::for_each(arg_list.begin(), arg_list.end(),
                  std::bind(&clazz::b, std::ref(obj), _1)
                 );
    return clazz::os_str();
}

int main() {
    clazz x("x"); clazz y("y"); clazz z("z");
    PX(b_for_each_arg(x, {2, 3, 5, 7, 11}));
}
```

- Which member function of `clazz` is called here repeatedly?
- For which object?
- Where come the arguments from?
- How could `std::bind` be replace with a lambda?
- (Make some changes to test your understanding.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_operation/demo.cpp

```
auto b_for_each_obj(std::initializer_list<clazz> obj_list,
                    int arg) {
    using std::placeholders::_1;
    std::for_each(obj_list.begin(), obj_list.end(),
                  std::bind(&clazz::b, _1, arg)
                 );
    return clazz::os_str();
}

int main() {
    clazz x("x"); clazz y("y"); clazz z("z");
    PX(b_for_each_obj({x, y, z}, 42));
}
```

- Which member function of `clazz` is called here repeatedly?
- Where come the objects from?
- Where comes the argument from?
- How could `std::bind` be replace with a lambda?
- What if the objects are non-copyable?
- (Make some changes to test your understanding.)

Example (cont.): callable_operation/demo.cpp

```
auto for_each_objptr(std::initializer_list<clazz*> objptr_list,
                     void (clazz::*mfn)(int) const,
                     int arg) {
    using std::placeholders::_1;
    std::for_each(objptr_list.begin(), objptr_list.end(),
                  std::bind(mfn, _1, arg)
                 );
    return clazz::os_str();
}

int main() {
    clazz x("x"); clazz y("y"); clazz z("z");
    PX(for_each_objptr({&x, &y, &z}, &clazz::c, 42));
}
```

- Which member function of `clazz` is called here repeatedly?
- Where come the objects from?
- What if the objects are non-copyable?
- Where comes the argument from?
- How could `std::bind` be replace with a lambda?
- (Make some changes to test your understanding.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::bind` vs. Lambdas

- `std::bind` generally avoids writing small wrappers
  - Lambdas are similar powerful …
  - … and oft more easily understood
- Especially the difference is more visible between
  - Call be value and
  - Call by reference

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): callable_operation/demo.cpp

```
struct xy {
    // ... from earlier example
};
class polygon {
    // ... from earlier example
}
std::ostream &testpen(std::ostream &lhs, double x, double y) {
    return lhs << "<x:" << x << "|y:" << y << ">";
}

int main() {
    std::ostringstream os;
    const xy a{1.5, 7.0};   PX(a.to_string());
    const xy b{2.0, 3.0};   PX(b.to_string());
    const xy c{9.1, 2.5};   PX(c.to_string());
    auto adapted_pen = [&os](int y, int x) {testpen(os, x, y);};
//  using namespace std::placeholders;
//  auto adapted_pen = std::bind(testpen, std::ref(os), _2, _1);
    PT(decltype(adapted_pen));
}
```

- How gets the order of arguments (`x` <=> `y`) changed in the lambda?
- How gets the order of arguments changed with `std::bind`?
- (Unrelated: What changes if `const` is prepended to `auto adapted_pen`?)

Example (cont.): callable_operation/demo.cpp

```
void relativepen(std::ostream &lhs, double xd, double yd) {
     lhs << "<xd:" << xd << "~yd:" << yd << ">";
}
```

Here the pen assumes relative movements (distances), while `polygon` supplies absolute `x` and `y` positions.

```
int main() {
    // ...
    const auto adapted_pen =
       [&os, xc=0.0, yc=0.0](double y, double x) mutable {
           relativepen(os, x-xc, y-yc); xc = x; yc = y;
       };
   // ...
}
```

- Understand how the lambda aobve achieves the adaption.
- Note that `std::bind` is **not** powerful enough to do this!
- On which assumption does the code still rely?
- What to change if C++14 *init captures* are not available?
- (So, why might you want to avoid init captures here anyway?)

Example (cont.): callable_operation/demo.cpp

```
auto for_each_mfn(const clazz& obj,
                  std::initializer_list<void (clazz::*)(int) const
                                       > mfnptr_list,
                  int arg) {
    using std::placeholders::_1;
    std::for_each(mfnptr_list.begin(), mfnptr_list.end(),
               // std::bind(_1,  std::cref(obj), arg)
                  [&obj, arg](void (clazz::*mfnptr)(int) const) {
                      (obj.*mfnptr)(arg);
                  }
                 );
    return clazz::os_str();
}

int main() {
    clazz x("x"); clazz y("y"); clazz z("z");
    PX(for_each_mfn(z, {&clazz::b, &clazz::a, &clazz::c}, 42));
}
```

- Which member function of `clazz` are called here in which order?
- Where come the object and the argument from?
- Note that the lambda can **not** replaced here with `std::bind`!
- Improve readability with a type alias for the member function pointer.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Features for Classes

- Direct Member Initialisation
- Constructor inheritance and delegation
- Explicit "automatic" type conversions
- `final` and `override`

Direct Member Initialisation

- Data members may be initialised directly
- Lower precedence as constructor MI-lists
- May not use other members (MI-list may)

http://en.cppreference.com/w/cpp/language/data_members#Member_initialization

(see box *default member initialisation*)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: member_init/demo.cpp

```
class xy {
public:
    const double xr = 0.0;
    const double yr = 0.0;
    xy(double x, double y) : xr(x), yr(y) {}
//  xy(double x = 0.0, double y = 0.0) : xr(x), yr(y) {}
protected:
    xy() {}
};

int main() {
    xy a;                // direct initialization
    PX(a.xr); PX(a.yr);
    xy b{3.5, 1.0};      // ctor with two arguments
    PX(b.xr); PX(b.yr));
}
```

http://coliru.stacked-crooked.com/a/3e12686ae858f3b1

- How is this different from default values to constructor arguments?
- What effect has the `protected` default constructor?
- Is uniform initialisation possible two?
- May an initializer refer another (already initialized) member?

Constructor Delegation

- Class constructor may call other constructor
- Useful to have
  - one "work-horse" constructor
  - and several "convenience" constructors

http://en.cppreference.com/w/cpp/language/initializer_list#Delegating_constructo

---

Be aware of special rules for the handling of exceptions in constructor member initialisation lists enclosed in `try` blocks.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): member_init/demo.cpp

```
class xy {
public:
    const double xr{}, yr{};   // rectangular coordinates
    const double rad{}, deg{}; // polar coordinates
private:
    xy(double x, double y, double r, double d)
        : xr(x), yr(y), rad(r), deg(d) {}
public:
    enum coord { rect_tag, pol_tag };    // these four lines are
    template<coord> struct ctor_tag {};  // the "c'tor-tag magic"
    using rect = ctor_tag<rect_tag>;     // (which is not the
    using pol = ctor_tag<pol_tag>;       // topic covered here)
    xy(double x, double y, rect) // c'tor for rectangular
        : xr(x), yr(y)
        , rad(std::sqrt(x*x + y*y)), deg(std::atan2(x, y)) {}
    xy(double r, double d, pol) // c'tor for polar
        : xr(r*std::cos(d)), yr(r*std::sin(d))
        , rad(r), deg(d) {}
}
```

- Identify the delegating and the delegated-to constructors.
- Why is the constructor with four arguments made `private`?
- (Unrelated: Explain the constructor-tag magic.)

Optional Example (cont.): member_init/demo.cpp

```
struct clazz {
    std::ostream &os;
    ~clazz() {
        os << "clazz d'tor executed";
    }
    clazz(std::ostream& os_, bool b) try : os(os_) {
        if (b) throw "clazz(bool) c'tor body";
    } catch(const char *thrown_by) {
        os << "caught in clazz(bool) c'tor: " << thrown_by;
    };
    clazz(std::ostream &os_, int x) try : clazz(os_, x == 0) {
        if (x == 1) throw "clazz(int) c'tor body";
    } catch(const char *thrown_by) {
        os << "caught in clazz(int) c'tor: " << thrown_by;
    }
};
```

- Analyze the code example for constructor delegation.
- Predict behaviour when constructor argument `x` is `false`.
- As before, for `true`, `0`, `1`, and `42` (of course :-)).

Optional Example (cont.): member_init/demo.cpp

```
struct clazz {
    // ... as shown before
};

int main() {
    std::cout.setf(std::ios::boolalpha);
    std::ostringstream os;
    try {
        auto arg = false;                     PX(arg);
    //  auto arg = true;                      PX(arg);
    //  auto arg = 0;                         PX(arg);
    //  auto arg = 1;                         PX(arg);
    //  auto arg = 42;                        PX(arg);
        PX(clazz{os, arg}, os.str());         PX(os.str());
    } catch (const char *thrown_by) {
        PX("caught in test context", thrown_by);
    }
}
```

- Try with various argument types and values and explain the output.
- Understand differences to ordinary `try`-`catch` chaining.

Constructor Inheritance

- Base class constructors can be made visible
    - `using`-syntax as for other members
    - overwriting still possible
- Specially useful if derived class
    - add no members at all
    - or added members can be directly initialised

http://en.cppreference.com/w/cpp/language/using_declaration#Inheriting_constru

Example (cont.): member_init/demo.cpp

```
// assuming class `xy` as shown earlier

class point : private xy {
public:
    using xy::xr;   // make data members publicly visible
    using xy::yr;
    using xy::rad;
    using xy::deg;

    using xy::rect; // make data types publicly visible
    using xy::pol;

    using xy::xy;   // inherit all(?) constructors

    // explicit delegation necessary here
    std::string xy_to_string() const {return xy::to_string();}
};
```

- Which constructors exactly are inherited?
- Is it possible to exclude some (unwanted) constructors.
- Rewrite the code **not** using constructor inheritance.
- Why is explicit delegation necessary in the last case?

Class Specific Type Conversions

- C++ provides automatic ways to automatically convert …
  - … a type or class **into** some other class
  - … some class **into** some other class or type

http://en.cppreference.com/w/cpp/language/converting_constructor

http://en.cppreference.com/w/cpp/language/cast_operator

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: explicit_typecast/demo.cpp

```
struct other {};
struct clazz {
    clazz() {} // default c'tor (reason see below)
    clazz(int)  {}                     // from int
    clazz(const other&) {}             // from other
    operator long() const {return {};} // to long
    long to_long() const {return {};} // member function
    operator other() const {return {};} // to other
};

void foo(clazz) {}
void flong(long) {}

int main() {
    foo(42);
    clazz z; // default c'tor required here!
    flong(z.to_long());
    // ...
}
```

http://coliru.stacked-crooked.com/a/57b243b26d57175c

- Learn more about automatic conversions by adding to the code above.

`explicit` Conversions

- Adding the keyword `explicit` disables automatic conversion
- It is still available with
  - Constructor call syntax
  - Cast operation
  - Member function call syntax

http://en.cppreference.com/w/cpp/language/explicit

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): explicit_typecast/demo.cpp

```
struct clazz {
             clazz() {} // default c'tor
             clazz(char) {}                     // from char
    explicit clazz(int)  {}                     // from int
             clazz(const other&) {}             // from other
    explicit operator bool() const {return {};} // to bool
//           operator char() const {return {};} // to char
    explicit operator long() const {return {};} // to long
              long to_long() const {return {};} // member function
            operator other() const {return {};} // to other
}
```

- Understand `explicit` conversions by changing to the code above.
- (Adapt the tests you have written so far.)

Special Case of `operator bool()`

- `explicit` type cast operations to bool
  - are still automatic in *boolean context*
  - need to be carried out explicit otherwise

---

http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2333.html

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): explicit_typecast/demo.cpp

```cpp
// assuming class `clazz` from the previous examples ...

int main() {
    clazz z;
//  bool b{z};
//  fbool(z);
    fbool(bool(z));
//  fbool(bool{z});
    fbool((bool)z);
    fbool(static_cast<bool>(z));
    if (z) {}
    if (!z) {}
//  if (z == true) {}
}
```

- Understand the rules for **`explicit`** `operator bool()`.
- (Enable commented-out lines and watch the Compile Errors.)

`final` and `override`

- `final` locks a member function of a class against overwriting
  - Mainly this allows the compiler to generate more efficient code
  - Also member functions cannot any more be unintentionally replaced
- `final` can also be applied to a whole class …
  - … with the effect that cannot be derived from it any more
- `override` expresses the intention that a overriding is intended
  - The compiler can check for the existence in a base class
  - If not it can provide better more meaningful message
  - (It can help much to avoid some hard to spot problems)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: override_final/demo.cpp

```
class clazz {
public:
 // virtual void m2() {}
    virtual void m2() final {}
 // void m3() override {}
    void m3() {}
};

class other : public clazz {
//  virtual void m1() override {}
    virtual void m1() const override {}
//  virtual void m2() {}
    virtual void m3() {}
//  virtual void m3() override {}
};
```

http://coliru.stacked-crooked.com/a/a22a7e3292779248

- Switch between member functions (commented-out / not).
  - After each change: Compile and watch the Errors.
  - Explain! (Or better yet predict what will happen)
- Also apply final to the whole class `clazz`.
- Loosely related: Does `private:` / `public:` make a difference?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Templated Variables (C++17)

- Variable can be templates too
  - Useful for providing constants of different type
  - E.g. accessible from templated functions

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: template_variable/demo.cpp

```
template<typename T> const T PI;
template<> const double PI<float> = 2*std::acos(0.f);
template<> const double PI<double> = 2*std::acos(0.0);

int main() {
    std::cout.precision(10);

    PT(       decltype(2*std::acos(0.f)));
    PT(       decltype(PI<float>));
    PX(                PI<float> );
    PX(std::to_string(PI<float>));

    PT(       decltype(2*std::acos(0.0)));
    PT(       decltype(PI<double>));
    PX(                PI<double> );
    PX(std::to_string(PI<double>));
}
```

http://coliru.stacked-crooked.com/a/3e583356c574d16f

- Is there a difference between the `PI<float>` and the `PI<double>`?
- If yes, how big?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Smart Pointers

- Unique Ownership
- Shared Ownership

http://en.cppreference.com/w/cpp/memory#Smart_pointers

Unique Ownership

- Unique pointers manage memory
  - used by a single object (of type `T`)
  - **or** used by `N` objects of type `T`
- Technically `std::unique_ptr` is an RAII-wrapper for
  - heap allocations with `new T` or `new T[N]`
  - automating `delete` **or** `delete[] operation
- Performance and memory footprint same as for native pointer

http://en.cppreference.com/w/cpp/memory/unique_ptr

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: unique_pointer/demo.cpp

```
struct clazz {
    int x;
    int get_x() const {return x;}
    int set_x(int x_) {return (x = x_);}
#define P(t)\
    std::cout << "=> clazz " t " @" << this << std::endl;
    clazz()                 {P("default c'tor")}
    clazz(int x_) : x(x_)   {P("(int) c'tor")}
    clazz(const clazz&)     {P("copy c'tor")}
    clazz(clazz&&)          {P("move c'tor")}
    ~clazz()                {P("d'tor")}
    clazz& operator=(const clazz&) =delete;
    clazz& operator=(clazz&&) =delete;
#undef P
    char sizer[20];
};
```

http://coliru.stacked-crooked.com/a/fbef0dab3cdd9059

- Understand how the above class (`clazz`) tracks c'tor and d'tor calls.
- (The class is used in all of the `std::unique_ptr` examples.)

Creating `std::unique_ptr`

- Either: by two consecutive operations:
  - result from `new` saved in temporary native pointer …
  - … then used to initialise the `std::unique_ptr` object
- Or: by using `std::make_unique`
  - (officially introduced wth C++14)
  - Simpler and no need to worry about throwing operations

http://en.cppreference.com/w/cpp/memory/unique_ptr/make_unique

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: unique_pointer/demo.cpp

```
std::unique_ptr<clazz> factory(int a, int b) {
   return std::unique_ptr<clazz>{new clazz(a*b)};
}

int main() {
    PX(sizeof(clazz));
    PX(sizeof(clazz*));

    std::unique_ptr<clazz> ptr{new clazz{42}};
    PX(sizeof(ptr));
    PX(sizeof(*ptr));   PT(decltype(*ptr));
//  PX(sizeof(ptr[0])); PT(decltype(ptr[0]));


    std::unique_ptr<clazz[]> arr{new clazz[3]{}};
    PX(sizeof(arr));
//  PX(sizeof(*arr));   PT(decltype(*arr));
    PX(sizeof(arr[0])); PT(decltype(arr[0]));

//  std::unique_ptr<clazz> pt2{ptr};
    std::unique_ptr<clazz> pt2{std::move(ptr)};

    std::unique_ptr<clazz> pt3{factory(6, 7)};
}
```

Example (cont.): unique_ptr/demo.cpp

```
int main() {
    {
        auto ptr = std::make_unique<clazz>();
        PX(ptr.get());
        auto pt2 = std::make_unique<clazz>(6*7);
        PX(pt2.get());
    }
    std::unique_ptr<clazz[]> arr;
    PX(arr.get());

    arr = std::make_unique<clazz[]>(std::size_t{3});
    PX(arr.get());  PX(&arr[0]);  PX(&arr[1]); PX(&arr[2]);

    arr = std::make_unique<clazz[]>(std::size_t{2});
    PX(arr.get());  PX(&arr[0]);  PX(&arr[1]);
}
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Testing `std::unique_ptr`

- There is the equivalent of a `nullptr`
  - Test by using `std::unique_ptr` in boolean context

http://en.cppreference.com/w/cpp/memory/unique_ptr/operator_bool

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): unique_ptr/demo.cpp

```
int main() {
    {
        auto ptr = std::make_unique<clazz>();
        PX(ptr.get());
        auto pt2 = std::make_unique<clazz>(6*7);
        PX(pt2.get());
    }
    std::unique_ptr<clazz[]> arr;
    PX(arr.get());

    arr = std::make_unique<clazz[]>(std::size_t{3});
    PX(arr.get());  PX(&arr[0]);  PX(&arr[1]); PX(&arr[2]);

    arr = std::make_unique<clazz[]>(std::size_t{2});
    PX(arr.get());  PX(&arr[0]);  PX(&arr[1]);
}
```

Pointee Access via `std::unique_ptr`

- If `std::unique_ptr` manages memory for **one** `T` access
  - is by `operator*` (whole `T`),
  - and `operator->` (member access)
- If `std::unique_ptr` manages memory for an array of `T`
  - access is by `operator[i]` (*i-th* element of array),
  - may be followed by dot (`.`) for member access

http://en.cppreference.com/w/cpp/memory/unique_ptr/operator*

http://en.cppreference.com/w/cpp/memory/unique_ptr/operator_at

Example (cont.): unique_ptr/demo.cpp

```
int main() …{
    {
        auto ptr = std::make_unique<clazz>(5);
        PX(ptr->get_x());
        PX(ptr->set_x(2));
        PX(ptr->x);
    }
    {
        auto arr = std::make_unique<clazz[]>(2);
        PX(arr.get());
        PX(arr[0].x++);
        PX(arr[1].set_x(arr[0].get_x()));
    }
    {
        auto pt2 = std::make_unique<double>(1.0/9.0);
        PX(*pt2);
    }
}
```

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Using `std::unique_ptr` with Legacy Code

- `get()`
  - gives access to the pointer held
  - keeping ownership
- `release()`
  - hands over the pointer held to new owner

http://de.cppreference.com/w/cpp/memory/unique_ptr/get

http://de.cppreference.com/w/cpp/memory/unique_ptr/release

Example (cont.): unique_ptr/demo.cpp

```
int main() {
    {
        std::unique_ptr<clazz> ptr;          PX((bool)ptr);
        ptr = std::make_unique<clazz>();     PX((bool)ptr);
        auto pt2 = std::move(ptr);           PX((bool)pt2);
                                             PX((bool)ptr);
        pt2 = nullptr;                       PX((bool)pt2);
    }
    auto arr = std::make_unique<clazz>();    PX((bool)arr);
    arr = nullptr;                           PX((bool)arr);
}
```

`std::unique_ptr` as Data Member

- `std::unique_ptr` is well suited to be used as data member
  - The class in which it is used gets *move only* by default
  - Often there is no need to implement move operations
- A copyable class with `std::unique_ptr` data members …
  - … needs to be analyzed for its "cloning semantics
  - This **either** leads to implementing copy operations
  - **or** might turn out ownership isn't really unique

In the latter case `std::shared_ptr` might provide the required semantics "out-of-the-box".

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): unique_ptr/demo.cpp

```
class point {
    std::unique_ptr<char[]> nm;
    double xc, yc;
public:
    point(const char *name, double xc_ = 0.0, double yc_ = 0.0)
        : nm(std::strcpy(new char[std::strlen(name)+1], name))
        , xc(xc_), yc(yc_)
    {}
    friend std::ostream& operator<<(std::ostream &lhs,
                                    const point &rhs) {
        lhs << "point{nm=" << (nm ? &rhs.nm[0] : "??")
            << ", xc=" << rhs.xc << ", yc=" << rhs.yc << "}";
    }
};

int main() {
    point x{"first", 3.0, 4.5}; PX(x);
    point y{"other"};           PX(y);
//  point z{x};                 PX(z); PX(x);
    // ...
}
```

- Modify (and add to) the above code to verify *move only* behaviour.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::unique_prt` Pitfalls to Avoid

- Same native pointer used to initialise several `std::unique_ptr`
- Address of non-heap memory used to initialise `std::unique_ptr`
- Type definition does not match heap allocation:
  - `std::unique_ptr<T> p{new T[N]};`
  - `std::unique_ptr<T[]> p{new T};`

Also see item 21 in Scott Meyers book Effective Modern C++ for reasons when using `std::unique_ptr` might cause memory leaks under rare conditions (also depending on code generation) and how to protect against this.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): unique_ptr/demo.cpp

```
int main() {
    int x = 0xDEADBEEF;                          PX(&x);     PX(x);
    auto ptr = std::unique_ptr<int>{&x};     PX(&*ptr);  PX(*ptr);
    auto p = new int{6*7};                   PX(p);      PX(*p);
    auto pt1 = std::unique_ptr<int>{p};      PX(&*pt1);  PX(*pt1);
//  auto pt2 = std::unique_ptr<int>{p};
//  auto pt3 = std::unique_ptr<int>{pt1.get()};
//  delete p;
//  std::unique_ptr<clazz> ptr{new clazz[5]};
//  std::unique_ptr<clazz[]> arr{new clazz(5)};
}
```

- Explain why the commented-out lines would cause undefined behaviour.
- (Also try to predict the point (and maybe kind) of failure.)

For more information on *Undefined Behaviour* (or *UB* in short) see here:

http://en.cppreference.com/w/cpp/language/ub

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Shared Ownership

- Shared pointers manage memory
  - used by a single object of type `T`
  - when there are (potentially) many referrers
- Technically `std::shared_ptr` uses a reference count
  - incrementing it when another (new) referrer occurs
  - decrementing it when some referrer goes out of scope
  - de-allocating the referred-to object when it drops to 0
- Performance typically close to native pointer
- Memory footprint may be somewhat larger

http://en.cppreference.com/w/cpp/memory/shared_ptr

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: shared_pointer/demo.cpp

```
int main() {
    PX(sizeof(clazz));  // assuming class `clazz` as has been
    PX(sizeof(clazz*)); // used in `std::unique_ptr` examples

    std::shared_ptr<clazz> ptr{new clazz{42}};
    PX(sizeof(ptr)); PX(sizeof(*ptr)); PT(decltype(*ptr));
    PX(ptr.get());  PX(ptr.use_count());  PX(ptr.unique());

    std::shared_ptr<clazz> pt2{ptr};
    PX(pt2.get());  PX(pt2.use_count());  PX(pt2.unique());
    PX(ptr.get());  PX(ptr.use_count());  PX(ptr.unique());

    std::shared_ptr<clazz> pt3;
    PX(pt3.get());  PX(pt3.use_count());  PX(pt3.unique());

    pt3 = factory(6, 7);
    // ...
}
```

http://coliru.stacked-crooked.com/a/0fc58836466e4dd6

- Add as much code as necessary to improve your understanding.

Creating `std::shared_ptr`

- Either: by two consecutive operations:
  - result from `new` saved in temporary native pointer …
  - … then used to initialise the `std::shared_ptr` object
- Or: by using `std::make_shared`
  - Simpler and no need to worry about throwing operations
  - Usually better performance and more efficient use of resources

http://en.cppreference.com/w/cpp/memory/shared_ptr/make_shared

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): shared_pointer/demo.cpp

```
int main() {
    auto ptr = std::make_shared<clazz>();
    PX(ptr.get()); PX(ptr.use_count()); PX(ptr.unique());

    auto pt2 = std::make_shared<clazz>(6*7);
    PX(pt2.get());
    std::shared_ptr<clazz> data[5];
    for (auto &e : data) {
        PX("before", e.use_count());
        PX((e = ptr).get());
        PX("after", e.use_count());
    }
    PX(ptr.use_count()); PX(ptr.unique()); PX(ptr.get());
    ptr.reset();
    PX(ptr.use_count()); PX(ptr.unique()); PX(ptr.get());
    for (auto &e : data) {
        PX("before", e.use_count());
        PX((e = nullptr).get());
        PX("after", e.use_count());
    }
}
```

- Predict (or explain) the output.

Testing `std::shared_ptr`

- There is the equivalent of a `nullptr`
  - Test by using `std::shared_ptr` in boolean context
- More possible tests:
  - `use_count()` – number of referrers
  - `unique` – whether there are other referrers

http://en.cppreference.com/w/cpp/memory/shared_ptr/use_count

http://en.cppreference.com/w/cpp/memory/shared_ptr/unique

http://en.cppreference.com/w/cpp/memory/shared_ptr/operator_bool

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): shared_pointer/demo.cpp

```
int main() {
    std::vector<std::shared_ptr<clazz>> data(3);
    {
        std::shared_ptr<clazz> ptr;             PX((bool)ptr);
        ptr = std::make_shared<clazz>();        PX((bool)ptr);
        auto pt2 = std::move(ptr);              PX((bool)pt2);
                                                PX((bool)ptr);
        PX(pt2.use_count()); PX(pt2.unique());
        for (auto &e : data) {
            PX((e = pt2).use_count());          PX((bool)pt2);
        }
        pt2 = nullptr;                          PX((bool)pt2);
        while (!data.empty()) {
        //  if (data.front().unique()) break;
            const auto &z = data.back();        PX((bool)z);
            PX(data.pop_back(), z.use_count());
        }
        PX(data.size());
    }
}
```

- What will happen if the commented-out line is enabled?
- What will change if z is not defined as reference?

Pointee Access via `std::shared_ptr`

- Usually with `operator*` and `operator->`
- `get()` may be used to interface with legacy code

http://en.cppreference.com/w/cpp/memory/shared_ptr/operator*

http://en.cppreference.com/w/cpp/memory/shared_ptr/get

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): shared_pointer/demo.cpp

```
int main() {
    std::vector<std::shared_ptr<clazz>> data;
    {
        auto ptr = std::make_shared<clazz>(5);
        PX(ptr->get_x());
        PX(ptr->set_x(2));
        PX(ptr->x);
        for (auto i = 1; i <= 3; ++i)  {
        //  data.push_back(ptr);
            data.push_back(std::make_shared<clazz>(*ptr));
            PX(data.back()->set_x(2*i+1));
            PX((*ptr).get_x());
        //  PX(ptr->get_x());
        }
    }
    auto pt2 = std::make_shared<double>(1.0/9.0);
    PX(*pt2);
}
```

- What will change if the commented-out `push_back` is enabled instead?
- Is there a difference between using `(*ptr).member` and `ptr->member`?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Custom Deleter for `std::shared_ptr`

- `std::unique_ptr` may have a custom deleter
  - Releasing memory with `delete` is the default
  - Custom deleter may increase memory footprint
- `std::allocate_shared` may be used instead of `std::make_shared`

http://en.cppreference.com/w/cpp/memory/shared_ptr/allocate_shared

Example (cont.): shared_pointer/demo.cpp

```
int main() {
    std::shared_ptr<std::FILE> out_fp{
        std::fopen("testfile.txt", "w"),
        [](FILE *fp) { if (fp) std::fclose(fp); }
    };
    PX(out_fp.get());
    if (out_fp.get()) {
        std::fputs("hello, world", out_fp.get());
//      std::fflush(out_fp.get());
    }
//  out_fp.reset();
    std::shared_ptr<std::FILE> in_fp{
        std::fopen("testfile.txt", "r"),
        [](FILE *fp) { if (fp) std::fclose(fp); }
    };
    PX(in_fp.get());
    if (in_fp.get()) {
        char s[80] = {'\0'};
        PX(std::fgets(s, sizeof s, in_fp.get()) ? s : "<eof>");
    }
}
```

- Explain the output. (Note that writing to a `FILE` is usually buffered!)
- What changes if any of the commented-out lines or both are enabled?

`std::shared_ptr` Pitfalls to Avoid

- Throwing operations during creation:
  - `new` may throw because out-of-memory
  - Constructor may throw for any reason
- Same native pointer used to initialise several `std::shared_ptr`
- Address of non-heap memory used to initialise `std::shared_ptr`
- Bad pairing with custom deleter
- Memory leaks via only "self-referential" object networks

Example (cont.): shared_pointer/demo.cpp

```
struct clazz {
    // ... as has been used in all the examples
    std::shared_ptr<clazz> get_self() {
        return std::shared_ptr<clazz>(this);
    }
    // ...
};

int main() {
    int x = 0xDEADBEEF;                         PX(&x);     PX(x);
//  auto ptr = std::shared_ptr<int>{&x};    PX(&*ptr);  PX(*ptr);
    auto p = new int{6*7};                      PX(p);      PX(*p);
    auto pt1 = std::shared_ptr<int>{p};     PX(&*pt1);  PX(*pt1);
//  auto pt2 = std::unique_ptr<int>{p};
//  auto pt3 = std::unique_ptr<int>{pt1.get()};
//  delete p;
    auto ptz = std::make_shared<clazz>();
//  auto pts = ptz->get_self();
}
```

- Explain why enabling any of the commented-out lines causes UB.
- Predict when and how the program might fail.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Sharing Existing Objects

- If a class needs to return a shared pointer to `this`
  - derive it from `std::enable_shared_from_this`
  - use `shared_from_this()` as return value
- **Never return `this` directly** (Understand why!)

http://en.cppreference.com/w/cpp/memory/enable_shared_from_this

http://en.cppreference.com/w/cpp/memory/enable_shared_from_this/shared_from_

Example: shared_from_this/demo.cpp

```
struct clazz : public std::enable_shared_from_this<clazz> {
//  std::shared_ptr<clazz> get_self() {return shared_from_this();}
    using std::enable_shared_from_this<clazz>::shared_from_this;
#define P(t)\
    std::cout << "\t=> my::clazz " t << this << std::endl;
    clazz()                 {P("default c'tor")}
    ~clazz()                {P("d'tor")}
#undef P
};

int main() {
    auto ptz = std::make_shared<clazz>();
    PX(ptz.get()); PX(ptz.use_count()); PX(ptz.unique());

//  auto pts = ptz->get_self();
    auto pts = ptz->shared_from_this();
    PX(pts.get()); PX(pts.use_count()); PX(pts.unique());
}
```

http://coliru.stacked-crooked.com/a/47eabec6a5458bfb

- Demonstrate the validity of above "cookbook-style" *shared from this*.
- Switch from delegating by `get_self` to inheriting `shared_from_this`.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Cyclic Referencing

- **Problem still not solved with *Smart Pointers*:**
  - Object may reference itself (directly or indirectly)
    - Class holds `std::shared_ptr` on its own type
    - Class holds `std::shared_ptr` …
      - … on other class which (somehow) …
      - … holds `std::shared_ptr` on original class
- May be avoided by change of design using `std::weak_ptr`
- C++ has no automatism to recognize inaccessible object chains!

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: cyclic_referencing/demo.cpp

```
template<typename T>
class link_elem {
public:
    using data_t = T;
    using next_t = std::shared_ptr<link_elem>;
private:
    data_t data_{};
    next_t next_{};
public:
    link_elem();
    ~link_elem();
    auto data() const {return data_;}
    auto data(const data_t &data) {return data_ = data;}
    auto next() const {return next;}
    auto next(const next_t &next) {return next_ = next;}
};
```

http://coliru.stacked-crooked.com/a/d3f1fc362e5ed6b1

- Explain why the above code has the **potential** for cyclic referencing.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: cyclic_referencing/demo.cpp

```cpp
int main() {
    using node_t = link_elem<int>;
    using node_ref = std::shared_ptr<node_t>;
    {
        std::vector<node_ref> chain;
        {
            node_ref last{};
            PX(node_t::instances);
            while (chain.size() < 1000) {
                chain.push_back(std::make_shared<node_t>());
                chain.back()->next(last);
                last = chain.back();
            }
            // how many `node_t` instances here?
        }
        // how many `node_t` instances here?
    }
    // how many `node_t` instances here?
}
```

- Predict how many `node_t` instances exist at the marked points.
- Instrument the code to verify the prediction.
- **So, is there a memory leak?** (Assuming main does not end.)

Optional Example (cont): cyclic_referencing/demo.cpp

```
class clazz {
    std::shared_ptr<clazz> next_{};
public:
    static std::size_t instances;
    clazz() {++instances;}
    ~clazz() {--instances;}
    void next(const std::shared_ptr<clazz> &next) {next_ = next;}
};
std::size_t clazz::instances{0};

int main() {
    std::vector<std::shared_ptr<my::clazz>> chain;
    PX(my::clazz::instances);
    while (chain.size() < 1000) {
        chain.push_back(std::make_shared<my::clazz>());
        chain.back()->next(chain.back());
//      chain.back()->next(chain.front());
    }
    PX(my::clazz::instances);
}
```

- How big is the memory leak?
- Will it vanish when the `back` links to the front (instead to itself)?
- Will it vanish if a `chain.clear()` is added after the loop?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.): cyclic_referencing/demo.cpp

```
struct airport;
struct flight;
using airport_ref = std::shared_ptr<airport>;
using flight_ref = std::shared_ptr<flight>;
struct airport {
    const std::string id;
    std::set<flight_ref> connections;
    // ...
};

class flight : std::enable_shared_from_this<flight> {
    const std::string id;
    std::vector<airport_ref> route;
    // ...
};
```

- Explain why the above code has a **potential** for cyclic references.
- (Complete and instrument the code to demonstrate a memory leak.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.): cyclic_referencing/demo.cpp

```
// Assuming completed code for the airport/flight example
// ...
struct airport {
    // ...
    airport(const std::string &id_) : id(id_) { ++instances; }
    void add_flight(flight_ref);
    void rm_flight(flight_ref);
    ~airport() { --instances; }
    static std::size_t instances;
};

struct flight : std::enable_shared_from_this<flight> {
    // ...
    flight(const std::string &id_) id id(id_) { ++instances; }
    ~flight() { --instances; }
    void set_route(std::initializer_list<airport_ref> airports);
    static std::size_t instances;
};

void airport::add_flight(flight_ref f) { connections.insert(f); }
void airport::rm_flight(flight_ref f) { connections.erase(f); }
```

- May memory leaks be fixed by strategic calls to `airport::rm_flight`?
- Could a solution be to place these in a loop in `flight::~flight()`?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.): cyclic_referencing/demo.cpp

```
int main() {
    {
        auto fra = std::make_shared<my::airport>("FRA");
        auto muc = std::make_shared<my::airport>("MUC");
        auto cpg = std::make_shared<my::airport>("CPG");
        auto mw000 = std::make_shared<my::flight>("MW000");
        auto mw001 = std::make_shared<my::flight>("MW001");
        mw001->set_route({fra});
        auto mw998 = std::make_shared<my::flight>("MW998");
        mw998->set_route({cpg, fra, muc});
        auto mw999 = std::make_shared<my::flight>("MW999");
        mw999->set_route({muc, fra, cpg});
        // <--- begin: desparately trying to fix memory leaks
        for (const auto& flight : {mw000, mw001, mw998, mw999}) {
            for (const auto& airport : flight->route)
                airport->rm_flight(flight);
        } // <--- end: desparately trying to fix memory leaks
    }
    PX(my::airport::instances);
    PX(my::flight::instances);
}
```

- Does the above addition finally fix the memory leaks?
- (In that light: are `std::shared_ptr`-s really "smart" pointers?)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Breaking Cycles with `std::weak_ptr`

- Memory leaks may occur despite use of `std::shared_ptr`

  For objects referencing their own class (type)

  - For objects mutually referencing each other

http://en.cppreference.com/w/cpp/memory/weak_ptr

Cyclic references causing memory leaks may be introduced via shared pointers in object networks that are designed "as if" there were garbage collector for clean-up.

- **This needs to be solved by a re-design!**

Example: weak_pointer/demo.cpp

```
struct airport;
struct flight;
using airport_ref = std::shared_ptr<airport>;
using flight_ref_strong = std::shared_ptr<flight>;
using flight_ref = std::weak_ptr<flight>;
struct airport {
    const std::string id;
    std::vector<flight_ref> connections;
    // ...
};

struct flight : std::enable_shared_from_this<flight> {
    const std::string id;
    std::vector<airport_ref> route;
    // ...
}
```

- Where is the **crucial** change of design that breaks the cycle.
- (Advanced: Why had the type of `connections` to be changed?)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Pointee Accessing via `std::weak_ptr`

- There is no direct access to pointed-to object
  - **First needs to obtain shared pointer from weak pointer**
  - Either via `std::shared_ptr` constructor
    - May throw if the weak pointer has lost the object
  - Or via `lock()` operation
    - May return shared pointer to "no object"
    - Therefore need to test prior access
    - If valid shared pointer holds pointee alive

http://en.cppreference.com/w/cpp/memory/weak_ptr/weak_ptr

http://en.cppreference.com/w/cpp/memory/weak_ptr/lock

Example (cont.): weak_pointer/demo.cpp

```
auto airport::clear_dead_connections() {
    const auto initial_connections = connections.size();
    auto endp = std::remove_if(connections.begin(),
                               connections.end(),
                               [](flight_ref f) {
                                   return !f.lock();
                               });
    connections.erase(endp, connections.end());
    return initial_connections - connections.size();
}

void airport::add_flight(flight_ref_strong fs) {
    (void) clear_dead_connections();
    auto known = [fs](flight_ref f) {
                  // return flight_ref_strong(f).get() == fs.get();
                     return f.lock() && f.lock().get() == fs.get();
                 };
    if (std::find_if(connections.begin(), connections.end(),
                     known) == connections.end())
        connections.push_back(fs);
}
```

- Where in the code above are `weak_ptr`-s turned into `shared_ptr`-s?
- What may be the better choice in the `known` lambda?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): weak_pointer/demo.cpp

```
std::ostream& operator<<(std::ostream& lhs,
                         const flight_ref_strong& rhs) {
    auto next = rhs->route.begin();
    if (next == rhs->route.end())
        return lhs << rhs->id << " (not operational)";
    lhs << (*next)->id;
    if (++next == rhs->route.end())
        return lhs << " => " << rhs->route.front()->id
                             << " (local flight)";
    while (next != rhs->route.end())
        lhs << " => " << (*next++)->id;
    return lhs;
}
```

- Are there any `weak_ptr`-s turned into `shared_ptr`-s in the above code?
- (So, is the code maximally robust?)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): weak_pointer/demo.cpp

```
std::ostream& operator<<(std::ostream& lhs,
                         const airport_ref& rhs) {
    if (!lhs) throw "std::logic_error"
    int dead_connections = rhs->clear_dead_connections();
    for (const auto &c : rhs->connections)
        lhs << flight_ref_strong(c)->id << "; ";
    if (dead_connections)
        lhs << "(" << dead_connections
            << " dead connection(s) removed)";
    return lhs;
}
```

- Are there any `weak_ptr`-s turned into `shared_ptr`-s in the above code?
- What else precautions are taken to protect against surprises?
- (So, is the code maximally robust?)

Garbage Collection ABI

- Garbage collection **is NOT**
  - part of C++11
  - nor C++14
  - nor C++1z
- There is **only an interface specified** …
  - … for "interested third parties"

http://en.cppreference.com/w/cpp/memory#Garbage_collector_support

---

For motivation see:

http://www.stroustrup.com/C++11FAQ.html#gc-abi

Date and Time

The contents of this library part falls into three main groups:

- Durations
- Time Points
- Clocks
- Classic Interface

http://en.cppreference.com/w/cpp/chrono

You may also be interested to read this:

http://www.informit.com/articles/article.aspx?p=1881386&seqNum=2

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: chrono_library/demo.cpp

To abbreviate the long namespace prefix without completely dropping it the following namespace alias is used in all examples:

```
#include <chrono>
namespace sc = std::chrono;
```

Also make sure you have a good type printer ready, especially if you want to experiment with chrono types to improve your understanding.

```
int main() {
    PT(sc::seconds);  PT(sc::minutes);  PT(sc::hours);

    const auto d1 = sc::seconds{12};    PT(decltype(d1));
    const auto& d2 = sc::hours{2};      PT(decltype(d2));
                                        PT(decltype(d2 - d1));
    auto t1 = sc::system_clock::now();  PT(decltype(t1));
    auto&& t2 = t1.time_since_epoch();  PT(decltype(t2));

    PT(decltype(t1 + sc::minutes{3}));
    PT(decltype(t2 + sc::minutes{3}));
}
```

http://coliru.stacked-crooked.com/a/2e3c17c27b91709b

Durations

- Durations are the first key abstraction of `std::chrono`
  - A duration carries its *resolution* as part of its type
  - Its base type is implementation defined
- There are convenient type aliases for the usual "time units"
- Operations result type is selected automatically …
  - … at compile time, not to loose accuracy
  - or else an `std::chrono::duration_cast` is necessary

http://en.cppreference.com/w/cpp/chrono#Duration

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: chrono_duration/demo.cpp

```
int main() {
    PT(sc::minutes);                 PT(sc::minutes::period);
    PX(sc::minutes::period::num);    PX(sc::minutes::period::den);
    PT(sc::minutes::rep);            PX(sc::minutes{}.count());
    PX(sc::minutes::min().count()); PX(sc::minutes::max().count());

    // continued on next page
    // ...
}
```

// http://coliru.stacked-crooked.com/a/5c6bc3bb243c5545

- Is the output what you would expect from the online documentation?
- Generate similar output for other standard durations.
- (Can you come up with particularly easy way to alter the type.)

Example (cont.): chrono_duration/demo.cpp

```
    // ...
    // continuing from previous page

//  PX(sc::minutes{1});
    PX(sc::minutes{1}.count());
    PX(sc::hours{123}.count());
    PX(sc::microseconds{-4}.count());
//  PX(sc::seconds{0.1}.count());
#if __cplusplus >= 201402L
    using namespace std::chrono_literals;
    PT(decltype(1min)); PX((1min).count());
    PT(decltype(123h)); PX((123h).count());
    PT(decltype(-4us)); PX((-4us).count());
    PT(decltype(0.1s)); PX((0.1s).count());
#endif
}
```

- Why do the commented-out lines do not compile?
- (Unrelated: what exactly means `__cplusplus >= 201402L`?)
- Why is `sc::seconds{0.1}` a *Compile Error* while `0.1s` works.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Examples (cont.): chrono_durations/demo.cpp

```
int main() {
    using dur1 = sc::duration<short, std::ratio<1, 35>>; PT(dur1);
    using dur2 = sc::duration<short, std::ratio<5, 28>>; PT(dur2);
    using dur3 = sc::duration<float, std::ratio<60>>;    PT(dur3);

    dur1 d1{22};    PT(decltype(d1));       PX(d1.count());
    dur2 d2{7};     PT(decltype(d2));       PX(d2.count());
    dur3 d3{1.5};   PT(decltype(d3));       PX(d3.count());

    {
        using dur = dur1;
        PT(dur);                PT(dur::period);
        PX(dur::period::num);   PX(dur::period::den);
        PT(dur::rep);
        PX(dur{}.count());      PX(dur::zero().count());
        PX(dur::min().count()); PX(dur::max().count());
    }
}
```

- Modify the above or add more examples for user-specified durations.
- E.g. show information for `dur2` or `dur3` instead of `dur1`.
- You may also copy the inner `{ … }`-block before you modify it … (in fact easy copy-pasting is the whole reason for having a block at all).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

```
int main() {
    sc::hours h{9};              PX(h.count());
    sc::minutes min{h/3};        PX(min.count());
    sc::seconds sec{min/30};     PX(sec.count());
    sc::milliseconds ms{sec};    PX(ms.count());

    sec = sc::duration_cast<sc::seconds>(ms);
    PT(decltype(sec));           PX(sec.count());
    sec = sc::duration_cast<sc::hours>(ms);
    PT(decltype(sec));           PX(sec.count());
    auto z = sc::duration_cast<sc::hours>(min);
    PT(decltype(z));             PX(z.count());

    using dur1 = sc::duration<short, std::ratio<1, 35>>; PT(dur1);
    using dur2 = sc::duration<short, std::ratio<5, 28>>; PT(dur2);
    using dur3 = sc::duration<float, std::ratio<60>>;    PT(dur3);
    dur1 d1{22};
    dur2 d2{7};     PT(decltype(d2-d1));    PX((d2-d1).count());
    dur3 d3{1.5};   PT(decltype(d3-d1));    PX((d3-d1).count());
    d1 = dur1{1};
    d2 = dur2{1};   PT(decltype(d2-d1));    PX((d2-d1).count());
}
```

- Add more examples to test your understanding for the rules.

```
int main() {
    sc::hours h{9};             PX(h.count());
    sc::minutes min{h};         PX(min.count());
    sc::seconds sec{h};         PX(sec.count());
    sc::milliseconds ms{h};     PX(ms.count());

    sec = sc::duration_cast<sc::seconds>(ms);
    PT(decltype(sec));          PX(sec.count());
    sec = sc::duration_cast<sc::hours>(ms);
    PT(decltype(sec));          PX(sec.count());
    auto z = sc::duration_cast<sc::hours>(min);
    PT(decltype(z));            PX(z.count());

    using dur1 = sc::duration<short, std::ratio<1, 35>>;
    using dur2 = sc::duration<short, std::ratio<5, 28>>;
    using dur3 = sc::duration<float, std::ratio<60>>;
    dur1 d1{100};                           PX(d1.count());
//  dur2 d2{d1};                            PX(d2.count());
    dur2 d2{sc::duration_cast<dur2>(d1)};   PX(d2.count());
//  auto d2 = sc::duration_cast<dur2>(d1);  PX(d2.count());
    auto d3 = sc::duration_cast<dur3>(d1);  PX(d3.count());
}
```

- Be sure to understand when `durations_cast`-s are **strictly** required.

```
int main() {
    using namespace std::chrono;
    const auto z = hours(1)
                 + minutes(14)
              // + minutes(100)
                 + seconds(59)
                 + milliseconds(500)
          + nanoseconds(12)
                 ;
    PT(decltype(z));
    PX(duration_cast<hours>(z).count());
    PX(duration_cast<minutes>(z % hours{1}).count());
    PX(duration_cast<seconds>(z % minutes{1}).count());
    PX(duration_cast<milliseconds>(z % seconds{1}).count());
    PX(duration_cast<nanoseconds>(z % seconds{1}).count());
}
```

- Explain the out put of the above code.
- Can you extract a clue from it how to implement "duration printer"?
- (Or a `duration_to_string` function?)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

```
template<typename R, typename T>
namespace my {
    std::string dur2(const T&t) {
        using namespace std;
        using namespace std::chrono;
        using dur = duration<long double, typename R::period>;
        return to_string(duration_cast<dur>(t).count());
    }
}

int main() {
    PX(my::dur2<sc::hours>(z));
    PX(my::dur2<sc::seconds>(z));
    PX(my::dur2<sc::milliseconds>(z));
}
```

- Advanced: Review the above code (and explain its output).

Clocks

- Basically similar to a duration …
  - … with special meaning for the value 0 – the *Epoch*
- Access is mainly via the static member `::now()`
- There are at least three clocks since C++11
  - `std::system_clock` ("wall clock" - may be adjusted)
  - `std::steady_clock` (monotonically increasing values)
  - `std::high_resolution_clock` ("fastest running" clock)
- Typically **but not necessarily** all three are different

http://en.cppreference.com/w/cpp/chrono#Clocks

Boost.Chrono supplies more clocks, also some tied to CPU usage (separating user mode from system mode).

http://www.boost.org/doc/libs/release/doc/html/chrono.html

Example: chrono_clocks/demo.cpp`

```
int main() {
    // the current time point ...
    auto t1 = sc::system_clock::now();

    // ... same time tomorrow
    const auto t1_tomorrow = t1 + sc::hours(24);
//  const auto t1_tomorrow = t1 + 24h;

    // ... let some time pass
    const auto t2 = sc::system_clock::now();
    const auto t_delta = t2-t1;

//  PX(t1);
    PX(my::days_since_epoch(t2));
}
```

http://coliru.stacked-crooked.com/a/74ddb26044bb44e9

- Review "cookbook style" use of clocks with `auto` variables.
- Is there an output operator for values returned from `…::now()`?
- Or might it have a `….count()` member?
- (A possible implementation of `my::days_since_epoch` follows.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): chrono_clocks/demo.cpp`

```
namespace my {
    template<typename T>
    float days_since_epoch(T t) {
        using seconds_per_day = std::ratio<24*60*60>;
        using float_days = sc::duration<float, seconds_per_day>;
        const auto tse = t.time_since_epoch();
        const auto dse = sc::duration_cast<float_days>(tse);
        return dse.count();
        // or - for better "job security" :-) - how about that:
        // return std::chrono::duration_cast<
        //               std::chrono::duration<float, std::ratio<
        //               86400, 1>>>(t.time_since_epoch()).count();
    }
}
```

- Try a review of the above by identifying "unknown parts" of C++.
- (Advanced: Parametrize the currently hard-coded type `float_days`.)

Much of the above will get clearer once you have some experience with *Template Meta-Programming*. Also understand that only very few people need to write or maintain code like above. So for most "`chrono`-users" holds: "*You are not expected to understand this.*" :-)

The Wall Clock (`std::system_clock`)

- Typically used when interfacing to users
  - Conversions to and from calendar dates
    - gives expected results
    - may not exactly predictable over a long time
- Time deltas might be negative (if clock has been adjusted)

http://en.cppreference.com/w/cpp/chrono/system_clock

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): chrono_clock/demo.cpp

```
int main() {
    PT(sc::system_clock);
    PT(sc::system_clock::rep);
    PT(sc::system_clock::duration);
    PT(sc::system_clock::period);
    PX(sc::system_clock::period::num);
    PX(sc::system_clock::period::den);

    const auto t = sc::system_clock::now();
    PT(decltype(t));
//  PX(t);
//  PX(t.count());
    const auto d = sc::system_clock::time_point{}.time_since_epoch();
    PT(decltype(d));
//  PX(d);
    PX(d.count());
}
```

- How to make the first lines "reusable" for the other (standard) clocks?
- What is the type of `t` and what the type of `d`?
- Understand that neither `t` nor `d` can be printed directly.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

The Monotonic Clock (`std::steady_clock`)

- Typically used to measure time deltas
  - Duration delta of accessing `::now()` consecutively …
    - … are never expected to be negative
    - (as long as tick counter does not overflow)
    - Epoch may change with next system boot time

http://en.cppreference.com/w/cpp/chrono/steady_clock

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): chrono_clock/demo.cpp

```
// generates deeply recursive call tree even for small arguments
// (therefore good for generating heavy CPU-load in user mode);
// see also: https://en.wikipedia.org/wiki/Ackermann_function
//
int ackermann(int m, int n) {
    return (m == 0) ? n+1
                    : (n == 0) ? ackermann(m-1, 1)
                               : ackermann(m-1, ackermann(m, n-1));
    }
}

int main() {
    auto t1 = sc::steady_clock::now();
    PX(ackermann(3, 4)); // <-- increase 2nd arg only for more load
    auto t2 = sc::steady_clock::now();
    PX((t2 - t1).count());
}
```

- Explain how the code above measures time.
- Loosely related: what time is measured here – CPU, process, real …?
- How to find out in which (time) units the measurement is taken?
- Generalize the measurement code into reusable function.
- (Hint: supply the *code to time* via a *callable* as argument.)

The Fastest Ticking Clock (`std::high_resolution_clock`)

- Typically used to measure **very short** time deltas
  - Difference of two consecutive access to `::now()` …
    - … might be negative in rare cases?
    - … because the tick counter did overflow??

http://en.cppreference.com/w/cpp/chrono/high_resolution_clock

---

To train the brain two quick "back of the envelope" calculations:

- A counter overflows a `signed long` of 32 bit (31/3.1 ~ 10 digits)
  - at 1 GHz (nsec resolution) in ~ (1e10 ÷ 1e9) ~ 10 seconds
  - at 100 MHz in ~ 100 sec (less than two minutes)
  - at 10 MHz in ~ 1000 sec (or a quarter of an hour)
  - at 1 MHz in (µsec resolution) **a few hours** (1 day = 86.400 sec)

- Counting picoseconds (1e-12) in 64 bit unsigned (64/3.1 ~ 20 digits)
  - overflows after ~ 1e8 (1e20 ÷ 1e12) seconds, or >3 years
  - (1 year = 3600 × 24 × 365 [seconds × hours × days per year])
  - (3.6e3 × 0.36e3 × 24 = ~ 1e6 × 1,25 × 24 = ~ 3e7)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Time Points

- Time points are the base abstraction for `std::chrono`-s clocks
  - Basically it is a duration coupled to a clock …
    - … where the latter determines the "epoch"
    - (i.e. the real-world meaning of the time point "zero")
- The type is seldom used directly
  - More convenient is to store clock values in `auto` variables
- If necessary, information on clock details can be retrieved from it
- **Between durations, time points, and plain numbers …**
  - **… only meaningful arithmetic operations are supported** …
  - **… everything else is turned into *Compile Errors* !**

http://en.cppreference.com/w/cpp/chrono#Time_point

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: chrono_timepoints/demo.cpp

```
int main() {                                // additive ------------------
    PT(decltype(  my::lhs_duration{}   + my::rhs_duration{}     ));
    PT(decltype(  my::lhs_duration{}   + my::rhs_time_point{}   ));
    PT(decltype(  my::lhs_time_point{} + my::rhs_duration{}     ));
//  PT(decltype(  my::lhs_time_point{} + my::rhs_time_point{}   ));
    PT(decltype(  my::lhs_duration{}   - my::rhs_duration{}     ));
    // ...
                                            // multiplicative ------------
    PT(decltype(  my::lhs_duration{}   * int{}                  ));
    // ...
    PT(decltype(  my::lhs_duration{}   * float{}                ));
    PT(decltype(  int{}                * my::rhs_duration{}     ));
    // ...
    PT(decltype(  float{}              * my::rhs_duration{}     ));
    PT(decltype(  my::lhs_duration{}   / my::rhs_duration{}     ));
    // ...
    PT(decltype(  my::lhs_duration{}   % int{}                  ));
    PT(decltype(  my::lhs_duration{}   % my::rhs_duration{}     ));
                                               // mixed ------------------
    PT(decltype(  int{} * sc::seconds{} / 3.5                   ));
    // ...
}
```

- Complete omitted parts (indicated by `// ...`).

Classic Interface (`std::time_t` / `std::clock_t`)

- An interface between `std::chrono` and `std::time_t` is available
  - via `std::chrono::system_clock::to_time_t`
  - and `std::chrono::system_clock::from_time_t`
- An interface between `std::chrono` and `std::clock_t` …
  - … might be built based on `CLOCKS_PER_SECOND`

http://en.cppreference.com/w/cpp/chrono/system_clock/to_time_t

http://en.cppreference.com/w/cpp/chrono/system_clock/from_time_t

http://en.cppreference.com/w/c/chrono/clock_t

http://en.cppreference.com/w/cpp/chrono/c/CLOCKS_PER_SEC

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: chrono_to_classic/demo.cpp

```
int main() {
    auto tp_chrono = sc::system_clock::now();
    PT(decltype(tp_chrono));
    PX(tp_chrono.time_since_epoch().count());

    auto tp_cstyle = sc::system_clock::to_time_t(tp_chrono);
    PT(decltype(tp_cstyle));
    PX(tp_cstyle);

    auto p_struct_tm = std::localtime(&tp_cstyle); PT(p_struct_tm);
    char buff[100];
    PX(std::strftime(buff, sizeof buff, "%c", p_struct_tm), buff);
    PX(p_struct_tm->tm_isdst);

    PX(tp_cstyle += (26*7) * (24*60*60));
    p_struct_tm = std::localtime(&tp_cstyle);
    PX(std::strftime(buff, sizeof buff, "%c", p_struct_tm), buff);
    PX(p_struct_tm->tm_isdst);

    tp_chrono = sc::system_clock::from_time_t(tp_cstyle);
    PX(tp_chrono.time_since_epoch().count());
}
```

- Explain (and add to or modify) the code to test your understanding.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: chrono_to_classic/demo.cpp

```
// (assuming ackermann function from earlier example)

int main() {
    PT(std::clock_t);
    std::clock_t t1 = std::clock(); PX(t1);
    std::clock_t t2 = t1;           PX(t2);
    PX(my::ackermann(3, 8)); // <-- generate some CPU load
    PX(t2 = std::clock());
//  PT(decltype(t2-t1));          PX(t2-t1);
    PT(decltype(CLOCKS_PER_SEC)); PX(CLOCKS_PER_SEC);
    PX("in msec =", 1e3*(t2-t1) / CLOCKS_PER_SEC);

    using clock_ratio = std::ratio<1, CLOCKS_PER_SEC>;
    using clock_duration = sc::duration<std::clock_t, clock_ratio>;
    clock_duration delta{t2-t1};
    using float_usec = sc::duration<float>;
    PX(sc::duration_cast<float_usec>(delta).count());
}
```

- Identify conversions *from* c-style (classic) `std::clock_t`.
- Identify conversions *to* c-style (classic) `std::clock_t`.
- Explain! (Or ask to get them explained.)
- Loosely related: what time is measured with `std::clock()`?

Regular Expressions

- Regular expression are supported since C++11
  - The standard mainly adopted what evolved as Boost.Regex
  - Who is familiar with regular expression will have a quick start
- This is **not** a general introduction to Regular Expressions
  - Therefore the regular expression syntax is not further explained
  - The content of this section is limited to some very basic examples

http://en.cppreference.com/w/cpp/regex

For quite a time the Linux distribution of `g++` was rolled out with a defective implementation of the regular expression library.

Usually it is easy to switch to Boost.Regex by simply changing the namespace (e.g. by means of an alias).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: regular_expressions/demo.cpp

```
bool euro_parser(const::std::string &str, double& ec) {
    using namespace std;
    regex re{"([1-9][0-9]*),([0-9][0-9])"};
//  regex re{"([1-9][0-9]{0,2}(?:[.][0-9]{3})*),([0-9]{2})"};
    smatch m;
    if (!regex_match(str, m, re))
        return false;
    const std::string es{m[1].str()};
    ec = std::stod(regex_replace(es, regex{"\\."}, ""))
       + std::stod(m[2].str())/100.0;
    return true;
}

int main() {
    double euro_cent;
    PX(euro_parser("1,89", euro_cent));         PX(euro_cent);
    PX(euro_parser("123,89", euro_cent));       PX(euro_cent);
    PX(euro_parser("1234567,89", euro_cent));   PX(euro_cent);
    PX(euro_parser("1.234.567,89", euro_cent)); PX(euro_cent);
}
```

http://coliru.stacked-crooked.com/a/749f61cf01481d90

- What changes when switching to the currently commented-out RE?

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: regular_expressions/demo.cpp

```
void int main() {
    const std::string re{
        "\\s*([^;]*);\\s*([^;]*);\\s*([^;]*);?.*"
    };
    PX(re);
    const std::string bulk_letter{R"(
       Dear $1 $2 in $3,
       today is your lucky day...
    )"};
    PX(bulk_letter);
    const std::vector<std::string> address_list{
        "Mr.;  Santa Claus;  North Pole; Important Client!!",
        "Mrs.; Daisy Duck;   Duckburg;   Beauty of the town",
        "Mr.;  Mickey Mouse; Duckburg;  no comment",
        "Mr.;  Snoopy; Charlie Brown's Backyard",
    };
    for (const auto &addr : address_list) {
        PX(addr);
        PX(std::regex_replace(addr, std::regex{re}, bulk_letter));
    }
}
```

- Simply enjoy the example – and ask all the questions you may have!

Random Numbers

- Since C++11 random number generation has to main aspects:
  - The generator producing *randomness* as such
  - The distribution *drawing* from a well-defined set of values
- Both are rather specialist areas
  - Therefore they are not covered in any depth
  - Some few examples are given to demonstrate basic usage …
  - … applicable in "cookbook-style" to non-demanding tasks

http://en.cppreference.com/w/cpp/numeric/random

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

C-Style Random Numbers (`std::rand`)

- In the C89 standard a random generator was specified
  - It used a simple algorithm to produce pseudo-randomness
  - For "true" randomness it could be "seeded"
  - Conforming implementations
    - had to support a minimum range of 0 … 32767
    - could repeated the sequence after 65534 iterations

http://en.cppreference.com/w/cpp/numeric/random/rand

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: random_numbers/demo.cpp

```
int main() {
    PT(decltype(rand()));
    PX(RAND_MAX);
//  enable line below for "true" randomness or
//  generated sequence is identical on each run
//  std::srand(std::time(nullptr));
    for (int i = 0; i < 10; ++i)
        PX(std::rand() % 6 + 1);
}
```

- Why might the distribution of 1 … 6 be not quite uniform here?
- (Which values are probably slightly preferred - low or high end?)

http://coliru.stacked-crooked.com/a/b93e31046acda10f

Random Generators

- The standard specifies a variety of random generators
  - All are "pseudo random" in so far that they start repeating …
  - … but this may not happen before the world ends
- Generators are defined as classes
  - They may be seeded in via a constructor argument …
  - … which may be a real random source (system dependant)
- Random values are retrieved via `operator()` with no arguments

http://en.cppreference.com/w/cpp/numeric/random#Random_number_engines

---

The frequently used `mt19937` generator uses an algorithm that starts repeating after 2^{19937} calls – so the chance of "predicting" what will come next by watching one full run and taking notes exists only in theory.

Generators may be used stand alone if a uniform distribution over a range that is a power of two is required (just pick some bits).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): random_numbers/demo.cpp

```
int main() {
    PT(std::mt19937::result_type);
    PX(std::mt19937::min());
    PX(std::mt19937::max());
    PT(std::mt19937_64::result_type);
    PX(std::mt19937_64::min());
    PX(std::mt19937_64::max());
    std::mt19937 gen;
    for (int n = 0; n < 10; ++n)
        PX(gen());
}
```

- Explain the output of the above code.

Loosely related: implement a "randomness checker" based on the fact that a high quality random distribution over the range 0 … 2^{N-1} must have:

- a 50:50 chance for any bit to be set or not set
- a 50:50 chance for any bit to keep its previous state
- a 50:50 chance for any two bits to be in the same state

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Random Distributions

- There are many choices available
  - Most may be of interest for experts on statistics only
  - Pure mortals will almost always use `std::uniform_distribution`
    - It is class constructed with two arguments:
      - the minimum and maximum value (inclusive)
      - **not** one beyond the maximum – **STL users be aware**
  - (Other distributions will typically require other arguments)

http://en.cppreference.com/w/cpp/numeric/random#Random_number_distribution

Example (cont.): random_numbers/demo.cpp

```
void random_distribution() {
#if 1
    // start randomly on each run
    std::random_device rd;
    std::mt19937 gen(rd());
#else
    // same sequence on each run
    std::mt19937 gen;
#endif
    std::uniform_int_distribution<> dice(1, 6);

    for (int n = 0; n < 10; ++n)
        PX(dice(gen));
}
```

- Explain the code above.

Loosely related: implement a "runlength checker" based on the fact that a high quality random sequence has typical values for all run-lengths (i.e. how often the same value occurs in a row) which depends **only** on the range of values (more exactly *high-low+1*).

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

STL-Extensions

- STL container and algorithms …
  - … **are covered in overview style only** …
  - … focusing on additions by C++11, C++14, and C++1z
- Also included here are additions to …
  - … the `std::string` class itself and
  - … convenience functions related to it.

http://en.cppreference.com/w/cpp/container

http://en.cppreference.com/w/cpp/algorithm

http://en.cppreference.com/w/cpp/string

Since the STL has been designed by HP more than 20 years ago, you will find lots of books and articles elsewhere, covering the topic at any level.

Optional Example: stl_container/demo.cpp

```
int main() {
    std::map<std::string, std::vector<int>> words;
    using isit = std::istream_iterator<std::string>;
    int pos{0};
    std::for_each(isit{std::cin}, isit{},
                  [&](const std::string &s) {
                      words[s].push_back(++pos);
                  });
    for (const auto &e : words) {
        std::cout << e.first << ':';
        for (const auto &w : e.second)
            std::cout << ' ' << w;
        std::cout << "; ";
    }
    std::cout << std::endl;
}
```

http://coliru.stacked-crooked.com/a/997f8aa290590323

- As "warm-up" you may want to analyze the above code example.
- Test Input:
  `Wenn hinter Fliegen Fliegen fliegen fliegen Fliegen Fliegen nach`
- Generated Output:
  `Fliegen: 3 4 7 8; Wenn: 1; fliegen: 5 6; hinter: 2; nach: 9;`

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

```cpp
int main() {
    std::set<std::string> words;
    using word_ref = decltype(words)::const_iterator;
    std::map<int, word_ref> positions;
    std::string groupstring;
    while (std::getline(std::cin, groupstring, ';')) {
        std::istringstream groupstream{groupstring};
        std::string word;
        groupstream >> std::ws;
        if (std::getline(groupstream, word, ':')) {
            auto w = words.insert(word).first;
            int p;
            while (groupstream >> p)
                positions.insert(std::make_pair(p, w));
        }
    }
    std::ostringstream text_line;
    for (const auto &e : positions) {
        std::cout << *e.second << ' ';
    }
    std::cout << std::endl;
}
```

- Explain how this program reverses the effect of the previous one.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

New Standard Container: `std::array`

- Drop-in replacement for native arrays
  - Same performance, same memory footprint
  - Does not decay to pointer to first element
  - Bounds-checked member access with `at()`
- **But**: size not automatically determined from initializer count

http://en.cppreference.com/w/cpp/container/array

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

New Standard Container: `std::forward_list`

- Singly linked list with minimal memory overhead:
  - Just one pointer (required) to access the list head
  - Just one extra pointer (required) per element
- Consequence: changed semantics for some container operations
  - `insert_after` (instead of `insert`)
  - `emplace_after` (instead of `emplace`)
  - `erase_after` (instead of `erase`)
  - no `size()`, only `empty()`

http://en.cppreference.com/w/cpp/container/forward_list

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Augmented Container Interface

- Two main reasons for augmenting the interface in C++11 were:
  - Provide `const` versions of iterators for `auto` variables
  - Support *move only* types as container elements
- Other additions fixed minor nuisances, like
  - Accessing a map element without automatic insertion

Beyond that there were some changes in the specification of some STL templates to allow better code generation and more reuse. Such changes were hardly visible from the client side, i.e. existing C++98 code would be affected only in very rare cases.

New Ways to Obtain Iterators

- `cbegin()`, `crbegin()`,cend()`, and`crend()`
    - Get non-modifiable iterator for modifiable container
    - Available as members and global functions

http://en.cppreference.com/w/cpp/iterator#Range_access

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

More Efficient Insertion of New Elements

- `emplace`-Variants to add elements
  - Forward arguments to element constructor
  - May therefore avoid a temporary and copying
  - **Works with *move only* types

http://en.cppreference.com/mwiki/index.php?title=Special%3ASearch&search=emplace

Example (cont.): stl_containers/demo.cpp

```
namespace my {
    struct clazz {
        clazz(int, std::string) {}
        clazz(const clazz&) =default;
        clazz(clazz&&)      =delete;
    };
}

int main() {
     std::vector<my::clazz> v;
//   v.push_back(my::clazz(42, "hi!"));
     v.emplace_back(42, "hi!");
}
```

- Enable and disable both, `push_back` and `emplace_back`.
- Try all possible combinations of `=default` and `=delete`.
- Demonstrate that "emplacing" works with *move only* types.

Note: `g++-4.7` bails out with an internal error if you delete both, copy **and** move constructor of `my::clazz`.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Hash-Based Containers

- Much like tree-based counterparts
  - Lookup O(1) instead of O($\log_2$(N))
  - (also when taking place as part of other operations)
- No sorting relation for element type
- Instead a hash function is required
  - already defined for basic types
  - also STL container classes

http://en.cppreference.com/w/cpp/container/unordered_set

http://en.cppreference.com/w/cpp/container/unordered_map

Example (cont.): `stl_containers/demo.cpp`

Given the two example programs used for warm-up:

```
int main() {
    std::map<std::string, std::vector<int>> words;
    using isit = std::istream_iterator<std::string>;
    int pos{0};
    std::for_each(isit{std::cin}, isit{},
                  [&](const std::string &s) {
                      words[s].push_back(++pos);
                  });
    // ...
}

int main() {
    std::set<std::string> words;
    using word_ref = decltype(words)::const_iterator;
    std::map<int, word_ref> positions;
    // ...
}
```

- Which of the associative containers may be `unordered_` too?
  - `std::map<std::string, std::vector<int>> words;` (in first)
  - `std::set<std::string> words;` (in second)
  - `std::map<int, word_ref> positions;` (in second)

Augmented Map Interface

- `at()`
  - Returns element by reference only if it exists
  - **Otherwise throws `std::out_of_range`**
- `try_emplace()`
  - Overwrites value only if key exists
  - **Otherwise does nothing**

http://en.cppreference.com/w/cpp/container/map/at

http://en.cppreference.com/w/cpp/container/unordered_map/at

http://en.cppreference.com/w/cpp/container/map/try_emplace

http://en.cppreference.com/w/cpp/container/unordered_map/try_emplace

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

New Algorithms

- Some new algorithms were added, mostly from Boost.Algorithm
- Some express already supported functionality more clearly, like
  - `std::copy_if` (copy container elements with given predicate)
  - which can be done in C++98 with `std::remove_copy_if`
- In the same vein some additions were seemingly redundant, like
  - `std::all_of`, `std::any_of`, and `std::none_of` …
  - … any of which can replace the two others

There are still algorithms available via the Boost Platform that come in handy at times but are not (yet) part of the C++ standard. See also:

http://www.boost.org/doc/libs/release/libs/algorithm/doc/html/index.html

http://www.boost.org/doc/libs/release/libs/range/doc/html/index.html

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example: stl_algorithms/demo.cpp

```
int main() {
    auto test_input = {3, 7, 5, 2, 9, 25, 8, 17, 3};
    std::vector<int> multiples_removed;
    auto is_divisible = [](int v, int n) {return (v%n == 0);};
    std::set<int> divisors;
    std::copy_if(test_input.cbegin(), test_input.cend()),
              std::back_inserter(multiples_removed),
              [is_divisible, &divisors](int v) -> bool {
                  auto v_is_div = [v](int n) {return (v%n == 0);};
                  const auto &cdivs = divisors;
                  if (std::find_if(cdivs.begin(), cdivs.end(),
                                   v_is_divi) != cdivs.cend())
                      return false;
                  divisors.insert(v);
                  return true;
              });
}
```

http://coliru.stacked-crooked.com/a/ee8ae6898261b662

- What is finally contained in `multiples_removed`?
- (You may guess from the name – or review the code.)
- Rewrite the code to use `std::remove_copy_if` from C++98.
- Unrelated: implement `v_is_div` with `std::bind` (instead of a lambda).

Optional Example (cont.): stl_algorithms/demo.cpp

```
int main() {
    auto data = {3, 7, 5, 2, 9, 25, 8, 17, 3};

    auto divides_25 = [](int n) {return (25%n == 0);};
    auto not_divides_25 = [](int n) {return (25%n != 0);};

    PX(std::any_of(data.cbegin(), data.cend(), divides_25));

    PX(std::any_of(data.cbegin(), data.cend(), not_divides_25));
}
```

- Predict the results of running the code above.
- Remove the value `25` from `data` and repeat.
- How to express the same tests with `std::all_of` (and negation)?
- How to express the same tests with `std::none_of`?
- Insert the lambda-s directly into the calls that make use of it.
- Loosely related:
  - reuse `is_divisible` (from last page) to implement `divides_25`
  - reuse `divides_25` to implement `not_divided_25`
  - (Advanced: replace all lambdas with `std::bind`.)

Tuples

- Tuples are collection of elements
  - Element count is fixed at compile time
  - Element types may differ from each other
- Avoids to define utility structures
- Generalisation of `std::pair`

http://en.cppreference.com/w/cpp/utility/tuple

Example: tuple_demo.cpp

```
int main() {
    std::tuple<char, float, std::string> a{'z', 1/.9f, "hi!"};
    PX(std::get<0>(a));  PT(decltype(std::get<0>(a)));
    PX(std::get<1>(a));  PT(decltype(std::get<1>(a)));
    PX(std::get<2>(a));  PT(decltype(std::get<2>(a)));

    auto b = std::make_tuple('?', 0.0, "hello");
    PX(std::get<0>(b));  PT(decltype(std::get<0>(b)));
    PX(std::get<1>(b));  PT(decltype(std::get<1>(b)));
    PX(std::get<2>(b));  PT(decltype(std::get<2>(b)));

    a = b;    PX(std::get<0>(a)); PX(--std::get<0>(a));
              PX(std::get<1>(a)); PX(std::get<1>(a) = 0.0);
              PX(std::get<2>(a)); PX(&(std::get<2>(a)[0] = 'H'));

    char c; double d; float f; std::string e;
//  std::tie(c, d, e) = a;              PX(c); PX(d), PX(e);
    std::tie(c, f, e) = a;              PX(c); PX(f), PX(e);
    std::tie(std::ignore, d, e) = b;    PX(c); PX(d); PX(e);
}
```

http://coliru.stacked-crooked.com/a/15a295275c123079

- Explore the code and add your own!

Creating a Tuple

- Tuples can be created in various ways:
  - Constructor call (slightly inconvenient)
  - `std::make_tuple` (constructor arguments only)
  - `std::tuple_cat` (from existing tuples)

http://en.cppreference.com/w/cpp/utility/tuple/make_tuple

http://en.cppreference.com/w/cpp/utility/tuple/tuple_cat

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Accessing Tuple Elements

- Tuples elements can be individually accessed via `std::get`
  - Element is identified by its index (0-based) …
  - … which **must be a compile time constant**
  - Alternative: element type (if unique, since C++14)

http://en.cppreference.com/w/cpp/utility/tuple/get

Example (cont.): tuple_basics/demo.cpp

```
int main() {
    std::tuple<char, float, std::string> a{'z', 1/.9f, "hi!"};
//  std::tuple<char, float, std::string> a{'z'};
    PX(std::get<0>(a));  PT(decltype(std::get<0>(a)));
    PX(std::get<1>(a));  PT(decltype(std::get<1>(a)));
    PX(std::get<2>(a));  PT(decltype(std::get<2>(a)));
//  for (i = 0; i < 3; ++i) PX(std::get<i>(a));

    auto b = std::make_tuple('?', 0.0, "hello");
    PX(std::get<0>(b));  PT(decltype(std::get<0>(b)));
    PX(std::get<1>(b));  PT(decltype(std::get<1>(b)));
    PX(std::get<2>(b));  PT(decltype(std::get<2>(b)));

    PX(++std::get<0>(b));
    PX(std::get<1>(b) = std::sqrt(2));
//  PX(std::get<2>(b)[0] = std::toupper(std::get<2>(b)[0]));
    PX(&(std::get<2>(a)[0] = std::toupper(std::get<2>(a)[0])));
}
```

- Why are the commented-out lines Compile Errors?
- Add some code demonstrating `std::tuple_cat`.

Tying Tuple Elements

- Convenient alternative to `std::get`
  - Especially for unpacking tuples returned from functions
  - `std::ignore` may be used for elements not of interest

http://en.cppreference.com/w/cpp/utility/tuple/tie

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): tuple_basics/demo.cpp

```
int main() {
    std::tuple<char, float, std::string> a('z', 1/.9f, "hi!");
    auto b = std::make_tuple('?', 0.0, "hello");

    char c; double d; float f; std::string e;
//  std::tie(c, d, e) = a;              PX(c); PX(d), PX(e);
    std::tie(c, f, e) = a;              PX(c); PX(f), PX(e);
    std::tie(std::ignore, d, e) = b;    PX(c); PX(d); PX(e);

    PX(f = 0.0); PX(std::get<1>(a));
    PX(std::get<1>(b) = 1e-1); PX(d);
}
```

- Why is the commented-out line a Compile Error?
- Continue to experiment with the code.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Compile Time Operations with Tuples

- Tuples also have Compile Time operations
  - Determine the size (number of elements)
  - Accessing the type of the i-th element
- Both are implemented in the usual Meta Programming style
  - The tuple is handed over as type of a template instantiation
  - A value result is accessed via `…::value`
  - A type result is access via `typename …::type`

http://en.cppreference.com/w/cpp/utility/tuple/tuple_size

http://en.cppreference.com/w/cpp/utility/tuple/tuple_element

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optional Example (cont.) tuple_basics/demo.cpp

```
int main() {
    using tt = std::tuple<char, float, std::string>;
    PX(std::tuple_size<tt>::value);
    PT(std::tuple_element<0, tt>::type);
    PT(std::tuple_element<1, tt>::type);
    PT(std::tuple_element<2, tt>::type);
#if __plusplus >= 201402L
    PX(std::tuple_size<tt>{});
    PX(std::tuple_element_t<0, tt>);
    PX(std::tuple_element_t<1, tt>);
    PX(std::tuple_element_t<2, tt>);
#endif

    auto b = std::make_tuple('?', 0.0, "hello");
    PX(std::tuple_size<decltype(b)>::value);
    PT(std::tuple_element<0, decltype(b)>::type);
    PT(std::tuple_element<1, decltype(b)>::type);
    PT(std::tuple_element<2, decltype(b)>::type);
#if __cplusplus >= 201402L
    // ... as before (mutatis mutandis)
#endif
}
```

- Feel free to extend the code following your own ideas.

Miscellaneous Tuple Features

- Tuples also support border cases:
  - Just one element (wouldn't require tuple)
  - No elements at all (similar to `void`)
- Access by (unique) element type (since C++14)
  - `std::get` also can access tuple elements by type
  - Often useful for small tuples and `std::pair`-s.

http://en.cppreference.com/w/cpp/utility/tuple/get

Completely empty tuples are also useful to terminate Compile-Time recursion when processing all elements of a tuple.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

```
int main() {
    std::tuple<bool> just_one;
    PX(std::tuple_size<decltype(just_one)>::value);
    PT(typename std::tuple_element<0, decltype(just_one)>::type);

//  std::tuple<void> none;

    std::tuple<> empty;
    PX(std::tuple_size<decltype(empty)>::value);

#if __cplusplus >= 201402L
    PX(std::tuple_size<decltype(just_one)>{});
    PT(std::tuple_element_t<0, decltype(just_one)>);
    PX(std::tuple_size<decltype(empty)>{});

    auto b = std::make_tuple('?', 0.0, "hello");
    PT(typename std::tuple_element<0, decltype(b)>::type);
    PX(std::get<char>(b));
    PX(std::get<0>(b));
    // ...
#endif
}
```

- (Still there? Go on your own, as usual.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Conversion from and to `std::pair`

- Tuples are (to some degree) compatible with `std::pair`
  - Of course this is only true for tuples with 2 elements

http://en.cppreference.com/w/cpp/utility/pair/get

Optional Example (cont.): tuple_basics/demo.cpp

```
int main() {
    auto x = std::make_tuple(42, true);
    PX(std::get<0>(x)); PX(std::get<1>(x));
//  PX(x.first); PX(x.second);

    auto y = std::pair<int, bool>{};
    PX(std::get<0>(y)); PX(std::get<1>(y));
    PX(y.first); PX(y.second);

    x = y; PX(std::get<0>(x)); PX(std::get<1>(x));

//  y = std::make_tuple(-1, true));
    std::tie(y.first, y.second) = std::make_tuple(-1, 1);
    PX(std::get<0>(y)); PX(std::get<1>(y));
    PX(y.first); PX(y.second);

#if __cplusplus >= 201402L
    std::set<std::string> words{"two"};
    PX(std::get<bool>(words.insert("one")));
    PX(std::get<bool>(words.insert("two")));
#endif
}
```

- Huh!! (Running out of ideas to play with the code? Hope not so!)

Conversion from and to `std::array`

- Tuples are (or can be made) compatible with `std::array`
  - Of course type conversions may need to take place

http://en.cppreference.com/w/cpp/container/array/get

http://en.cppreference.com/w/cpp/utility/integer_sequence#Example

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`std::string`-Related Additions

- Additions related to `std::string` were motivated by:
  - Providing easy conversions between arithmetic types, i.e.
    - **from** all integral types (base 10 only)
    - **to** all integral types (base 2 to 36)
    - **to** and **from** all floating point types
  - Closing the gap between `std::string`-s and `std::vector`-s, i.e.
    - `std::string` has (superset) of interface of `std::vector<char>`
    - same for `std::wstring` and `std::vector<wchar_t>`

http://en.cppreference.com/w/cpp/string

Example: string_related/demo.cpp

```
int main() {
    int i = 42;                 PX(i);
    float f{};                  PX(f);
    std::string s{"hello"};     PX(s);
    PX(s = std::to_string(i));
    PX(s += ".50");
    PX(f = std::stof(s));
    PX(s = std::to_string(f));
    PX(f = std::stoi(s));
    PX(s.erase(s.end()-3, s.end()), s);
    PX(s.resize(s.size()-1), s);
    PX(std::stoi(s));
    PX(std::stoi("0x" + s, nullptr, 0));
    PX(std::stoi(s, nullptr, 16));
}
```

http://coliru.stacked-crooked.com/a/78e0bd9b2829a48e

- Explore more cases, also possibly including grey areas.

Type Aliases and Conversion Helpers

- Type aliases are provided according to those for character types:
  - `std::u16string` with elements (code units) of type `char16_t`
  - `std::u32string` with elements (code units) of type `char32_t`
- Class template `std::codevect` provides a base for conversions
  - between UTF-8 and UCS2/UCS4 (`std::codecvt_utf8`)
  - between UTF-16 and UCS2/UCS4 (`std::codecvt_utf16`)
  - between UTF-8 and UTF-16 (`std::codecvt_utf8_utf16`)

http://en.cppreference.com/w/cpp/locale/wstring_convert

http://en.cppreference.com/w/cpp/locale/codecvt

`noexcept` (Specifier and Operator)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Optimising with `noexcept` Specifier

- Only when the goal is to reduce book-keeping code …
  - … add `noexcept` - to functions that actually cannot throw
  - or when there is no meaningful way to continue after failure
- **Diligently** write `catch`-blocks otherwise …
  - … maybe even `catch(...) {}` (if nothing can be done) …
  - … but care for channeling "out-of-band" failure indication

http://en.cppreference.com/w/cpp/language/noexcept_spec

Adding `noexcept` to functions that actually throw will terminate the program after calling `std::terminate`!

http://en.cppreference.com/w/cpp/error/set_terminate

Conditional `noexcept`

- Use `noexcept` as Compile-Time operation to indicate …
  - a generic function will not throws by itself
  - but may throw depending on its instantiation arguments

http://en.cppreference.com/w/cpp/language/noexcept

Example: noexcept_usage/demo.cpp

```
namespace my {
    struct clazz {
        void xthrow(bool b) const noexcept(false) { /* ... */ }
        void nthrow(bool b) const noexcept(true) { /* ... */ }
    };
    struct other {
        void nthrow(bool) const;
        void xthrow(bool) const;
    };

    void first (const clazz &c) noexcept/*(true)*/ { /* ... */ }

    void second(const clazz &c) noexcept(false) { /* ... */ }

    template<typename T>
    void third(const T &c) noexcept(
        noexcept(c.nthrow(true))
    ) { /* ... */ }
```

http://coliru.stacked-crooked.com/a/f7ed04fd0ccac524

- Identify usages of `noexcept` as specifier and as operator.
- (Follow the live example for more - then experiment on your own.)

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Testing for `noexcept`

- SFINAE based techniques (or `std::enable_if`) may be used …
  - … but usually not necessary to learn for C++ developers
  - … as most the common case is selecting between copy and move
- In the example only the core technique is presented
  - Based on template specialization for `bool`
  - More options come with *Template Meta Programming*

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): noexcept_usage/demo.cpp

```
template<bool B> struct use_either_or;

template<> struct use_either_or<true>
{static const char* call() {return "use this for true";}};

template<> struct use_either_or<false>
{static const char* call() {return "or that for false";}};

int main() {
   PX(use_either_or<true>::call());
   PX(use_either_or<false>::call());
   PX(use_either_or<noexcept(my::first(my::clazz{}))>::call());
   PX(use_either_or<noexcept(my::second(my::clazz{}))>::call());
   PX(use_either_or<noexcept(my::third(my::clazz{}))>::call());
   PX(use_either_or<noexcept(my::third(my::other{}))>::call());
}
```

- (Follow the live example for more - then experiment on your own.)

`noexcept`-Based Move/Copy Selection

- `std::move_is_noexcept` is a **conditional cast** with the effect to …
  - … select the move constructor or assignment for its argument …
    - … **only when it is `noexcept`**
    - … thereby giving the guarantee of a non-throwing operation
  - … select the copy constructor or assignment otherwise …
    - … which may or may not throw
- So the overall guarantee is
  - **not** that the operation will always succeed, but
  - only that its operand is untouched **if** it throws

http://en.cppreference.com/w/cpp/utility/move_if_noexcept

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example (cont.): noexcept_usage/demo.cpp

```
struct safe_move {
    const char *used;
    safe_move()                     {used = "default c'tor";}
    safe_move(const safe_move&)     {used = "copy c'tor";}
    safe_move(safe_move&&) noexcept {used = "move c'tor";}
};

void move_or_copy() {
    // Important: moving from the SAME object more than once is
    // ONLY OK here as the move operation leaves its operand
    // intact which is UNTYPICAL for real world applications.

    safe_move s; PX(s.used);
    safe_move s_must_move{std::move(s)};
    PX(s_must_move.used);
    safe_move s_may_copy{std::move_if_noexcept(s)};
    PX(s_may_copy.used);
}
```

- Demonstrate selection by making the move in `safe_move` "unsafe".
- Add examples for assignment.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Multi-Threading (Overview)

- Asynchronous Tasks
- Mutexes and Locks

Note: The purpose of this chapter is only to give an overview.

The topic gets a more detailed coverage in specialised trainings.

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Asynchronous Tasks

- Give any callable as argument to `std::async`
- Store the returned `std::future`
- At any time later call the `get()` member function
  - This will wait for the callable to return
  - Forwarding its result (if not `void`) …
  - … or an exception (if one were thrown)
- Very simple and straight forward …
  - … as long as asynchronous tasks work on independent data
  - **Access to shared resources must be serialized**

http://en.cppreference.com/w/cpp/thread/async

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Packaged Tasks

- Similar to asynchronous tasks
  - With a little bit more of explicit control …
  - … at the price of slightly more complicated use

http://en.cppreference.com/w/cpp/thread/packaged_task

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Promises and Futures

- Connect tasks in separate thread and caller
- Provides infra-structure to
  - Announce availability of result in callee
  - Wait for result in the caller
- Technically `std::async` returns an `std::future`
- Typically received via an `auto`-typed variable

http://en.cppreference.com/w/cpp/thread/promise

http://en.cppreference.com/w/cpp/thread/future

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Mutexes

- Mutexes are used to serialize access to shared resources
- Wrong use of mutexes may lead to
  - Deadlocks (two threads waiting on each other)
  - Sub-optimal performance (bottle-necks)
  - Corrupted Data (if race conditions occur)

http://en.cppreference.com/w/cpp/thread#Mutual_exclusion

Locks

- Locks are RAII-style wrappers to mutexes
  - Wrong use of locks has same problems as mutexes …
  - … except that lock release is a little less error prone
- There is also some support for deadlock avoidance
  - Several locks may be acquired atomically
  - Operation fails if not all are available

http://en.cppreference.com/w/cpp/thread#Generic_mutex_management

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Condition Variables

- Useful for consumer/producer synchronization
  - No busy waiting
  - No sub-optimal latency

http://en.cppreference.com/w/cpp/thread/condition_variable

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Lock Free Algorithms

- Lock Free algorithms use special atomic operations

http://en.cppreference.com/w/cpp/atomic/atomic_compare_exchange

Direct Use of Threads

- Simply hand-over callable to `std::thread`-constructor
- But be aware that threads
  - either must be joined
  - or detached
- **Otherwise thread object destructor terminates program**

http://en.cppreference.com/w/cpp/thread/thread

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: multi_threading/demo.cpp

– will be given live on request –

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

`alignas` and `alignas`

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: aligned_memory/demo.cpp

– will be given live on request –

Experimental Features

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

File System Access

http://en.cppreference.com/w/cpp/experimental/fs

Example: experimental_filesystem/demo.cpp

– will be given live on request –

Parallelism

http://en.cppreference.com/w/cpp/experimental/parallelism

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Other Library Extensions

http://en.cppreference.com/w/cpp/experimental/lib_extensions

`std::experimental::optional`

- Optional data on stack
  - avoids heap allocation
  - may "waste" memory if unused
- Alternative: Boost.Optional

http://en.cppreference.com/w/cpp/experimental/optional

Example: experimental_optional/demo.cpp

– will be given live on request –

`std::experimental::any`

- Most general type erasure
- Alternative: Boost.Any

http://en.cppreference.com/w/cpp/experimental/any

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH

Example: experimental_any/demo.cpp

– will be given live on request –

`std::experimental::string_view`

- Non-owning shared strings
    - Avoids hidden COW
    - Highly efficient …
    - … but may dangle

http://en.cppreference.com/w/cpp/experimental/basic_string_view

Example: experimental_stringview/demo.cpp

– will be given live on request –

(CC) BY-SA: Dipl.-Ing. Martin Weitzel für MicroConsult Training & Consulting GmbH