Computer Science

Even Simpler C++ with C++14

ADC C++ 2014, Garching

slides: http://wiki.hsr.ch/PeterSommerlad/

IFS INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

A Quick Reality Check - please raise

- I use C++ regularly (ISO 1998/2003/2011/2014).
- I write "MyClass *x=new MyClass();" regularly.
- I know how to use std::vector<std::string>.
- I prefer using STL algorithms over loops.
- I am familiar with the Boost library collection.
- I've read Bjarne Stroustrup's "The C++ Programming Language 1st/2nd/3rd/4th ed"
- I've read Scott Meyers' "Effective C++. 3rd ed."
- I've read and understood Andrej Alexandrescu's "Modern C++ Design"
- I've read the ISO C++11 standard
- I wrote parts of the ISO C++ standard

```cpp
#include <iostream>
#include <iomanip>
using namespace std;

int main() {
  for (int i=1; i <=20; ++i){
    cout << '\n';
    for (int j=1; j <=20; ++j)
      cout << setw(4) << j*i ;
  }
  cout << '\n';
}
```

What's bad?

Just a running gag… aka example
a multiplication table

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
2	4	6	8	10	12	14	16	18	20	22	24	26	28	30	32	34	36	38	40
3	6	9	12	15	18	21	24	27	30	33	36	39	42	45	48	51	54	57	60
4	8	12	16	20	24	28	32	36	40	44	48	52	56	60	64	68	72	76	80
5	10	15	20	25	30	35	40	45	50	55	60	65	70	75	80	85	90	95	100
6	12	18	24	30	36	42	48	54	60	66	72	78	84	90	96	102	108	114	120
7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140
8	16	24	32	40	48	56	64	72	80	88	96	104	112	120	128	136	144	152	160
9	18	27	36	45	54	63	72	81	90	99	108	117	126	135	144	153	162	171	180
10	20	30	40	50	60	70	80	90	100	110	120	130	140	150	160	170	180	190	200
11	22	33	44	55	66	77	88	99	110	121	132	143	154	165	176	187	198	209	220
12	24	36	48	60	72	84	96	108	120	132	144	156	168	180	192	204	216	228	240
13	26	39	52	65	78	91	104	117	130	143	156	169	182	195	208	221	234	247	260
14	28	42	56	70	84	98	112	126	140	154	168	182	196	210	224	238	252	266	280
15	30	45	60	75	90	105	120	135	150	165	180	195	210	225	240	255	270	285	300
16	32	48	64	80	96	112	128	144	160	176	192	208	224	240	256	272	288	304	320
17	34	51	68	85	102	119	136	153	170	187	204	221	238	255	272	289	306	323	340
18	36	54	72	90	108	126	144	162	180	198	216	234	252	270	288	306	324	342	360
19	38	57	76	95	114	133	152	171	190	209	228	247	266	285	304	323	342	361	380
20	40	60	80	100	120	140	160	180	200	220	240	260	280	300	320	340	360	380	400

The output

Why is C++ “in” again?

- more computing per Watt!
  - mobile - battery powered
  - servers - cloud computing
  - high-performance computing & GPUs
- better abstractions than C
  - without performance price (e.g. of a VM)
  - embedded (higher-level type safety)
  - security (buffer overruns, pointers)

C++11 - What was new? (partial)

- “*It feels like a new language*” Bjarne Stroustrup
- **`auto`** for variable type deduction
- {} (almost) uniform initialization
- λ Lamdas - anonymous functions/functors
- **`&&`** Move-semantic, move-only types
- enums - strongly typed and scoped, constexpr
- **<T>** better template meta programming support
    - `<...>` variadic templates, type traits
- several library additions: function, tuple, regex
    - smart pointers for memory management

```
void multab_loops(std::ostream& out) {
  for (auto i=1; i <=20; ++i){
    out << '\n';
    for (auto j=1; j <=20; ++j)
      out << std::setw(4) << j*i ;
  }
  out << '\n';
}
```

C++11 auto

A testable multiplication table!

```
void testMultabLoopsDirect(){
   std::string const expected=R"(
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
   2   4   6   8  10  12  14  16  18  20  22  24  26  28  30  32  34  36  38  40
   3   6   9  12  15  18  21  24  27  30  33  36  39  42  45  48  51  54  57  60
   4   8  12  16  20  24  28  32  36  40  44  48  52  56  60  64  68  72  76  80
   5  10  15  20  25  30  35  40  45  50  55  60  65  70  75  80  85  90  95 100
   6  12  18  24  30  36  42  48  54  60  66  72  78  84  90  96 102 108 114 120
   7  14  21  28  35  42  49  56  63  70  77  84  91  98 105 112 119 126 133 140
   8  16  24  32  40  48  56  64  72  80  88  96 104 112 120 128 136 144 152 160
   9  18  27  36  45  54  63  72  81  90  99 108 117 126 135 144 153 162 171 180
  10  20  30  40  50  60  70  80  90 100 110 120 130 140 150 160 170 180 190 200
  11  22  33  44  55  66  77  88  99 110 121 132 143 154 165 176 187 198 209 220
  12  24  36  48  60  72  84  96 108 120 132 144 156 168 180 192 204 216 228 240
  13  26  39  52  65  78  91 104 117 130 143 156 169 182 195 208 221 234 247 260
  14  28  42  56  70  84  98 112 126 140 154 168 182 196 210 224 238 252 266 280
  15  30  45  60  75  90 105 120 135 150 165 180 195 210 225 240 255 270 285 300
                                             192 208 224 240 256 272 288 304 320
                                             204 221 238 255 272 289 306 323 340
  18  36  54  72  90 108 126 144 162 180 198 216 234 252 270 288 306 324 342 360
  19  38  57  76  95 114 133 152 171 190 209 228 247 266 285 304 323 342 361 380
  20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
)";
   std::ostringstream out;
   multab_loops(out);
   ASSERT_EQUAL(expected,out.str());
}
```

A test for the multiplication table

C++11

Raw String Literal

C++11 - What was new? (more)

- "*It feels like a new language*" Bjarne Stroustrup
- multi-threading, memory model, thread_local
  - library: thread, mutex, timed_mutex, condition_variable
- range-for + easier algorithm use with lambdas
- inheriting/delegating ctors, `=delete`, `=default`, non-static member initializers (NSDMI)

`class`

- `constexpr` functions and literal types
- `noexcept` spec instead of `throw()`
- `decltype(expr)` for type specification
- User-defined Literals - suffixes: `42_km`, `4_min`

{}

```
void multab_loops_vector(std::ostream &out) {
   std::vector<int> v{1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20};
   for (auto const i:v){
      out << '\n';
      for (auto const j:v)
         out << std::setw(4)<< i*j;
   }
   out << '\n';
}

void multab_iterator_vector(std::ostream &out) {
   std::vector<int> v(20); // not {20}!
   iota(begin(v),end(v),1);
   for (auto  i=v.cbegin(); i < v.cend(); ++i){
      out << '\n';
      for_each(cbegin(v),cend(v),[i,&out](int j){
         out << std::setw(4)<< *i * j;
      });
   }
   out << '\n';
}
```

C++11
auto, range for

C++11
Initializer List

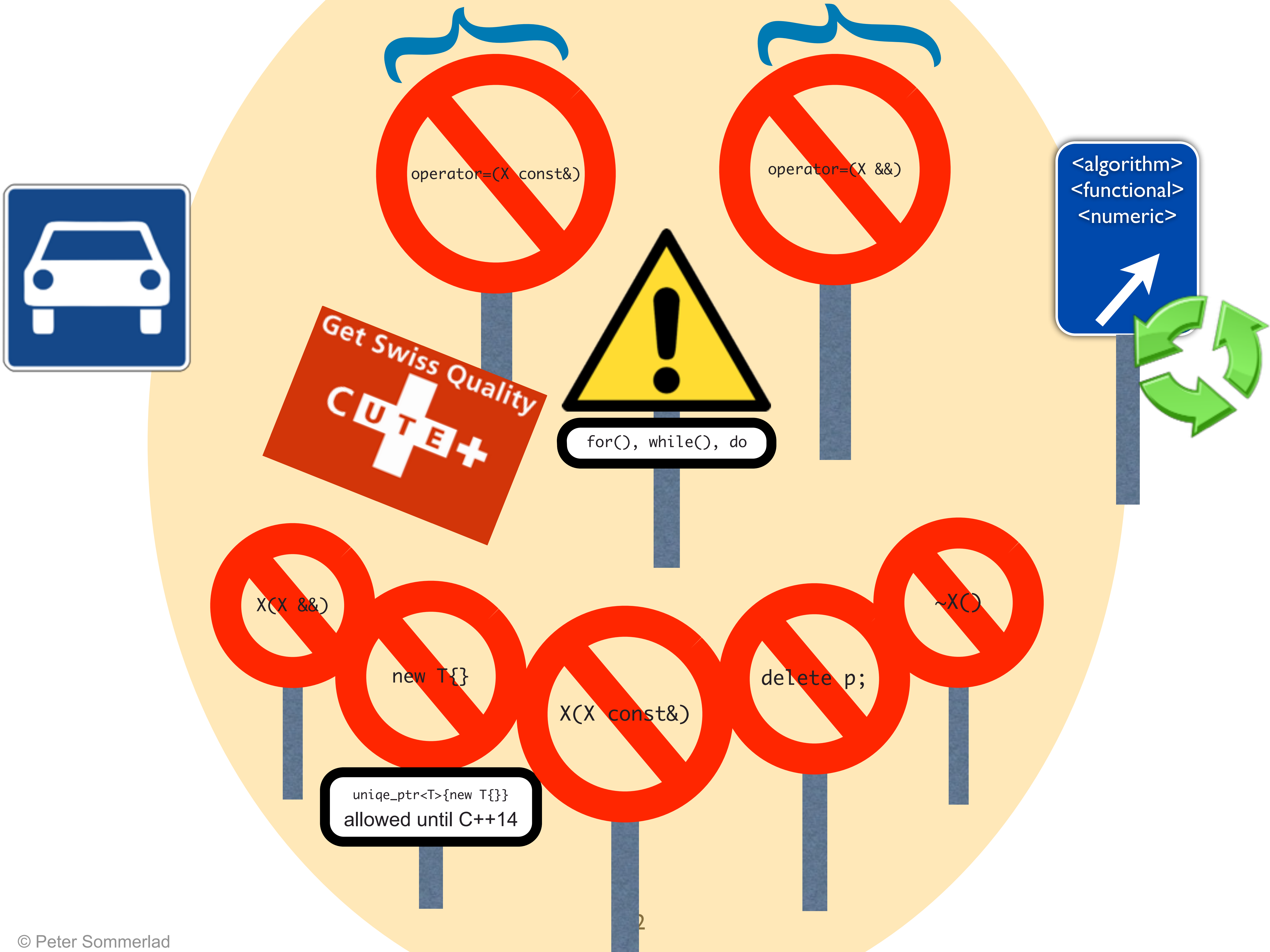
C++11
new algorithm

C++11
lambda

Demo of features

C++11 What does this imply?

- Many classic coding rules no longer applicable
  - "~~Rule of Three - Canonical Class~~"
  - ~~virtual Destructor with virtual Members~~
  - ~~manual memory management~~
- Many "complicated" things become easier or obsolete
  - complicated types in favor of auto
  - iterator usage vs. range-for loop
  - lambdas instead of functor classes


operator=(X const&)
operator=(X &&)
<algorithm>
<functional>
<numeric>
Get Swiss Quality
CUTE+
for(), while(), do
X(X &&)
new T{}
X(X const&)
delete p;
~X()
uniqe_ptr<T>{new T{}}
allowed until C++14

C++14 - What is new? (partial)

- “*Bug fix release*” Herb Sutter
- **`auto`** for function return type deduction
- **&&** Move-ability removes Copy-ability
- λ Lamdas - generic and variadic
- relaxed constexpr requirements (near full C++)
- make_unique<T>
- even better template meta programming
  - more convenient type_traits, tuples,
  - variable templates

C++14 - What is new? (partial)

- UDL operators for std::string, duration and complex
  - `“hello”s, 10s, 100ms, 3+5i`
- binary literals and digit separators
  - `0b1010`, `0b0’1000’1111`, `1’234’567.890’123`
- `shared_timed_mutex` (untimed to come)
- heterogeneous lookup in associative containers
  - `set<string>` can efficiently find “literals”
- heterogeneous standard functors, i.e., `plus<>`

```cpp
void multab_loops_binary(std::ostream& out) {
   for (int i=0b1; i <=0b10100; ++i){
      out << '\n';
      for (int j=0b0'0000'0001; j <=0b0'0001'0100; ++j)
         out << std::setw(0b100) << j*i ;
   }
   out << '\n';
}
template<typename MULTABFUNC>
void testMultab(MULTABFUNC multab) {
   using namespace std::string_literals;
   auto const expected=R"(
   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16  17  18  19  20
. . .
  20  40  60  80 100 120 140 160 180 200 220 240 260 280 300 320 340 360 380 400
)"s;
   std::ostringstream out;
   multab(out);
   ASSERT_EQUAL(expected,out.str());
}
```

C++14
binary literals

C++14
digit separators

C++14
UDL

Demo of C++14 features

C++1y - what’s next ?

- Library Fundamentals TS
  - `optional<T>` - optional values
  - boyer_moore_searcher - efficient search!
  - `any` - represent a value of arbitrary type
  - `string_view` - low-overhead wrapper for character sequences (no memory management)
  - polymorphic allocators - pooling, etc.
  - `sample()` algorithm - random sampling
  - `apply()` - call function with args from a tuple

C++1y more nexts?

- File System TS - standardized file access
- Dynamic arrays - (in C VLAs, but not the same)
- Feature test macros TS - version portability
- Parallel TS - vector(gpu?) and multi-core
- Concurrency TS - tasks, thread pools, continuations, executors
- Transactional memory TS - STM for C++
- Concepts-lite TS - categories for template Args

C++11/14 Examples

and some of my guidelines

auto

- deduction like template typename argument
- type deduced from initializer, use =
- use for local variables where value defines type

```
auto var= 42;
auto a   = func(3,5.2);
auto res= add(3,5.2);
```

- use for late function return types (not really useful, except for templates)

```
auto func(int x, int y) -> int {
  return x+y;
}
```

```
template <typename T, typename U>
auto add(T x, U y)->decltype(x+y){
  return x+y;
}
```

C++14: auto

- type deduction even for function return types (not only lambdas)

```
auto func(int x, int y) {
   return x+y;
}
```

- can even use decltype(auto) to retain references

```
template <typename T, typename U>
decltype(auto) add(T &&x, U &&y){
   return x+y;// may be overloaded
}
```

useful auto

- Use auto for variables with a lengthy to spell or unknown type, e.g., container iterators

```
std::vector<int> v{1,2,3,4,5};

auto it=v.cbegin();
std::cout << *it++<<'\n';

auto const fin=v.cend();
while(it!=fin)
    std::cout << *it++ << '\n';
```

- Also for for() loop variables
  - especially in range-based for()
  - can use &, or const if applicable

```
for (auto i=v.begin(); i!=v.end();++i)
    *i *=2;
```

```
for (auto &x:v)
    x += 2;
for (auto const x:v)
    std::cout << x << ", ";
```

C++ 03 - Problem: Initialization

- Plain Old Data - POD can be initialized like in C
  - But that doesn't work with non-POD types
  - except `boost::array<T,n>` all STL-conforming containers are NON-POD types.
- Using Constructors can have interesting compiler messages when omitting parameters
  - instead of initializing a variable, you declare a function with no arguments
  - who has not fallen into that trap?

```
struct B {};
B b();
```

  - declares a function called b returning a B and doesn't default-initialize a variable b

universal initializer {}

- C-struct and arrays allow initializers for elements for ages, C++ allows constructor call

```
struct point{
   int x;
   int y;
   int z;
};
point origin={0,0,0};
point line[2]={{1,1,1},{3,4,5}};
```

```
int j(42);
std::string s("hello");

int f();
std::string t();
```

What's wrong here?

- C++11 uses {} for "universal" initialization:

```
int i{3};
int g{};
std::vector<double> v{3,1,4,1,5,9,2,6};
std::vector<int> v2{10,0};
std::vector<int> v10(10,0);
```

Caveat: use () if ambiguous!

caveat: auto and initializer

```
auto i={3};
```

std::initializer_list<int>

{}

caveat: constructor overloads with initializer_list

std::initializer_list<int>

```
std::vector<int> v(10u); // vector(size_t n, T t=T{})
                         // 10 elements

std::vector<int> v{10u,2}; // vector with two int elements
// because initializer_list<int> matches parameters

std::vector<std::string> vs{10u}; // vector with 10 strings
// because intializer_list<std::string> doesn't match {10u}
// fallback to regular ctor overload
```

- initializer_list constructors have precedence over other ctors with matching parameters

C++14: decltype(auto) retains “referencyness” (©J.Wakely)

```
int i=3; // int
int &j=i;//int&
auto k=j; //int
decltype(auto) l=j;//int&
auto &&m=j; //int&
auto &&n=i; //int&&
```

```
template <size_t...I>
constexpr
auto make_compile_time_sequence(size_t const row,std::index_sequence<I...>)
{
   return std::array<size_t,sizeof...(I)>{{row*(1+I)...}};
}

void testIndexSequenceArray(std::ostream &out){
   auto const v=
make_compile_time_sequence(1,std::make_index_sequence<20>{});
   for (auto  i=v.cbegin(); i < v.cend(); ++i){
      out << '\n';
      std::for_each(cbegin(v),cend(v),[i,&out](auto j){
         out << std::setw(4)<< *i * j;
      });
   }
   out << '\n';
}
```

C++11
variadic template

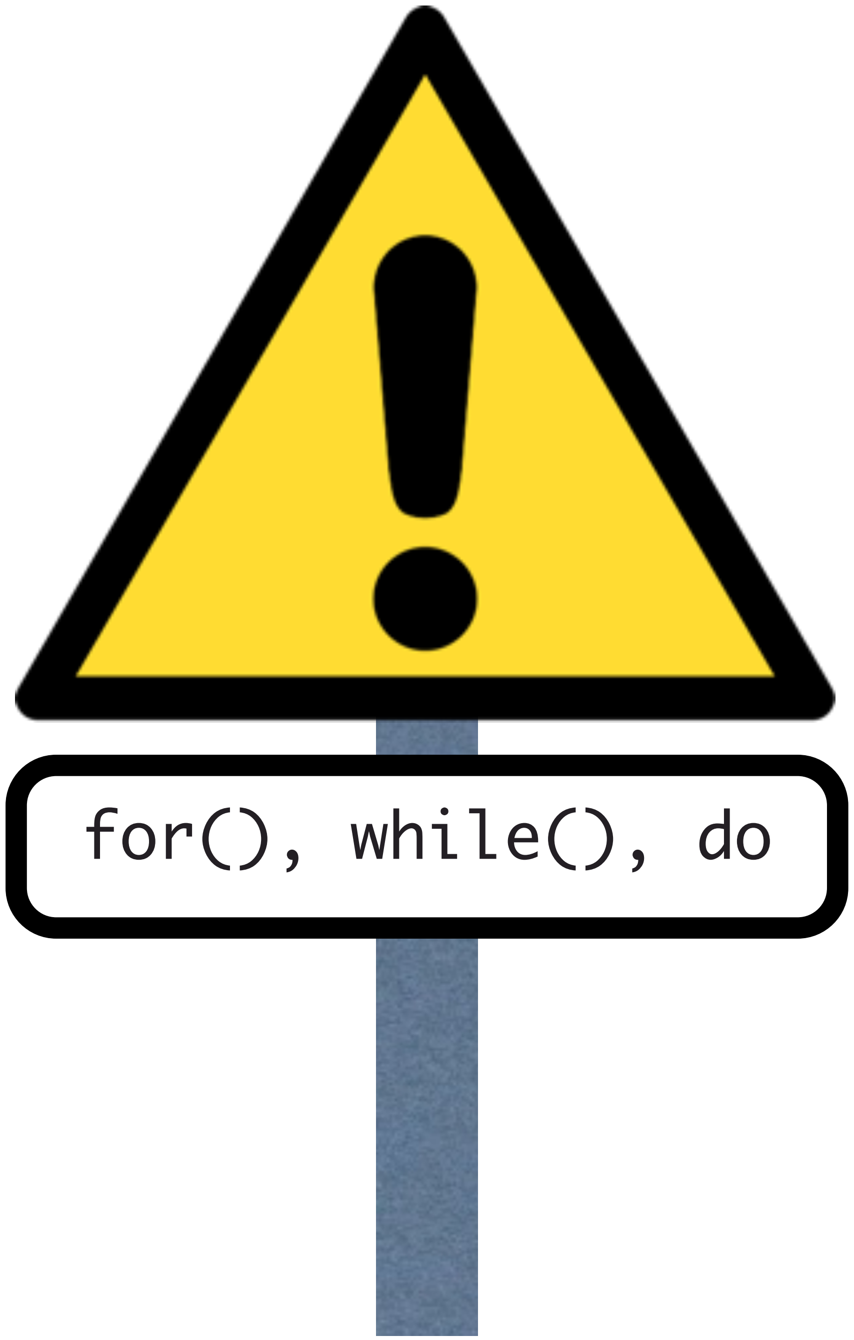
C++14
index_sequence

C++14
cbegin/cend

Demo of more features

Algorithms & λ

Re-Cycle instead of Re-Invent the Wheel


for(), while(), do


<algorithm>
<functional>
<numeric>
λ

Lambda - in-line functions/functors for use as algorithm parameters

- **Like many "functional" programming languages C++11 allows to define functions in "lambda" expressions**
  - `auto hello=[] { cout << "hello world" << endl;}; // store function`
  - `[]` -- lambda introducer/capture list
  - `(params)` -- parameters optional,
  - `-> return type` -- optional
  - `{...}` -- function body

```
#include <iostream>
int main(){
  using std::cout;
  using std::endl;
  auto hello=[]{
    cout << "Hello Lambda" << endl;
  };
  hello(); // call it
}
```

- **"lambda magic" -> return type can be deduced if only a single return statement**

```
auto even=[](int i){ return i%2==0;};
```

C++14 allows arbitrary body with type-consistent return statements and the use of auto for lambda parameters -> generic lambdas

- **or explicitly specified**

```
auto odd=[](int i)->bool{ return i%2;};
```

```
void multab_lambda_bind_vector(std::ostream &out) {
   std::vector<int> v(20); // not {20}!
   iota(begin(v),end(v),1);
     using oi=std::ostream_iterator<int>;
     using std::placeholders::_1;
   out << '\n';

     std::for_each(v.begin(),v.end(),[&out,v](auto y){
         transform(v.begin(), v.end(), oi{out,"\t"},
                 bind(std::multiplies<>{},y,_1));
                 out << '\n';
     });
}
```

C++11
iota() algo

C++14
generic lambda

C++14
generic functors

C++11
function binder

Demo of more features

Range-based for(auto const &x : c)-Loop

- **easy to use loop construct for iterating over containers, including arrays**
  - every container/object c where c.begin() or (std::)begin(c) and c.end() or (std::)end(c) are defined in a useful way
  - all standard containers
- **preferable to use auto for the iteration element variable**
  - references can be used to modify elements, if container allows to do so
    - `for (auto &x:v) { ... }`
  - in contrast to std::for_each() algorithm with lambda, where only value access is possible
- **initializer lists are also possible (all elements must have same type)**
  - `for (int i:{2,3,5,8,13}) { cout << i << endl;}`
- **my guideline: prefer algorithms over loops, even for range-based for.**
  - unless your code stays simpler and more obvious instead! (see outputting std::map)

Memory

```
new T{}
```

```
delete p;
```

```
uniqe_ptr<T>{new T{}}
```

allowed until C++14

```
NULL
(void*)0
```

```
use nullptr
```

Prefer unique_ptr/shared_ptr for heap-allocated objects over T*. Use std::vector and std::string instead of heap-allocated arrays.

std::unique_ptr<T> for C pointers

- some C functions return pointers that must be deallocated with the function `::free(ptr)`
- We can use unique_ptr to ensure that
  - __cxa_demangle() is such a function

```
std::string demangle(char const *name){
   std::unique_ptr<char,decltype(&::free)>
      toBeFreed{ __cxxabiv1::__cxa_demangle(name,0,0,0),&::free};
   std::string result(toBeFreed.get());
   return result;
}
```

- Even when there would be an exception, free will be called on the returned pointer, no leak!

Use shared_ptr<Base> for heap-allocated polymorphic Objects

- use make_shared<Derived>(…) to create them
- Works like Java/C# garbage collected references
- Sidesteps the need to define ~Base() destructor
  - shared_ptr will memorize the required derived class destructor
- Overhead for reference counting is there
  - plain unique_ptr<Base> doesn’t work without virtual destructor…

unique_ptr<Base,Special>

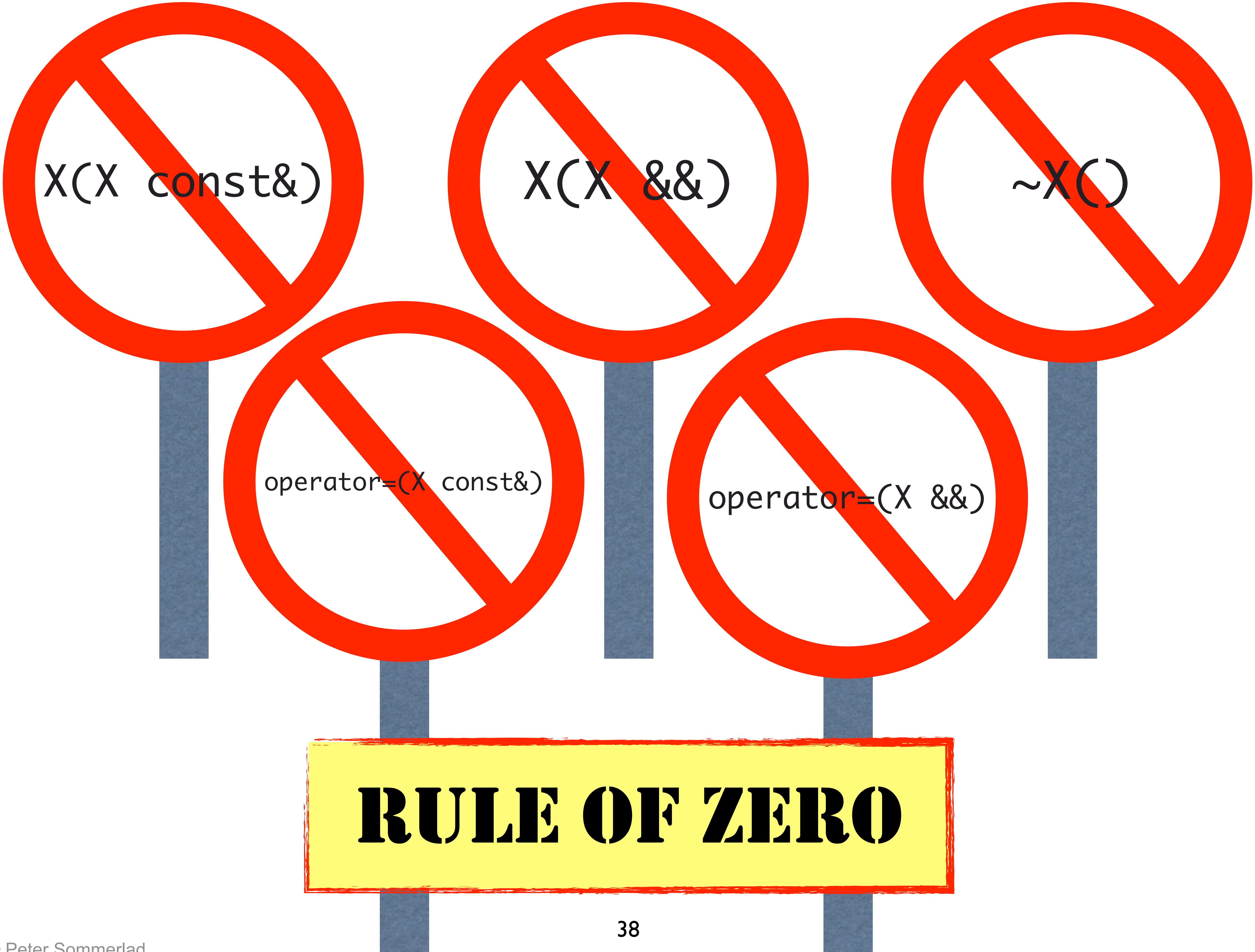
- A custom polymorphic destructor can solve the efficiency overhead of shared_ptr, when only one reference to a heap-allocated polymorphic object can exist. Requires a little more overhead than plain unique_ptr
- Won’t go into details here, similar to a unique_ptr<Base,std::function<void(Base*)>>
- Might be proposed for a future version of the standard

Rule for Resource Management

- Do not use raw Pointers or self-allocated buffers from the heap!

> Prefer unique_ptr/shared_ptr for heap-allocated objects over T*.
> Use std::vector and std::string instead of heap-allocated arrays.

- use smart pointers & standard library classes for managing resources.


X(X const&)
X(X &&)
~X()
operator=(X const&)
operator=(X &&)
RULE OF ZERO

Sommerlad's rule of zero

- As opposed to the "rule of three" of C++03
  - aka "canonical class"

> Write your classes in a way that you do not need to declare/define neither a destructor, nor a copy/move constructor or copy/move assignment operator

- use smart pointers & standard library classes for managing resources

Example: delegating constructors

- **Inspired a bit by Java**
  - but much less needed, because of default arguments
- **Example: Date class with overloaded constructors**
  - supports different cultural contexts in specifying dates

```
struct Date {
   Date(Day d, Month m, Year y) {
      // do some interesting calculation to determine valid date
   }
   Date(Year y, Month m, Day d):Date{d,m,y}{...}
   Date(Month m, Day d, Year y):Date{d,m,y}{...}
   Date(Year y, Day d, Month m):Date{d,m,y}{ }
};
```

Object completely constructed here

Inheriting constructors

```
template<typename T,typename CMP=std::less<T>>
struct indexableSet : std::set<T,CMP>{
    using SetType=std::set<T,CMP>;
    using size_type=typename SetType::size_type;
    using std::set<T,CMP>::set; // inherit constructors of std::set

    T const & operator[](size_type index) const {
        return this->at(index);
    }
    T const & at(size_type index) const {
        if (index >= SetType::size())
      throw std::out_of_range{"indexableSet::operator[] out of range"};
        auto iter=SetType::begin();
        std::advance(iter,index);
        return *iter;
    }
};
```

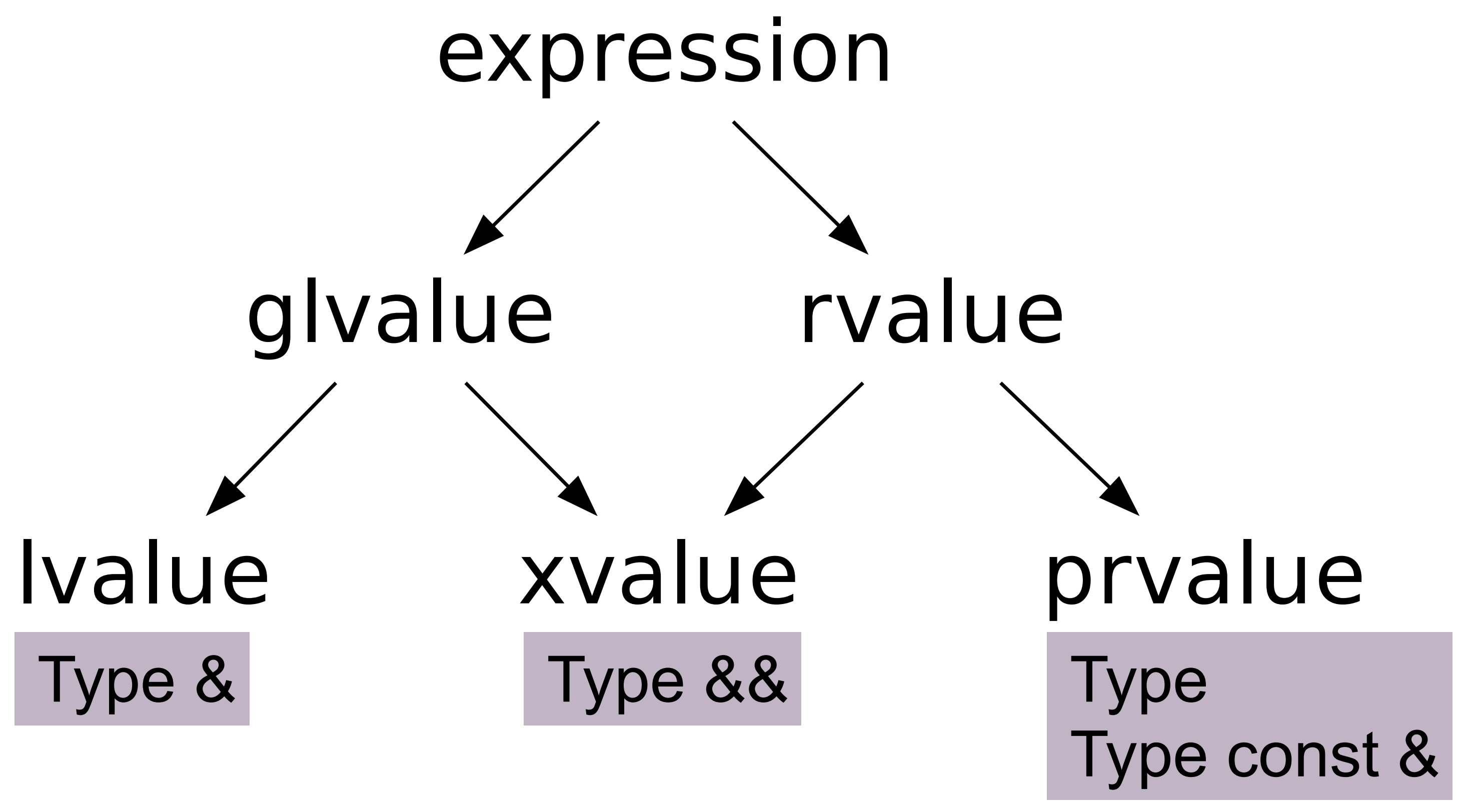
obtain all of std::set's ctors

- A std::set adapter providing indexed access
    - just don't add data members!
      and don’t expect dynamic polymorphism

Move Semantics - Do I need to care? -> **Usually not!**

- **In non-library code you might not need to care at all, things just work!**
  - often you do not need to care! Only (library) experts need to.
  - for elementary (aka trivial) types move == copy
- **R-Value-References allow optimal "stealing" of underlying object content**
  - copy-elision by compilers does this today for many cases already
    - e.g., returning std::string or std::vector
  - Type&& denotes an r-value-reference: reference to a temporary object
- **std::move(var) denotes passing var as rvalue-ref and after that var is "empty"**
  - if used as argument selects rvalue-ref overload, otherwise using var would select lvalue-ref overload or const-ref overload
- **like with const &, rvalue-ref && bound to a temporary extends its lifetime**

New Value Terminology in the Standard - rvalue-References



- **lvalue - "left-hand side of assignment" - can be assigned to**
  - glvalue - "general lvalue" - something that can be changed
- **rvalue - "right-hand side of assignment" - can be copied**
  - prvalue - "pure rvalue" - return value, literal
- **xvalue - "eXpiring value - object at end of its lifetime" - can be pilfered - moved from**

```
template <size_t...I>
constexpr
auto make_compile_time_square(std::index_sequence<I...> ){
   return std::array<std::array<size_t,sizeof...(I)>,sizeof...(I)>
         {{make_compile_time_sequence(1+I, // row
            std::make_index_sequence<sizeof...(I)>{})...}};
}
constexpr auto a = make_compile_time_square(std::make_index_sequence<20>{});

void testCompileTimeArray(std::ostream &out){
   using namespace std;
   constexpr auto a = make_compile_time_square(make_index_sequence<20>{});
   for_each(begin(a),end(a),[&out](auto row){
      out << '\n';
      for_each(begin(row),end(row),[&out](auto elt){
         out << setw(4) << elt;
      });
   });
   out << '\n';
}
```

Remember our example?

C++14 multiplication table computed at compile time

Announcement: Cevelop.com

- We will provide a one-stop download for C++ developers for an Eclipse-based IDE with our refactoring, unit testing and code generation plug-ins at **cevelop.com**
- Refactorings: Namespactor, Macronator, Elevator…
- Includator: include optimization
- cute-test, Mockator
- TBA: Cevelop quick-start C++11 including compiler

```
template <char... s>
using char_sequence=std::integer_sequence<char,s...>;
template <char ...s>
constexpr auto newline(char_sequence<s...>){
	return char_sequence<s...,'\n'>{};
}
constexpr auto newline(){
	return char_sequence<'\n'>{};
}

constexpr char make_digit_char(size_t const digit, size_t
const power_of_ten=1,char const zero=' '){
	return char(digit>=power_of_ten?digit/power_of_ten
+'0':zero);
}

template <size_t num>
constexpr auto make_chars_from_num(){
	static_assert(num < 1000, "can not handle large
numbers");
	return char_sequence< ' ',
		make_digit_char(num,100),
		make_digit_char(num%100,10,num>=100?'0':' '),
		char(num%10+'0')
	>{};
}



template <typename INT, INT ...s, INT ...t>
constexpr auto
concat_sequence(std::integer_sequence<INT,s...>,std::integer_
sequence<INT,t...>){
	return std::integer_sequence<INT,s...,t...>{};
}

template<size_t N, size_t ...I>
constexpr auto add(std::index_sequence<I...>){
	return std::index_sequence<(N+I)...>{};
}
template<size_t factor, size_t ...cols>
constexpr auto multiply(std::index_sequence<cols...>){
	return std::index_sequence<(factor*cols)...>{};
}
// can not use overload, but need to delegate to class
template with specialization
template <size_t...elts>
struct convert_to_charseq_impl;
// need level of indirection to do variadic template
recursion head-tail cutting
template <size_t n, size_t ...rest>
constexpr auto convert_to_charseq(){
	return
concat_sequence(make_chars_from_num<n>(),convert_to_charseq_i
mpl<rest...>{}());
}
template <size_t...elts>
struct convert_to_charseq_impl{
	constexpr auto operator()()const {
		return convert_to_charseq<elts...>();
	}
};
template <>
struct convert_to_charseq_impl<>{
	constexpr auto operator()()const{
		return char_sequence<>{};
	}
};

template<size_t ...cols>
constexpr auto makerowcharseq(std::index_sequence<cols...>){
	return newline(convert_to_charseq<cols...>());
}

template <size_t row, size_t num>
constexpr auto makerow(){
	constexpr auto
indices=multiply<row>(add<1>(std::make_index_sequence<num>{})
);
	return makerowcharseq(indices);
}

template <size_t n,size_t...cols>
struct make_rows_as_charseq_impl;

template <size_t n, size_t col, size_t ...rest>
constexpr auto make_rows_as_charseq(){
	return
concat_sequence(makerow<col,n>(),make_rows_as_charseq_impl<n,
rest...>{}());
}

template <size_t n,size_t...cols> //recurse indirectly
struct make_rows_as_charseq_impl {
	constexpr auto operator()()const{
		return make_rows_as_charseq<n,cols...>();
	}
};
template<size_t n> // base case
struct make_rows_as_charseq_impl<n> {
	constexpr auto operator()()const{
		return char_sequence<>{};
	}
};


template <size_t n,size_t ...cols>
constexpr auto makerows(std::index_sequence<cols...>){
	 return make_rows_as_charseq<n,cols...>();
}
template <char ...s>
constexpr static char const make_char_string[]={s... , '\0'};
template <char ...s>
constexpr auto const &
make_char_string_from_sequence(char_sequence<s...>){
	return make_char_string<s...>;
}

template <size_t dim>
constexpr auto multiplication_table=

make_char_string_from_sequence(concat_sequence(newline(), //
initial newline just to match raw literal

makerows<dim>(add<1>(std::make_index_sequence<dim>{}))));

constexpr char const * const
expectedresult=multiplication_table<20>;
```

C++14 multiplication table computed at compile time as char sequence

```
template <size_t dim>
constexpr auto multiplication_table=make_char_array(
     makerows<dim>(add1(std::make_index_sequence<dim>{})));

void test10by10VariableTemplateVersion(){
   using namespace std::string_literals;
   std::ostringstream out;
   out << multiplication_table<0b10>.data();
   auto const expected=R"(
 1   2
 2   4
)"s;
   ASSERT_EQUAL(expected,out.str());
   ASSERT_EQUAL(expected,multiplication_table<2>.data());
}
```

C++14
variable template

We can have is also as a VARIABLE template!

```cpp
//@main.cpp
#include <cstring>


int main() {
char filename[] = "myfile.txt";
strncpy(filename + strlen(filename) - 3, "doc", 3);
strncpy(filename - 3 + strlen(filename), "doc", 3);
strncpy(strlen(filename) - 3 + filename, "doc", 3);
strncpy(strlen(filename) + filename - 3, "doc", 3);
strncpy(-3 + strlen(filename) + filename, "doc", 3);
strncpy(-3 + filename + strlen(filename), "doc", 3);
}
```

```cpp
//=
#include <cstring>
#include <string>

int main() {
std::string filename = "myfile.txt";
filename.replace(filename.size() - 3, 3, "doc", 0, 3);
filename.replace(-3 + filename.size(), 3, "doc", 0, 3);
filename.replace(filename.size() - 3, 3, "doc", 0, 3);
filename.replace(filename.size() - 3, 3, "doc", 0, 3);
filename.replace(-3 + filename.size(), 3, "doc", 0, 3);
filename.replace(-3 + filename.size(), 3, "doc", 0, 3);
}
```

A Teaser

Upcoming C++ Refactoring
replace char* with std::string
actual test case

Questions ?

Software Health Check

Ask me!

- http://cute-test.com http://mockator.com
- http://linticator.com http://includator.com
- http://sconsolidator.com http://cevelop.com
- peter.sommerlad@hsr.ch http://ifs.hsr.ch

Have Fun with C++ Try TDD, Mockator and Refactoring!