Computer Science

Simple C++

Einfacheres C++ mit C++11

ADC C++ 2013

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

INSTITUTE FOR SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software
Bad Aibling, 8. Mai 2013

HSR
HOCHSCHULE FÜR TECHNIK
RAPPERSWIL
FHO Fachhochschule Ostschweiz

Peter Sommerlad
peter.sommerlad@hsr.ch

Credo:

- **Work Areas**
  - Refactoring Tools (C++, Scala, ...) for Eclipse
  - Decremental Development (make SW 10% its size!)
  - C++ (ISO C++ committee member)
- **Pattern Books (co-author)**
  - Pattern-oriented Software Architecture Vol. 1
  - Security Patterns
- **Background**
  - Diplom-Informatiker / MSc CS (Univ. Frankfurt/M)
  - Siemens Corporate Research - Munich
  - itopia corporate information technology, Zurich
  - Professor for Software Engineering, HSR Rapperswil, Director Institute for Software

- **People create Software**
  - communication
  - feedback
  - courage
- **Experience through Practice**
  - programming is a trade
  - Patterns encapsulate practical experience
- **Pragmatic Programming**
  - test-driven development
  - automated development
  - Simplicity: fight complexity

A Quick Reality Check - please raise your hands

- **I use C++ regularly (ISO 1998/2003).**
- **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"**
- **I've read Scott Meyers' "Effective C++. 3rd ed."**
- **I've read Andrej Alexandrescu's "Modern C++ Design"**
- **I've read the new ISO C++11 standard**
- **I wrote parts of the ISO C++ standard**
- **I know most of the ISO C++ standard by heart (not me :-)**

Smallest C++ program - is that simple?

- **the smallest valid standard compliant complete C++ program: smallest.cpp**

```
int main(){}
```

- **But it can even be smaller: evensmaller.cpp**

```
_
```

- **who can guess how?**
  - g++ -D_='int main(){}' evensmaller.cpp

A classic: helloworld.c (as generated from Eclipse CDT)

- **What is wrong with this? hello.c 1. it is C not even C++!**

```c
/*
 ==========================================================================
 Name        : helloc.c
 Author      :
 Version     :
 Copyright   : Your copyright notice
 Description : Hello World in C, Ansi-style
 ==========================================================================
 */

#include <stdio.h>
#include <stdlib.h>

int main(void) {
   puts("!!!Hello World!!!"); /* prints !!!Hello World!!! */
   return EXIT_SUCCESS;
}
```

belongs into version management system

not really needed in C, since functions are implicitly declared

ridiculous comment

redundant

HSR HOCHSCHULE FÜR TECHNIK RAPPERSWIL
FHO Fachhochschule Ostschweiz

IFS INSTITUTE FOR SOFTWARE

A reduced classic in C

- **Is that simpler? helloc_simpler.c**

```
int main() {
   puts("!!!Hello World!!!");
}
```

back to C++ hello world (as generated by Eclipse CDT)

- **What is wrong here**

```
//============================================================================
// Name        : helloworld.cpp
// Author      :
// Version     :
// Copyright   : Your copyright notice
// Description : Hello World in C++, Ansi-style
//============================================================================

#include <iostream>
using namespace std;

int main() {
	cout << "!!!Hello World!!!" << endl; // prints !!!Hello World!!!
	return 0;
}
```

belongs into version management system

bad practice, very bad in global scope

ridiculous comment

redundant

inefficient, redundant

using global variable! really bad :-(

A better (hello) world? How would it be?

- **Unit testable code mustn't use global (non-const) variables**
  - separate functionality from main() into a separate compilation unit or library
  - write unit tests against the library
  - make main() so simple, it cannot be wrong
- **using namespace pollutes namespace of compilation unit**
  - therefore never ever put "using namespace" in global scope within a header
  - prefer using declarations, like "using std::cout;" to name what you are actually using
    - functions and operators are automatically found when called, because of argument dependent lookup
- **ostream std::cout will flush automatically when program ends anyway**
  - use of std::endl most of the time a waste, because of flushing (except asking for input)

C++ Unit Testing with CUTE in Eclipse CDT

TDD Test-Driven Development and Refactoring

- **CUTE http://cute-test.com - free!!!**
- **simple to use - test is a function**
  - understandable also for C programmers
- **designed to be used with IDE support**
  - can be used without, but a slightly higher effort
- **deliberate minimization of #define macro usage**
  - macros make life harder for C/C++ IDEs and for programmers

Get Swiss Quality

CUTE+

A tested/testable hello world

Test:

```
#include "cute.h"
#include "ide_listener.h"
#include "cute_runner.h"
#include <sstream>
#include "sayHello2.h"
void testSayHelloSaysHelloWorld() {
   std::ostringstream out;
   sayHello(out);
   ASSERT_EQUAL("Hello world!\n",out.str());
}
void runSuite(){
   cute::suite s;
   s.push_back(CUTE(testSayHelloSaysHelloWorld));
   cute::ide_listener lis;
   cute::makeRunner(lis)(s, "The Suite");
}
int main(){
    runSuite();
}
```

Library:

```
#ifndef SAYHELLO_H_
#define SAYHELLO_H_

#include <iosfwd>
void sayHello(std::ostream &out);

#endif /* SAYHELLO_H_ */
```

```
#include "sayhello2.h"
#include <ostream>
void sayHello(std::ostream &out){
   out << "Hello world!\n";
}
```

Executable:

```
#include "sayhello2.h"
#include <iostream>
int main(){
   sayHello(std::cout);
}
```

Guideline for starting with simpler C++

- **Separate functionality from main()**
  - You can not write unit tests for main()!
- **Make both main() program as well as tests link to the "real" code in a library**
- **Write unit tests deliberately**
  - std::(i/o)stringstream is a real help for testing I/O
    - C++11 provides also std::to_string() function overloads for all numeric types
    - VS12 native unit testing requires you to overload a ToString() function for your type
  - requires "Parameterize from Above" for stream objects
    - that in turn requires pass-by-non-const-reference
- **Avoid using global <iostreams> variables in code or any global variables!**
  - except for passing them from main as arguments to functions called

C++ 03 - Problem: Complicated Return Types for Variables

- **Some library functions return complicated types, especially when using templates**
  - some help through typedefs or traits, but still cumbersome
  - std::vector<std::string>::const_reverse_iterator it=v.rbegin();
  - std::iterator_traits<iterator_type>::value_type v = *it;
- **With some library functions the return type is even "unspecified"**
  - You are not intended to keep track of it, e.g., bind1st(), tr1::bind()
  - how can you save its return value in a variable
    - well, it works in a template context, but not in general
- **for initializing heap objects, one even needs to repeat the type, like in Java**
  - there exists an alternative way in C++11: make_shared<type>(...)

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++11 **auto** provides Automatic Type Deduction for Initialized Variables

- **auto is a real life saver now**
  - `auto it=find(v.rbegin(),v.rend(),42);`
  - `auto first= *aMap.begin(); // std::pair<key,value>`
- **auto can be combined with (const) reference or pointer**
  - `auto i=42; auto &iref=i; // i is of type int, iref of type int&`
  - caveat: cannot use easily uniform initializer syntax without specifying the type
    - `auto i{42};` -> i is of type std::intializer_list<int>

- **Rule of Thumb:**
- **Define (local) variables with auto and determine their type through their initializer**
  - especially handy within template code!

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()
  - could 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!

C++11 provides "Uniform Initialization Syntax" using { }

- **about 13 pages of the standard explain initializers! (plus references to other chapters)**
- **however, one simple rule to understand it is sufficient in most applications**
  - `type variable {` *`list of expressions for initialization`* `};`
    - works for almost all types in all contexts, even for initializing container elements
    - `int i{}; double pi{3.14}; string s{"hello"}; vector<int> v{1,2,3,4,5,6};`
    - direct initialization, copy initialization uses = between the variable and the {
- **constructing a value for a given type is:**
  - `type {` *`list of expressions for initialization`* `}`
  - `int{}, double{5.5}, string{"hello world"}`
- **That's it.** (except for a few exceptions due to ambiguities)
- **Choosing Parentheses:**
  - () roun**D** **D**efinition, **D**eclaration -- {} **C**urly **C**onstruction, **C**reation

Examples for Using Uniform Initialization Syntax

Simple Variables

```
int i{}; // zero
std::string hello{"Hello World"};
char a[]{'h','a','i',char(i)}; // not only constants!
```

Vector Elements:

```
  std::vector<std::string> v { "eins", "zwei", "drei" };
  for (auto it=v.rbegin(); it != v.rend(); ++it){
    std::cout << "item : "<< *it << std::endl;
  }
```

Constructor Initializer Lists and Member Initializers

```
struct A {
  A(int i):val{i}{}
  int val;
  int const c{42}; // member initializer
};
```

caveat: auto and initializer

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

std::initializer_list<int>

Caveat - Constructor Overloads and initializer lists

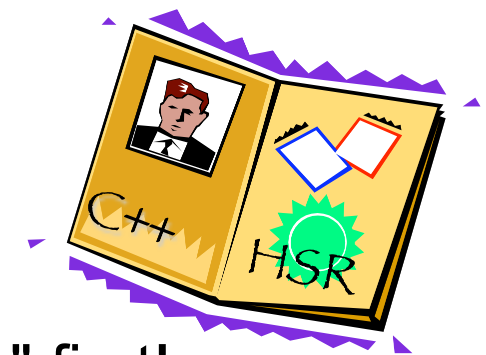
- **for a class with overloaded constructors and one overload is with initializer_list<T> using one of the other constructors that also takes parameters of type T might require using "traditional" parenthesis syntax.**

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

- **use uniform initializer syntax, but be prepared to use "old" parenthesis syntax in case to access non-initializer list constructor overload if both would match.**
  - problem with member-initializers (no viable syntax, must use constructor's initialization)

C++11 Example for Using initializer lists in own Classes

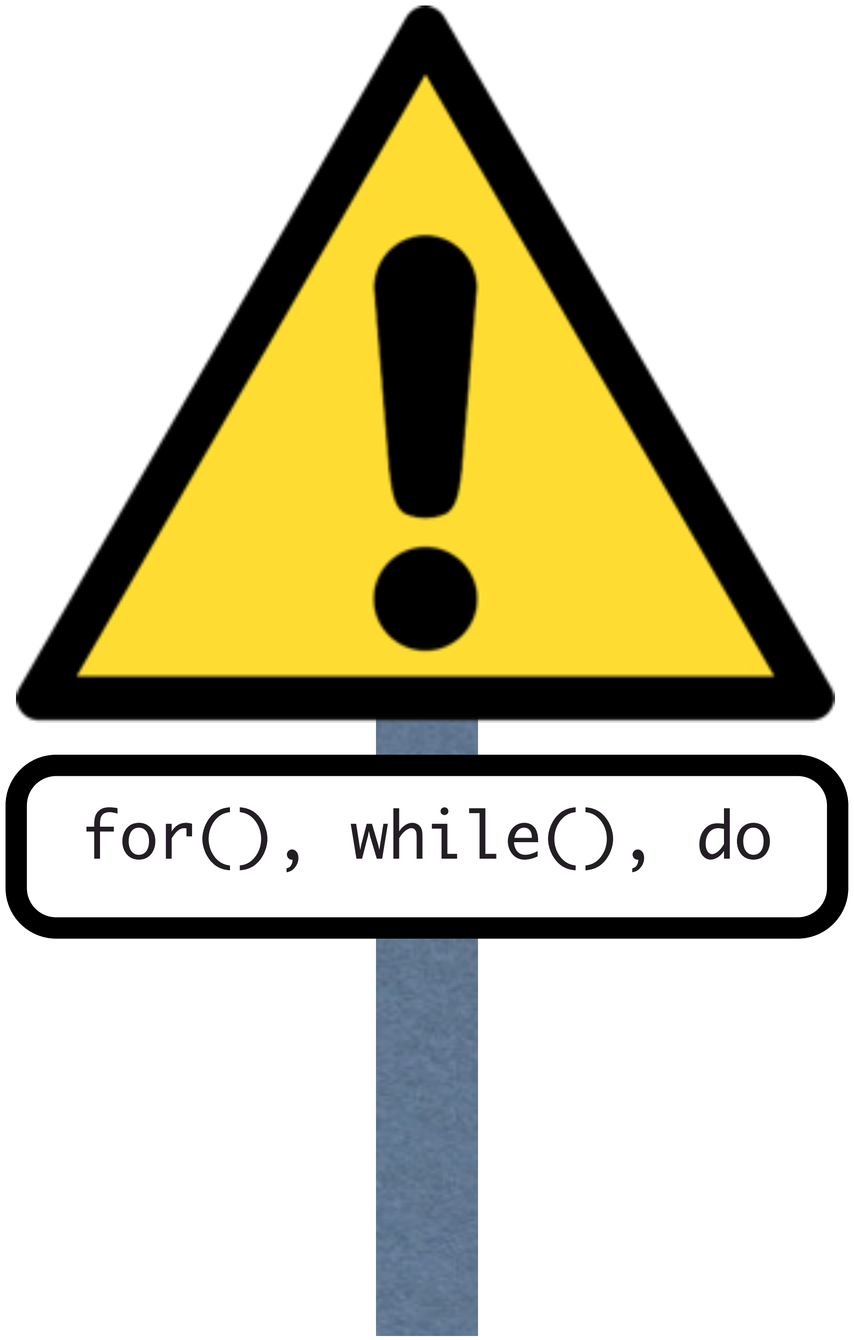
- **To use arbitrary length lists of values of the same type for your constructors the standard provides std::intializer_list<T> as a parameter type.**
  - Do not use std::initializer_list<T> for anything else.
  - Only if you design your own "container-like" types.
  - advanced stuff, beyond this talk --> you need your "C++11 pilot license" first! (can show on request...)




```
template<typename T> struct use_vector_memberT
{
    use_vector_memberT(std::initializer_list<T> l)
    :v{l}{}
    void print(std::ostream & out){
        for(auto x: v){
            out << "value = " << x << std::endl;
        }
    }
private:
    std::vector<T> v;
};
```

Algorithms & λ

Re-Cycle instead of Re-Invent the Wheel


for(), while(), do


<algorithm>
<functional>
<numeric>

Counting Stuff - wc like filters

1. **Count the number of non-whitespace characters in standard input**
2. **Count the number of bytes in standard input (aka wc -c)**
3. **Count the number of (whitespace separated) words in standard input (aka wc -w)**
4. **Count the number of lines in standard input (aka wc -l)**
5. **Tally the number of occurrences of each (alphabetical) character in input**
6. **Tally the number of occurrences of each word in input**
7. **sum up the numbers given in standard input**
8. **create a vector with the integers 1..20 and print a multiplication table**
   - in several variations...

Count the number of non-whitespace characters in standard input


```
#include <iostream>
int main(){
    size_t count{0};
    char c{};
    while (std::cin >> c) ++count;
    std::cout << count << '\n';
}
```

Universal Initializer Syntax! Avoids Problems with inadvertently declaring a function() when initializing a variable or creating a value.

```
#include <iostream>
#include <iterator>
int main(){
    using iter = std::istream_iterator<char>;
    std::cout << distance(iter{std::cin},iter{}) << '\n';
}
```

Count the number of bytes in standard input (aka wc -c)


```cpp
#include <iostream>
int main() {
    size_t count { 0 };
    while (std::cin.get())
        ++count;
    std::cout << count << '\n';
}
```

```cpp
#include <iostream>
int main() {
    size_t count { 0 };
    char c{};
    while (std::cin.get(c))
        ++count;
    std::cout << count << '\n';
}
```

```cpp
#include <iostream>
#include <iterator>
int main(){
    using iter = std::istream_iterator<char>;
    std::cout << distance(iter{std::cin},iter{}) << '\n';
}
```

Count the number of bytes in standard input (aka wc -c)


```cpp
#include <iostream>
int main() {
    size_t count { 0 };
    auto const eof=std::istream::traits_type::eof();
    while (std::cin.get()!=eof)
        ++count;
    std::cout << count << '\n';
}
```

```cpp
#include <iostream>
#include <iterator>
int main(){
    using iter = std::istreambuf_iterator<char>;
    std::cout << distance(iter{std::cin},iter{}) << '\n';
}
```

Count the number of words in standard input (aka wc)


```cpp
#include <iostream>
#include <string>
int main(){
    size_t count{0};
    std::string s{};
    while (std::cin >> s) ++count;
    std::cout << count << '\n';
}
```

```cpp
#include <iostream>
#include <iterator>
#include <string>
int main(){
    using iter = std::istream_iterator<std::string>;
    std::cout << distance(iter{std::cin},iter{}) << '\n';
}
```

create a vector with the integers 1..20 and print a multiplication table (2)


```
using veci = std::vector<int>;

veci create_iota(){
    veci v(20); // v{20} wouldn't work!
    iota(v.begin(),v.end(),1);
    return v;
}
void print_times(std::ostream& out, veci const& v) {
    typedef veci::value_type vt;
    typedef std::ostream_iterator<vt> oi;
    using std::placeholders::_1;

    std::for_each(v.begin(),v.end(),[&out,v](vt y){
        transform(v.begin(), v.end(), oi{out,", "},
                bind(std::multiplies<vt>{},y,_1));
                out << '\n';
    });
}
int main(){
    print_times(std::cout,create_iota());
}
```

- **for vector<int> initializer with {20} would create a vector with just this element**
- **iota takes the 1 and assigns the value and increments it for each step**
  - its name comes from APL ι
  - there is no iota_n()
- **lambda capture by reference and by copy/value here**
  - best to explicitly name captured variables
  - avoid dangling references!
- **bind is now part of std:: namespace**
  - in contrast to boost::bind need namespace placeholders
  - better with using ... _1

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)

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
  - `(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;};
```

- **or explicitly specified**

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

Lambda Captures

- **In addition to function parameters passed on call, Lambdas can "capture" variables from the scope where they are defined. These can be used in the lambda body.**
- **auto with_capture=[x](int i){ cout << x+i << endl;}**

```
void demo_lambda_capture(){
   int x=3;
   std::vector<int> v(10u);
   iota(v.begin(),v.end(),1);
   auto it=find_if(v.begin(),v.end(),[x](int i){return i>x;});
   std::cout << "found "<< (it !=v.end()?*it:0)<<std::endl;
}
```

- **using [=] captures all variables implicitly by copy**
  - recommended practice!
- **using [&] captures all used variables implicitly by reference**
  - CAUTION: calling the lambda after a captured by reference variable's lifetime is over results in undefined behavior!
- **using the variables names and preceding them with & or not can provide a mixture of capture by copy and capture by reference**

Lambdas without captures are compatible with function pointers

- **Lambdas without captures are compatible with the corresponding function pointer type**
  - they can be used as simple callback functions declared as function pointers
  - ```
    void qsort( void * base, size_t num, size_t size,
                int ( * comparator ) ( const void *, const void * ) );
    ```

```
array<int,10> a;
iota(a.begin(),a.end(),1);
random_shuffle(a.begin(),a.end());
copy(a.begin(),a.end(),ostream_iterator<int>(cout,", "));
qsort(a.data(),a.size(),sizeof(int),
       [](void const *i,void const *j)
       {return *static_cast<int const*>(j)-*static_cast<int const*>(i);}
);
cout << endl;
copy(a.begin(),a.end(),ostream_iterator<int>(cout,", "));
```
```

Lambda Special Case: mutable

```
int x{}; // memory for lambda below
generate_n(std::back_inserter(v),10,[x]() mutable {
   return ++x , x*x; // mutable allows change
});
```

allow changing x

- variables captured by copy (=) are const within the lambda, unless ...
  - the lambda is marked **`mutable`**
  - the lambda gets its own copy of the variable
- lambdas are mapped internally to functors

Lambdas in Member Functions

```
struct DemoLambdaMemberVariables {
   int x{};
   std::vector<int> demoAccessingMemberFromLambda() {
      std::vector<int> v;
      generate_n(back_inserter(v),10,[=] {
         return ++x, x*x; // member x can be changed
      });
      return v;
   }
};
```

capture this by copy

- Captured member variables are always captured by reference, even if the default is [=]
  - Reason: this is a pointer (reference) and if copied, member variables are referred from it
  - `this` can not be captured by reference

Universal function holder - function<FUNC> template

- **Template std::function<> can store functions, functors, lambdas or binders**
- **template parameter defines call signature**
- **can be "empty" and that can be checked**
- **ideal to keep callbacks or command objects, even when they are functors or lambdas or the result of bind() expressions**

```
#include <functional>
#include <iostream>
int main(){
  using std::cout;
  using std::endl;
  std::function<bool(int)> odd;
  if (not odd) cout <<"odd is empty"<< endl;
  odd = [](int i)->bool{ return i%2;};
  if (odd) cout << "odd is defined" << endl;
  if (odd(2)) cout << "2 is odd"<< endl;
  if (odd(3)) cout << "3 is odd"<< endl;
}
```

Function Binder: bind()

- **obsoletes deprecated bind1st, bind2nd, mem_fun, mem_fun_ref, ptr_fun functions**
- **universal function composition and argument binding**
  - often with standard functors
  - can change arity of functions by "fixing" argument
  - can combine functions
  - reference arguments can be passed through ref() and cref() wrappers
- **namespace std::placeholders defines _1,_2, ... to be used in place where remaining parameters appear in bind**
- **less important with lambdas, but boost::bind provides same features for C++03**
  - std::bind() can do a bit more than lambdas in generic code
    - lambdas parameter's type can not (yet) be deduced, must be named, bind's placeholders are generic

std::bind() example


```
#include <functional>
#include <algorithm>
#include <vector>
#include <iostream>
#include <iterator>

int main(){
  using namespace std;
  using placeholders::_1;
  vector<int> numbers(10);
  iota(numbers.begin(),numbers.end(),1);
  ostream_iterator<int> out(cout,", ");
  copy(numbers.begin(),numbers.end(),out); cout << endl;
  transform(numbers.begin(),numbers.end(),out,
      bind(plus<int>(),_1,10));
  cout << endl;
  transform(numbers.begin(),numbers.end(),out,
      bind(multiplies<int>(),_1,_1));
  cout << endl;
}
```

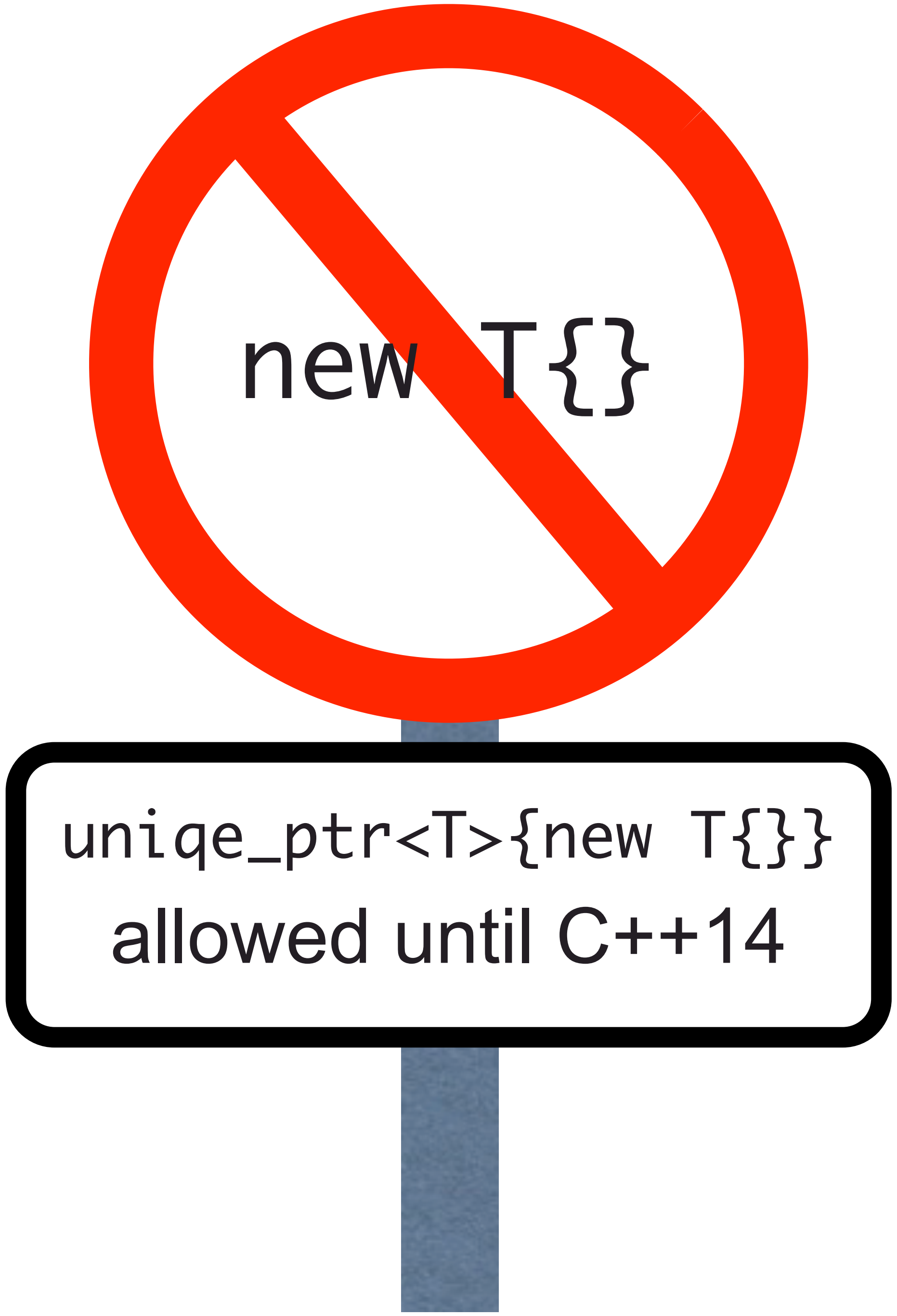
```
1, 2, 3, 4, 5, 6, 7, 8, 9, 10,
11, 12, 13, 14, 15, 16, 17, 18, 19, 20,
1, 4, 9, 16, 25, 36, 49, 64, 81, 100,
```

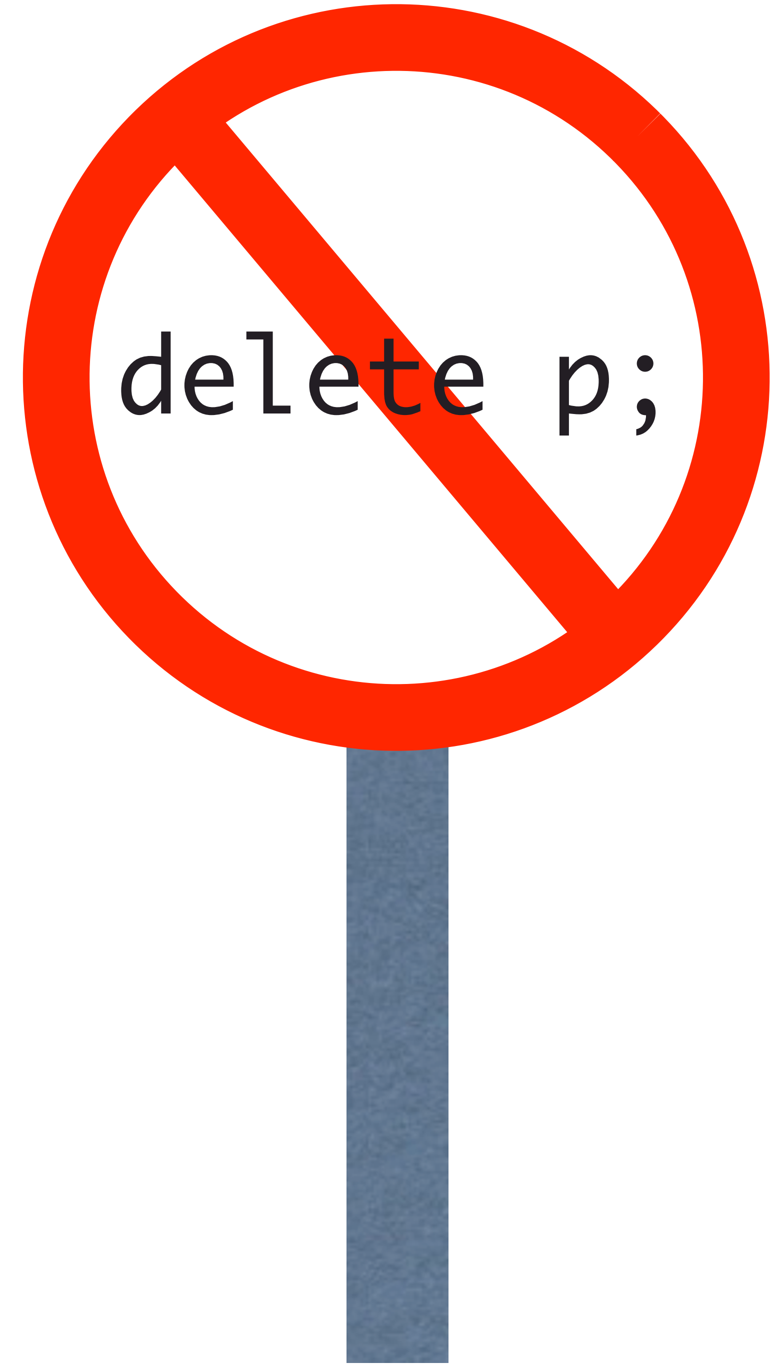
Pointers? No more naked pointers!

- **nullptr**
  - explicit type and value for singular pointer values (no more 0 or NULL confusion)
- **shared_ptr<T>**
  - obtain one with make_shared<T>(ctor_arguments)
  - a reference counting type
- **uniqe_ptr<T>**
  - for non-shared resource-managed pointers and RAII
  - a move-only type (it just works)

- **C++14: use optional<T> or optional<reference_wrapper<T>> for optional values instead of a pointer, e.g., as return value of something that might not exist.**
  - boost::optional available today!

Memory





- **C++11 finally defines a definite value for null pointers: `nullptr`**
- **has a distinct type nullptr_t**
  - not like 0 which could be interpreted as an int or as a null pointer
  - can not indadvertedly be mixed with numbers in calculations
- **is compatible with all (raw and smart) pointer types in C++**
  - all can be compared against nullptr
  - nullptr evaluates to false in a bool context
- **nullptr_t is important for providing overloads in case of passing a nullptr value around**
  - especially for comparison operators of smart pointers

RULE: in C++11 use nullptr wherever you would have written NULL or 0 (as pointer)

No More "naked" Pointers: Resource Management with shared_ptr<T>

- **If you really need to keep something explicit on the heap, use a factory like that:**

```
#include <memory>
#include <string>
struct A{
    A(int a,std::string b, char c){}
};

std::shared_ptr<A> A_factory(){
    return std::make_shared<A>(5,"hi",'a');
}
```

- **example usages:**

```
int main(){
    auto an_a=A_factory();
    auto b=an_a; // second pointer to same object
    A c{*b}; // copy ctor.
    auto another = std::make_shared<A>(c); // copy ctor on heap
}
```

No More "naked" Pointers: Class hierarchies with shared_ptr<T>

- **use std::ostream, just as an example for a base class**
  - and a very primitive factory function, need to use concrete type with make_shared

```
std::shared_ptr<std::ostream> os_factory(bool str){
    if (str)
        return std::make_shared<std::ostringstream>();
    else
        return std::make_shared<std::ofstream>("hello.txt");
}
```

- **a very simple usage scenario:**

```
int main(){
    auto out = os_factory(true);
    auto file= os_factory(false);
    *out << "hello";
    *file << "world";
}
```

No More "naked" Pointers: resource Management with unique_ptr<T>

- **for non-shared heap-allocated objects aka "what auto_ptr wanted to be"**
  - still usable for factory functions and
  - it is guaranteed a Highlander: "There can be only one!" - no second reference

```
#include <memory>
#include <iostream>
std::unique_ptr<int> afactory(int i){
    return std::unique_ptr<int>{new int{i}};
}
int main(){
    auto pi=afactory(42);

    std::cout << "*pi =" << *pi << '\n';
    std::cout << "pi.valid? "<< std::boolalpha
              << static_cast<bool>(pi) << std::endl;
    auto pj=std::move(pi);
    std::cout << "*pj =" << *pj << '\n';
    std::cout << "pi.valid? "<< std::boolalpha
              << static_cast<bool>(pi) << std::endl;

}
```

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!

- **unique_ptr<T> for RAII and factories and non-copyable PIMPL idiom**
  - only and exactly one owner
  - can be empty (aka null) -> check in bool context or check if get() returns nullptr
  - unfortunately no "make_unique<T>(args)" function -> there are reasons for it!
  - "last" place where you explicitly (have to) write "`new Type{arguments}`"
- **shared_ptr<T> for reference counted, safely removed, heap-allocated objects**
  - take care of cyclic references -> might need to use weak_ptr<T> also
  - shared ownership, safe access, can be empty as well
  - can be stored in containers safely, can be used in a thread-safe manner (atomic update)
- **weak_ptr<T> for non-owning access to a shared_ptr<T> referred object**
  - can become invalid, but you can check
  - must obtain underlying shared_ptr<T> before using object with wp.lock()

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.

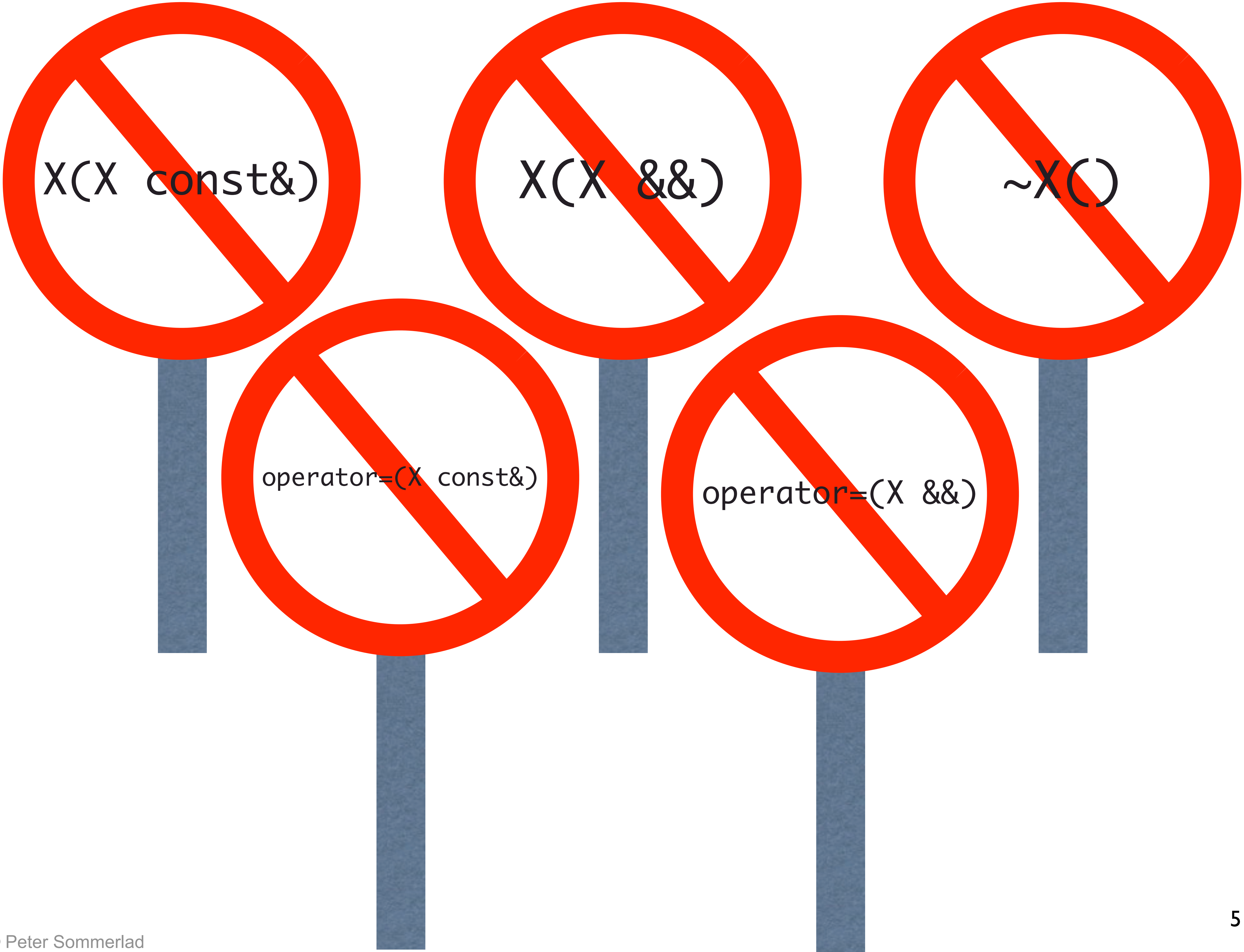
- use smart pointers & standard library classes for managing resources. C++14 will even provide dynamic arrays on stack/heap.

simpler classes in C++11

- **non copyable classes, e.g., for keeping unique resources**
  - pre C++11: declare copy ctor and assignment operator private
  - or, inherit from boost::noncopyable
- **default default constructor**
  - defining a constructor deletes the automatic definition of a default default constructor
- **Inheriting from a class with many constructors without adding members requires to respell all constructors and delegate to the parent (pre C++11)**
  - ugly, when you want to wrap a simple behavioural extension (no release impl. yet)
- **Unwanted type conversions can be annoying when using built-in parameters and overloaded functions -> easily lead to ambiguity**
- **Defining many constructors for a class with many members/bases can be annoying because of duplication**
  - sometimes needed, because of overload rules -> delegating ctors

Defining classes!

- **about 7 pages of rules for compiler provided copy/move ctor and destructor in the standard document.**
  - It is very hard to know and apply them all correctly.
  - However, C++11 allows to use appropriate library stuff so that you do not need to care!
    - containers, strings, smart pointers
- **Write your classes in a way that you do not need to care**
  - let the standard library provide all resource management for you!
    - std::string, std::vector and the other containers are your friends
  - use std::shared_ptr<X> if you intend to keep your class copyable and share the resource
    - keeps your class default copyable and moveable
  - use std::unique_ptr<Y> if you want to be the only one caring for your resources
    - this makes your class move-only, if you do not inhibit move-ctor/move-assignment otherwise
- **No more manual memory management needed, if done right**
  - even better than in Java, because destruction is deterministic!


X(X const&)
X(X &&)
~X()
operator=(X const&)
operator=(X &&)

Sommerlad's rule of zero

- As opposed to the "rule of three"
  - 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

A non-copyable class: no_copy_old in previous years

- **reasonable for classes managing resources that cannot be easily shared**
  - not Singletons (just don't, another story for another day...)

```
class no_copy_old {
   int *pi; // manages a resource
public:
   no_copy_old(int x):pi(new int(x)){}
   ~no_copy_old();
private:
   no_copy_old(no_copy_old const &);
   no_copy_old& operator=(no_copy_old const &);
};
```

```
#include "no_copy_old.h"
void foo(no_copy_old nc){}


int main(){
   no_copy_old nco(42);
   foo(nco); // error
   no_copy_old cco(nco);// error
}
```

- **Error messages are subtle and first point to declaration not usage, e.g.,**

```
../no_copy_old.h:9:2: error: 'no_copy_old::no_copy_old(const no_copy_old&)' is private
../no_copy_main.cpp:6:9: error: within this context
../no_copy_main.cpp:2:6: error:   initializing argument 1 of 'void foo(no_copy_old)'
```

A non-copyable class: no_copy


```
../no_copy_main.cpp:13:15: error: use of deleted function 'no_copy::no_copy(const no_copy&)'
```

- **reasonable for classes managing resources that cannot be easily shared**
  - not Singletons (just don't, another story for another day...)

```cpp
class no_copy {
   int *pi;// manages a resource
public:
   no_copy(int i);
   ~no_copy();

   no_copy(no_copy const&)=delete;
   no_copy& operator=(no_copy const&)=delete;
};
```

```cpp
#include "no_copy.h"
no_copy::no_copy(int i) :
      pi{new int(i)} {
}
no_copy::~no_copy() {
   delete pi;
}
```

- **define copy ctor and assignment as =delete**
  - still are defined and take part in overload resolution

```cpp
#include "no_copy.h"
void foo(no_copy nc){
   // doesn't compile, at when called
}
int main(){
   no_copy nc(42);
   // no_copy nul;
   // foo(nc);
   // crashes with double delete
   //when copying is allowed & foo called
}
```

- **defining a ctor removes default ctor, would need to re-implement**
  - subtle differences between own default ctor and compiler generated one.

```
class no_copy {
   int *pi;// manages a resource
public:
   no_copy(int i);
   no_copy()=default;
   ~no_copy();
   no_copy(no_copy const&)=delete;
   no_copy& operator=(no_copy const&)=delete;
};
```

- **problem with default default ctor, members might have indeterminate value**
  - solution: provide member initializer with nullptr:

```
class no_copy {
   int *pi{nullptr}; // manages a resource
public:
   no_copy(int i);
   no_copy()=default;
...
```

Other purposes for =delete

- **Overloads with parameters where non-explicit automatic conversion takes place**
  - and you do not want to have it called with another type

```
struct only_unsigned_long_long {
template <typename T> only_unsigned_long_long(T) = delete;
only_unsigned_long_long(unsigned long long){};
};

only_unsigned_long_long pi{42ULL};
only_unsigned_long_long fails_because_of_int{3}; // error
```

- **works a bit like an inverted explicit**
  - can only call function with the correct type
  - can also be used without template to inhibit just a specific overload

Prohibit instantiation template class

```
template <typename T>
struct Sack<T*> {
  ~Sack()=delete;
};
```

- Avoid instantiating a container with naked pointers
- A class template specialization can have any content, even no content at all
  - it can be completely unrelated to the original template, there is really **no relationship**!!!
- One means to prohibit instantiating a class is to prohibit the ability to its destruction by declaring its destructor as `=delete;`

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 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

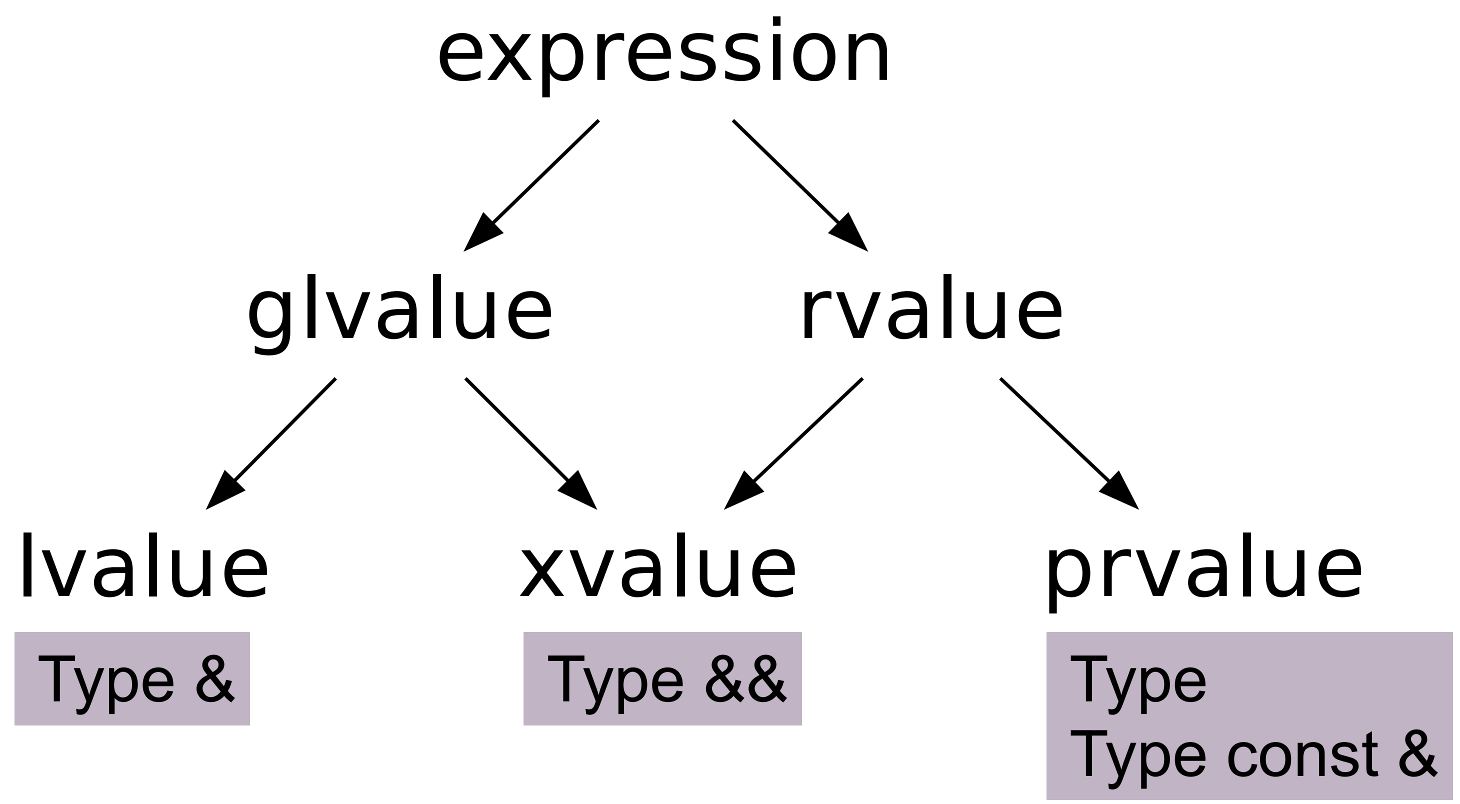
Class Templates that inherit

- Rule: always use `this->` or the class name to refer to inherited members in a template class
  - if the name could be a dependent name the compiler will not look for it when compiling the template definition.
  - Checks might only be made for dependent names at template usage (= template instantiation)

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**

Move Semantic

- **Goal: no unnecessary object copies, transfer temporaries efficiently**
- **Mechanisms:**
  - r-value references: `Type&&`
  - move-ctors `Type(Type&&)`, move-assignment `operator=(Type&&)`
  - deleted copy-ctor, copy assignment -> move-only type
  - `std::move(lvalue)` -> prepare lvalue(aka variable) to move from
- **relevant for classes that manage (expensive) internals and can hand those over**
  - e.g. containers
- **relevant for classes that couldn't keep invariant if copied**
  - e.g. std::unique_ptr, std::future, "single-ownership" objects

Copy Elision works even before C++11

- **"normal" users can rely on good compiler technology**
- **return by value is OK (even with C++03)**
  - aka "return value optimization"

- **With moveable types, pass by value can be more efficient than pass by const&**
  - subtle things happen, with pass by value a temporary gets moved into parameter
  - moving from a const& when a copy of the parameter is needed, doesn't work
- **If you do not explicitly design your own move-only/move-enabled types you usually do not need to care when using the standard library**

Example of Move Semantic with Move-only Type


```
Move constructor called
f(MoveOnly) called
g(MoveOnly&&) called
moving lvalues:
Move constructor called
f(MoveOnly) called
g(MoveOnly const&) called
g(MoveOnly&&) called
```

```cpp
   struct MoveOnly
{
   MoveOnly() = default;
   MoveOnly(MoveOnly&&) {
       std::cout << "Move constructor called\n";
   }
   MoveOnly(const MoveOnly&) = delete;
};
void f(MoveOnly) { std::cout << "f(MoveOnly) called\n"; }
void g(MoveOnly&&) { std::cout << "g(MoveOnly&&) called\n"; }
void g(MoveOnly const &){ std::cout << "g(MoveOnly const&) called\n"; }
int main(){
   f( MoveOnly{} ); // rvalue temporary
   g( MoveOnly{} ); // rvalue temporary
   std::cout << "moving lvalues:\n";
   MoveOnly mv{}; // lvalue
   //f(mv); // doesn't compile, lvalue cannot be passed (would require copy-ctor)
   f(std::move(mv)); // make an rvalue from lvalue
   g(mv); // binds to const-ref
   g(std::move(mv)); // binds to rvalue-ref
}
```

Questions ?

Software Health Check

Ask me!

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

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