Computer Science

Simpler C++ with C++11/14

Einfacheres C++ mit C++11/14

Meeting C++ 2013

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

IFS
INSTITUTE FOR
SOFTWARE

Prof. Peter Sommerlad
Director IFS Institute for Software
Credo:

- **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

**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
A Quick Reality Check - please raise your hands

- 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
\begin{itemize}
\item \texttt{operator=(X const&)}
\item \texttt{operator=(X &&)}
\item for(), while(), do
\item \texttt{X(X &&)}
\item \texttt{new T{}}
\item \texttt{X(X const&)}
\item \texttt{delete p;}
\item \texttt{unique_ptr<T>{new T{}}} \textit{allowed until C++14}
\item \texttt{~X()}
\end{itemize}
Smallest C++ program - is that simple?

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

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

- But it can even be smaller: evensmaller.cpp

```
-
```

- who can guess how?
  - `g++ -D_="int main(){}" evensmaller.cpp`
C++ hello world (as generated by Eclipse CDT)

What is wrong here

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

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

- `#include <iostream>` belongs into version management system
- `using namespace std;` bad practice, very bad in global scope
- `int main() {` ridiculous comment
- `cout << "!!!Hello World!!!" << endl;` redundant
- `return 0;` inefficient, redundant
- `using global variable! really bad if not in main() :-(`
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

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
A tested/testable hello world

Test:

```cpp
#include "cute.h"
#include "ide_listener.h"
#include "xml_listener.h"
#include "cute_runner.h"
#include <sstream>
#include "sayHello2.h"

void testSayHelloSaysHelloWorld() {
    std::ostringstream out;
    sayHello(out);
    ASSERT_EQUAL("Hello world!
", out.str());
}

void runAllTests(int argc, char const *argv[]){
    cute::suite s;
    s.push_back(CUTE(testSayHelloSaysHelloWorld));
    cute::xml_file_opener xmlfile(argc,argv);
    cute::xml_listener<cute::ide_listener<> > lis(xmlfile.out);
    cute::makeRunner(lis,argc,argv)(s, "AllTests");
}

int main(int argc, char const *argv[]){
    runAllTests(argc,argv);
    return 0;
}
```

Library:

```cpp
#ifndef SAYHELLO_H_
#define SAYHELLO_H_
#include <iosfwd>

void sayHello(std::ostream &out);
#endif /* SAYHELLO_H_ */

#include "sayhello2.h"
#include <iostream>

void sayHello(std::ostream &out){
    out << "Hello world!
";
}

#include "sayhello2.h"
#include <iostream>

int main()
{
    sayHello(std::cout);
    return 0;
}
```
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
  - 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
  - if done seriously <iostream> should only be included where main() is defined
    - if using streams include <istream> and/or <ostream> instead -> no globals!
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., `std::bind1st()`, `std::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: `std::make_shared<type>(...)`
- and in C++14 you even get `std::make_unique<type>(...)`

C++14 even allows you to deduce the return type of functions, if its return statements are consistent with respect to its return type
auto

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

```cpp
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)

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

```cpp
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)

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

- can even use decltype(auto) to retain references

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

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

```cpp
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';

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 << " , ";
```
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:

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

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

```cpp
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);
```

- int j(42);

- std::string s("hello");

- int f();

- std::string t();

What's wrong here?

Caveat: use () if ambiguous!
caveat: auto and initializer

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

```cpp
std::initializer_list<int>
```
C++14: decltype(auto) retains reference

```cpp
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
```
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 initializer_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)
Algorithms & \( \lambda \)

Re-Cycle instead of Re-Invent the Wheel
for(), while(), do

<algorithm>
<functional>
<numerics>
1. Count the number of non-whitespace characters in standard input
2. Count the number of bytes in standard input (aka \texttt{wc -c})
3. Count the number of (whitespace separated) words in standard input (aka \texttt{wc -w})
4. Count the number of lines in standard input (aka \texttt{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

```cpp
#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.

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

C++11: Simpler typedef aliases using `using`. Using also allows template aliases.
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{0};
    while (std::cin.get(c))
        ++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 << '
';
}
```

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

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

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

```
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

```
auto odd=[](int i)->bool{ return i%2;};
```
create a vector with the integers 1..20 and print a multiplication table (2)

```cpp
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 << '
';
    });
}

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 \(\iota\)
  - 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
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)
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

```cpp
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!
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
### Prohibit instantiation template class

```cpp
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

```cpp
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

```cpp
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;
    }
};
```

- A std::set adapter providing indexed access
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

---

ISO/IEC N3290

There is a partial ordering on cv3qualifiers so that a type can be said to be more cv-qualified than another. Table shows the relations that constitute this ordering.

<table>
<thead>
<tr>
<th>No cv-qualification</th>
<th>cv</th>
<th>const</th>
<th>volatile</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;</td>
<td>&lt;</td>
<td>const</td>
<td>volatile</td>
</tr>
<tr>
<td>const</td>
<td>&lt;</td>
<td>const</td>
<td>volatile</td>
</tr>
<tr>
<td>volatile</td>
<td>&lt;</td>
<td>const</td>
<td>volatile</td>
</tr>
</tbody>
</table>

In this International Standard, the notation cv3sor cv1cv2 etc used in the description of types represents an arbitrary set of cv3qualifiers, i.e., one of {const} {volatile} {const volatile} or the empty set.

Cv3qualifiers applied to an array type attach to the underlying element type, so the notation "cv T" where T is an array type refers to an array whose elements are so3qualified. Such array types can be said to be more or less cv3qualified than other types based on the cv3qualification of the underlying element types.

3.10 Lvalues and rvalues

Expressions are categorized according to the taxonomy in Figure.

- An lvalue is so-called historically because lvalues could appear on the left-hand side of an assignment expression and designates a function or an object.
  - Example: If E is an expression of pointer type, then *E is an lvalue expression referring to the object or function to which E points. As another example, the result of calling a function whose return type is an lvalue reference is an lvalue.

- An xvalue is an "eXpiring" value, also refers to an object, usually near the end of its lifetime, so that its resources may be moved, for example. An xvalue is the result of certain kinds of expressions involving rvalue references.
  - Example: The result of calling a function whose return type is an rvalue reference is an xvalue.

- A glvalue is a "generalized" lvalue, is an lvalue or an xvalue.

- An rvalue is so-called historically because rvalues could appear on the right-hand side of an assignment expression and is an xvalue, a temporary object or subobject thereof or a value that is not associated with an object.

- A prvalue is a "pure" rvalue, is an rvalue that is not an xvalue.
  - Example: The result of calling a function whose return type is not a reference is a prvalue. The value of a literal such as 12, 7.3e5 or true is also a prvalue.

Every expression belongs to exactly one of the fundamental classifications in this taxonomy - lvalue, xvalue, or prvalue. This property of an expression is called its value category.

(Note: The discussion of each built-in...
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

```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?

- http://sconsolidator.com
- peter.sommerlad@hsr.ch http://ifs.hsr.ch

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