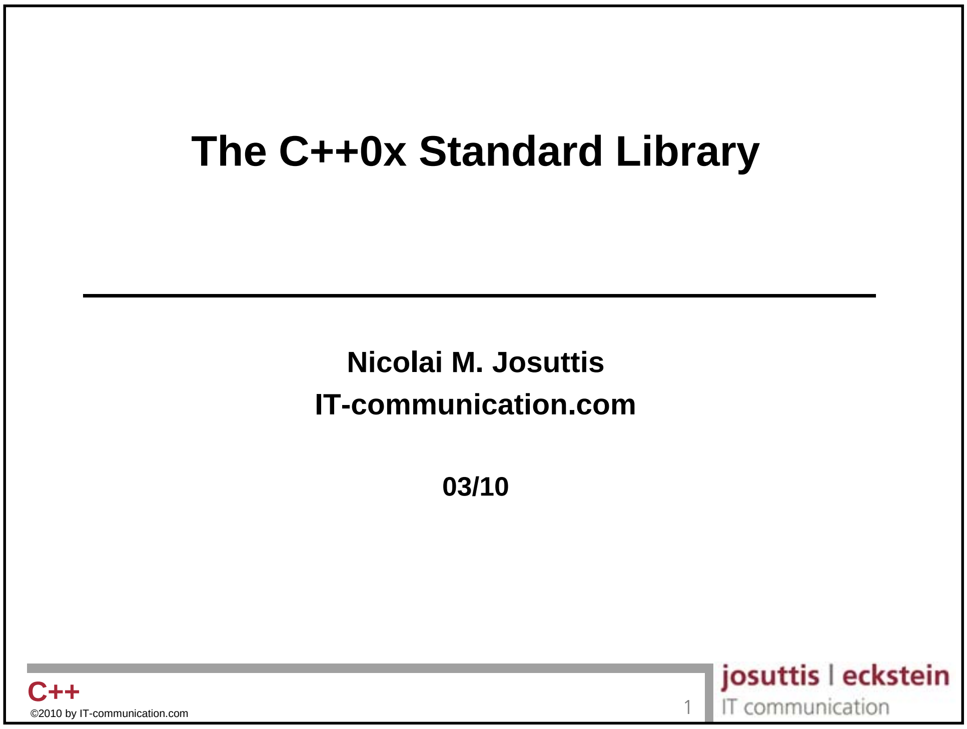
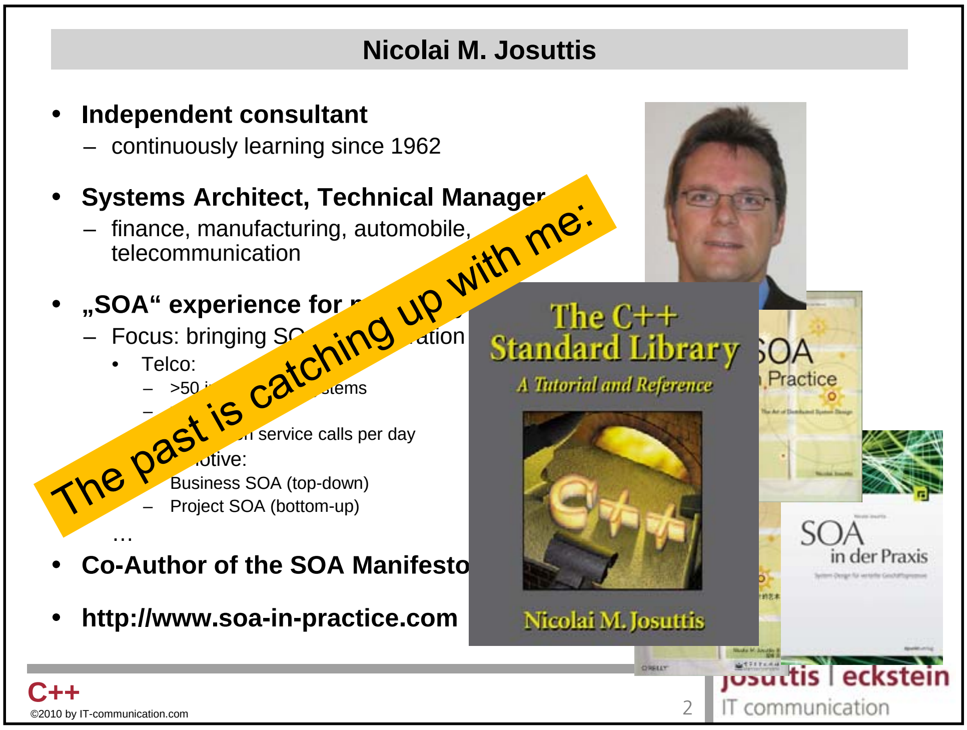


The C++0x Standard Library

Nicolai M. Josuttis
IT-communication.com

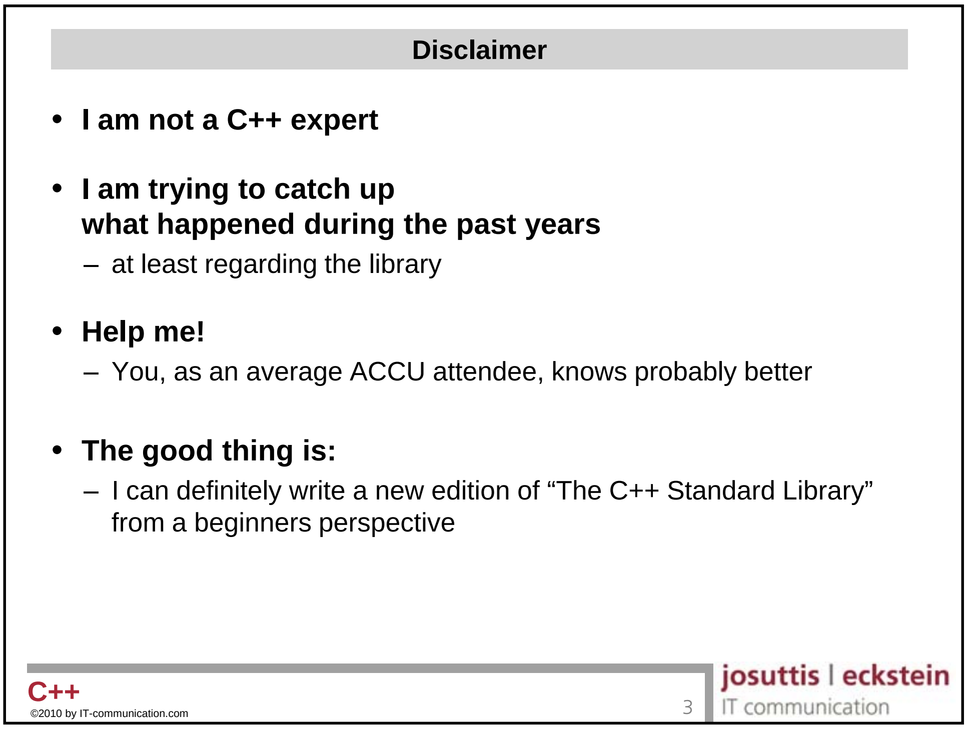
03/10



Nicolai M. Josuttis

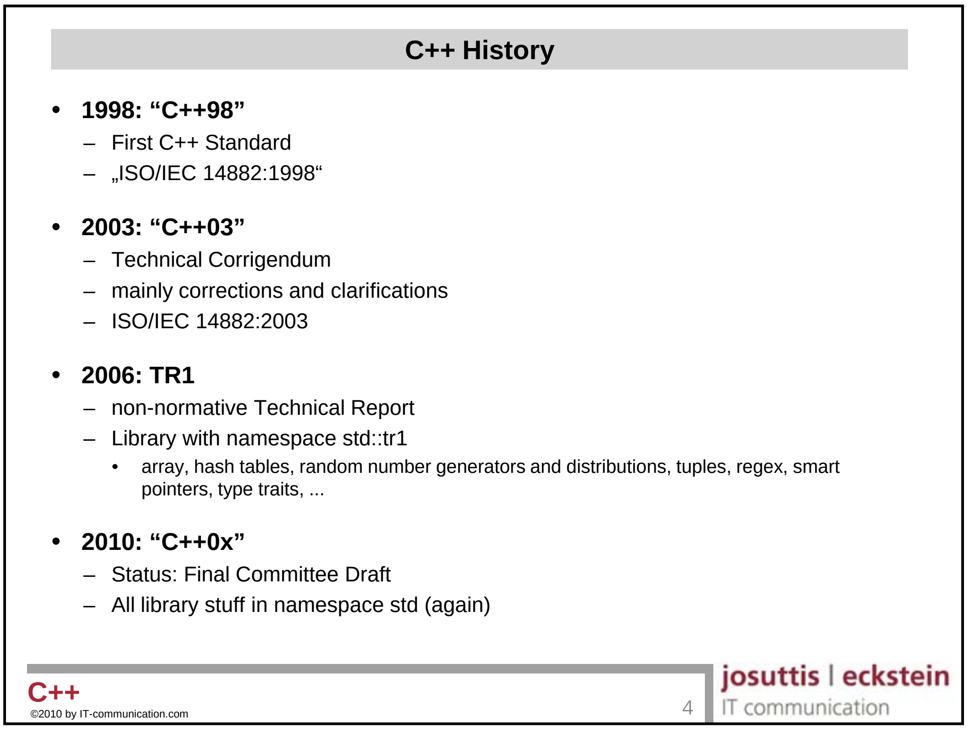
- **Independent consultant**
  - continuously learning since 1962
- **Systems Architect, Technical Manager**
  - finance, manufacturing, automobile, telecommunication
- **„SOA“ experience for r**
  - Focus: bringing SO ation
    - Telco:
      - >50 i stems
      -
      - n service calls per day
    - otive:
      - Business SOA (top-down)
      - Project SOA (bottom-up)
    - …
- **Co-Author of the SOA Manifesto**
- **http://www.soa-in-practice.com**

The past is catching up with me:



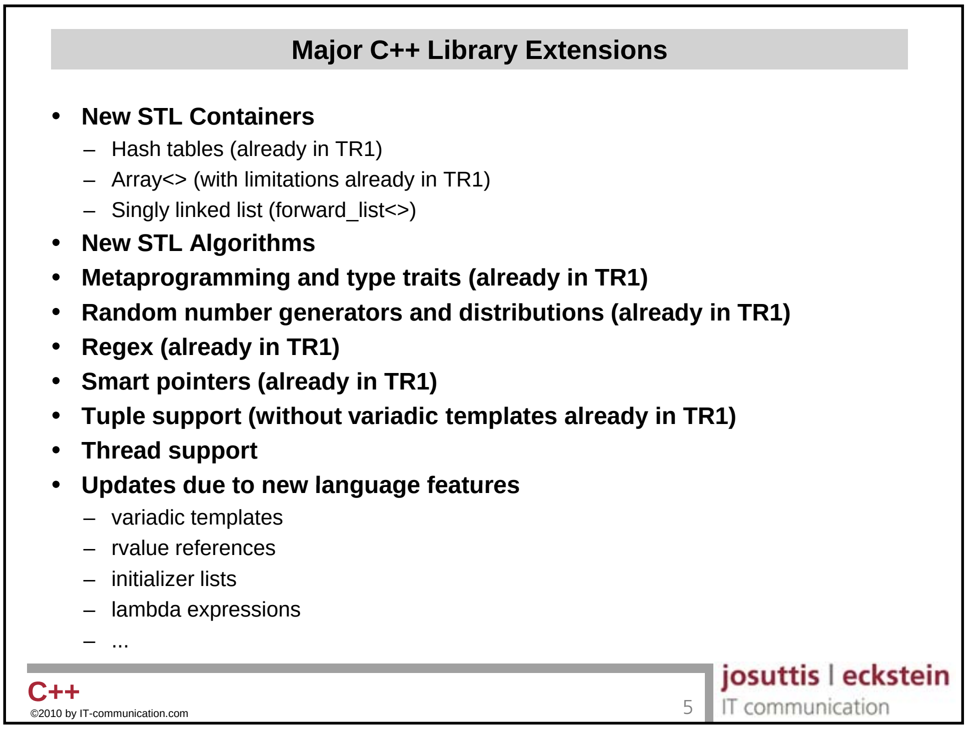
Disclaimer

- **I am not a C++ expert**
- **I am trying to catch up what happened during the past years**
  - at least regarding the library
- **Help me!**
  - You, as an average ACCU attendee, knows probably better
- **The good thing is:**
  - I can definitely write a new edition of “The C++ Standard Library” from a beginners perspective



C++ History

- **1998: “C++98”**
  - First C++ Standard
  - „ISO/IEC 14882:1998“
- **2003: “C++03”**
  - Technical Corrigendum
  - mainly corrections and clarifications
  - ISO/IEC 14882:2003
- **2006: TR1**
  - non-normative Technical Report
  - Library with namespace std::tr1
    - array, hash tables, random number generators and distributions, tuples, regex, smart pointers, type traits, ...
- **2010: “C++0x”**
  - Status: Final Committee Draft
  - All library stuff in namespace std (again)

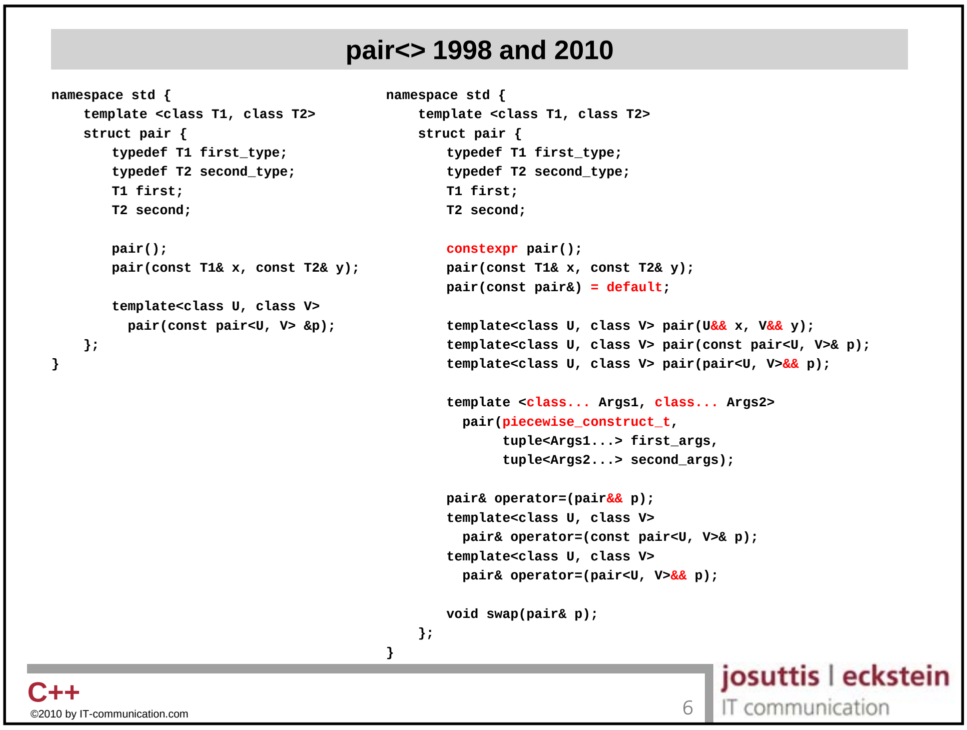


Major C++ Library Extensions

- **New STL Containers**
  - Hash tables (already in TR1)
  - Array<> (with limitations already in TR1)
  - Singly linked list (forward_list<>)
- **New STL Algorithms**
- **Metaprogramming and type traits (already in TR1)**
- **Random number generators and distributions (already in TR1)**
- **Regex (already in TR1)**
- **Smart pointers (already in TR1)**
- **Tuple support (without variadic templates already in TR1)**
- **Thread support**
- **Updates due to new language features**
  - variadic templates
  - rvalue references
  - initializer lists
  - lambda expressions
  - ...

C++

josuttis | eckstein
IT communication



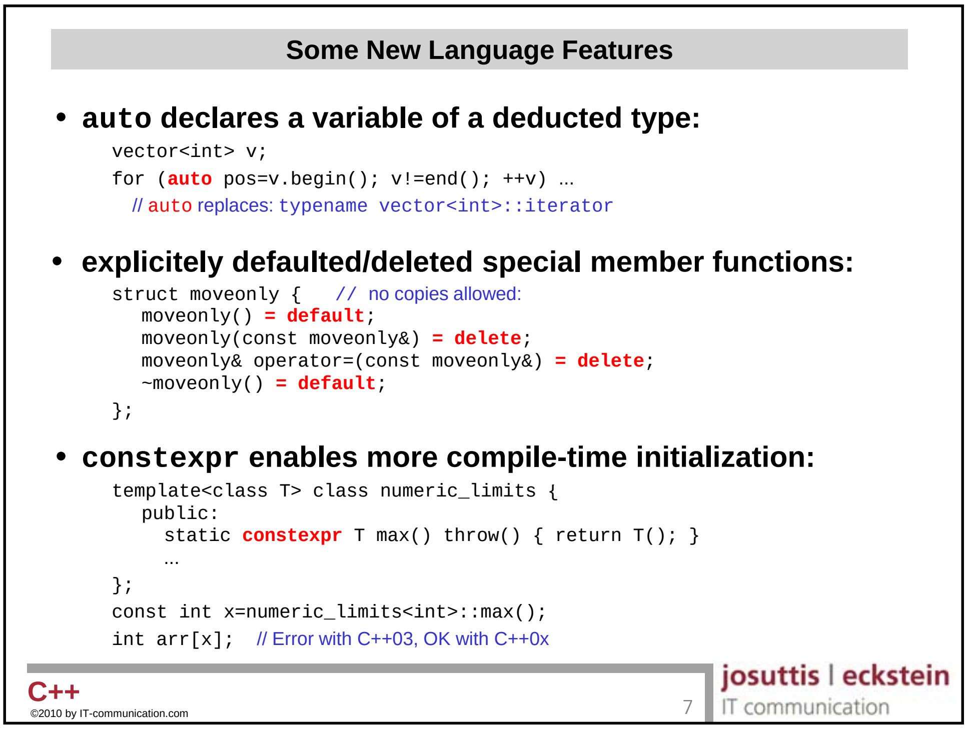
pair<> 1998 and 2010

```
namespace std {
    template <class T1, class T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;

        pair();
        pair(const T1& x, const T2& y);

        template<class U, class V>
          pair(const pair<U, V> &p);
    };
}
```

```
namespace std {
    template <class T1, class T2>
    struct pair {
        typedef T1 first_type;
        typedef T2 second_type;
        T1 first;
        T2 second;

        constexpr pair();
        pair(const T1& x, const T2& y);
        pair(const pair&) = default;

        template<class U, class V> pair(U&& x, V&& y);
        template<class U, class V> pair(const pair<U, V>& p);
        template<class U, class V> pair(pair<U, V>&& p);

        template <class... Args1, class... Args2>
          pair(piecewise_construct_t,
               tuple<Args1...> first_args,
               tuple<Args2...> second_args);

        pair& operator=(pair&& p);
        template<class U, class V>
          pair& operator=(const pair<U, V>& p);
        template<class U, class V>
          pair& operator=(pair<U, V>&& p);

        void swap(pair& p);
    };
}
```

C++

josuttis | eckstein
IT communication



Some New Language Features

- **`auto` declares a variable of a deducted type:**

```
vector<int> v;
for (auto pos=v.begin(); v!=end(); ++v) …
  // auto replaces: typename vector<int>::iterator
```

- **explicitely defaulted/deleted special member functions:**

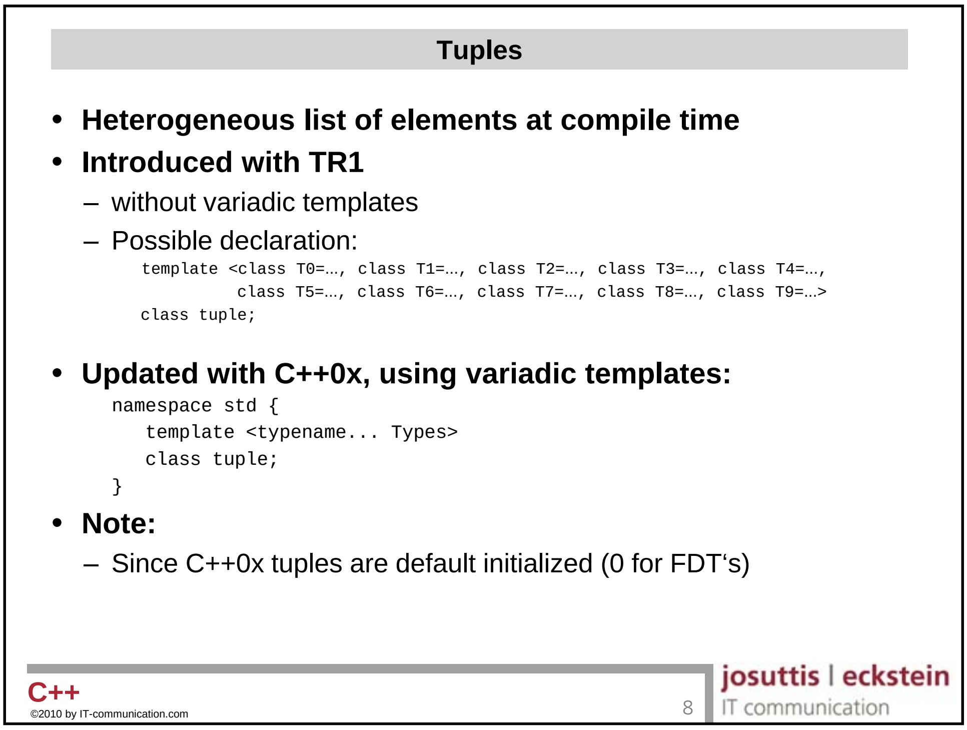
```
struct moveonly {    // no copies allowed:
   moveonly() = default;
   moveonly(const moveonly&) = delete;
   moveonly& operator=(const moveonly&) = delete;
   ~moveonly() = default;
};
```

- **`constexpr` enables more compile-time initialization:**

```
template<class T> class numeric_limits {
   public:
     static constexpr T max() throw() { return T(); }
     …
};
const int x=numeric_limits<int>::max();
int arr[x];   // Error with C++03, OK with C++0x
```

C++

josuttis | eckstein
IT communication



Tuples

- **Heterogeneous list of elements at compile time**
- **Introduced with TR1**
  - without variadic templates
  - Possible declaration:

```
template <class T0=…, class T1=…, class T2=…, class T3=…, class T4=…,
             class T5=…, class T6=…, class T7=…, class T8=…, class T9=…>
class tuple;
```

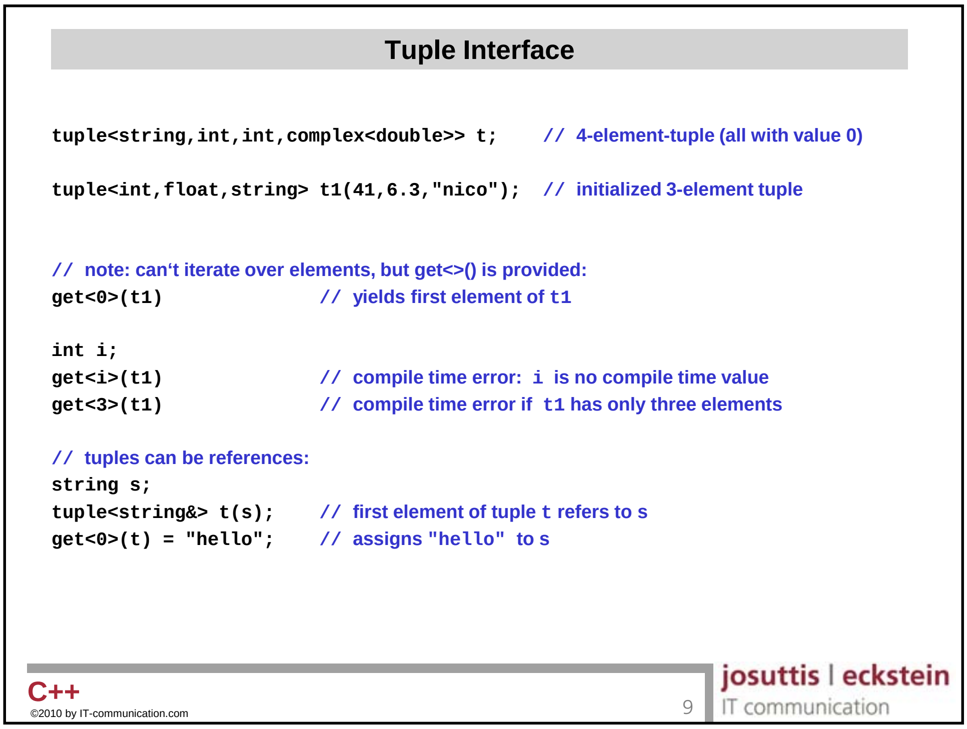
- **Updated with C++0x, using variadic templates:**

```
namespace std {
    template <typename... Types>
    class tuple;
}
```

- **Note:**
  - Since C++0x tuples are default initialized (0 for FDT‘s)

C++

josuttis | eckstein
IT communication



Tuple Interface

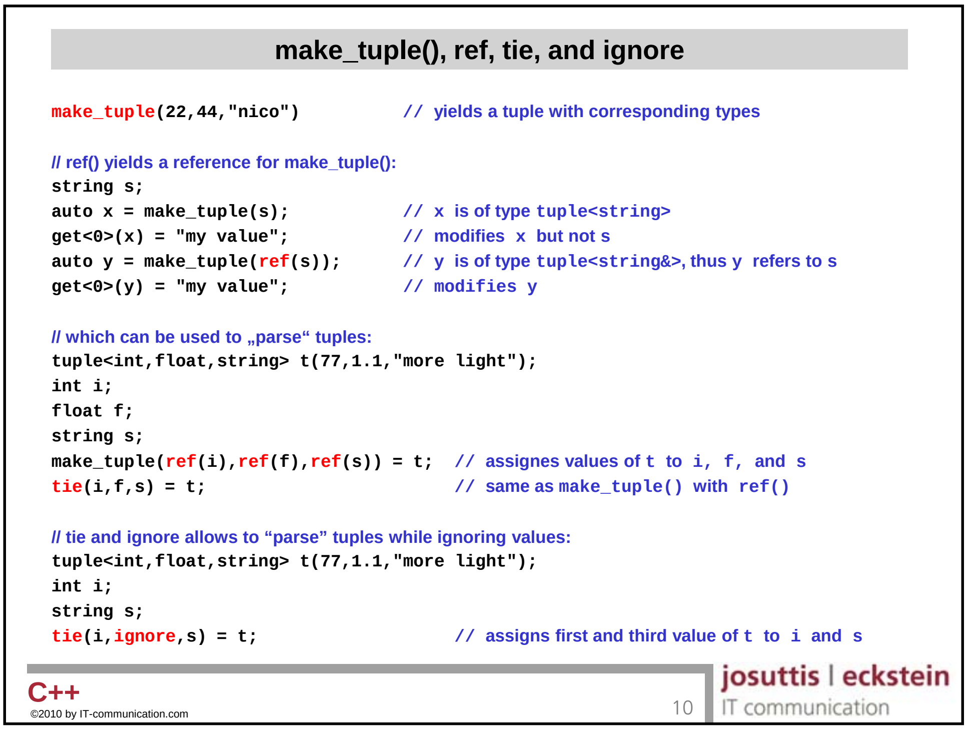
```
tuple<string,int,int,complex<double>> t;     // 4-element-tuple (all with value 0)

tuple<int,float,string> t1(41,6.3,"nico");   // initialized 3-element tuple


// note: can‘t iterate over elements, but get<>() is provided:
get<0>(t1)                 // yields first element of t1

int i;
get<i>(t1)                 // compile time error:  i  is no compile time value
get<3>(t1)                 // compile time error if  t1 has only three elements

// tuples can be references:
string s;
tuple<string&> t(s);       // first element of tuple t refers to s
get<0>(t) = "hello";       // assigns "hello"  to s
```

C++

9

josuttis | eckstein

IT communication



make_tuple(), ref, tie, and ignore

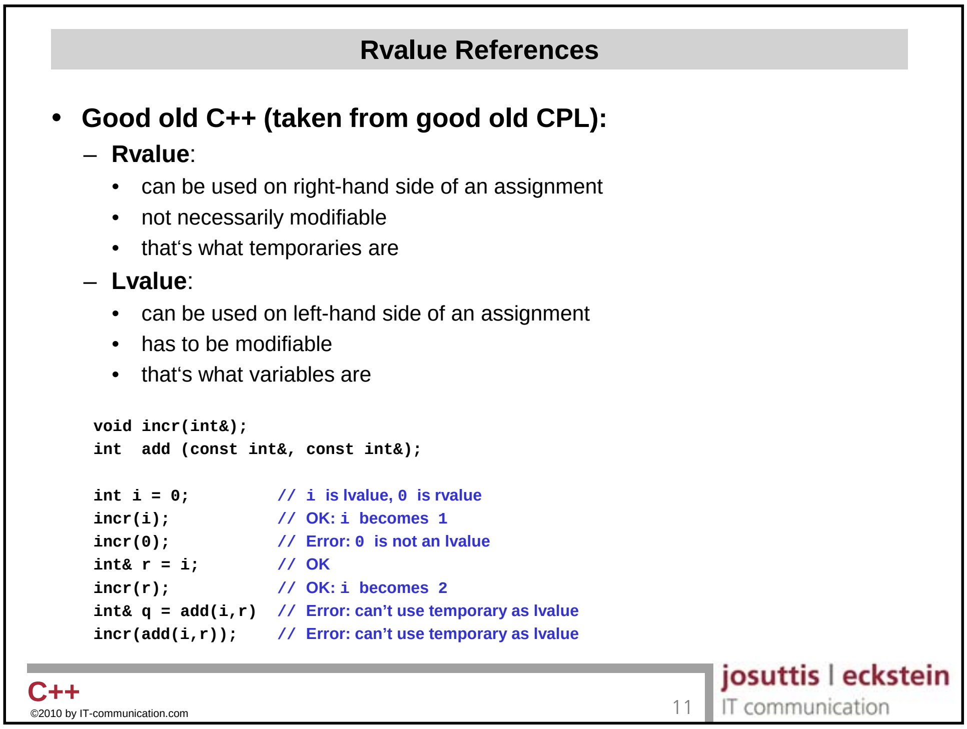
```
make_tuple(22,44,"nico")          // yields a tuple with corresponding types

// ref() yields a reference for make_tuple():
string s;
auto x = make_tuple(s);           // x  is of type tuple<string>
get<0>(x) = "my value";           // modifies  x  but not s
auto y = make_tuple(ref(s));      // y  is of type tuple<string&>, thus y  refers to s
get<0>(y) = "my value";           // modifies  y

// which can be used to „parse“ tuples:
tuple<int,float,string> t(77,1.1,"more light");
int i;
float f;
string s;
make_tuple(ref(i),ref(f),ref(s)) = t;  // assignes values of t  to  i,  f,  and  s
tie(i,f,s) = t;                        // same as make_tuple() with  ref()

// tie and ignore allows to “parse” tuples while ignoring values:
tuple<int,float,string> t(77,1.1,"more light");
int i;
string s;
tie(i,ignore,s) = t;                   // assigns first and third value of t  to  i  and  s
```

C++

10

josuttis | eckstein

IT communication



Rvalue References

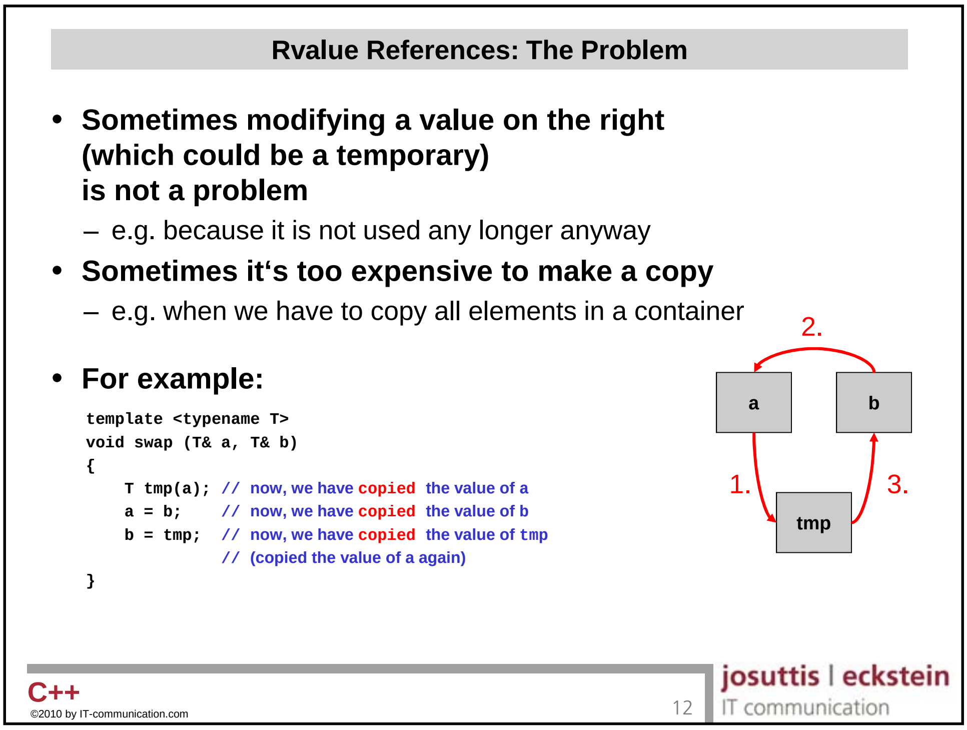
- **Good old C++ (taken from good old CPL):**
  - **Rvalue**:
    - can be used on right-hand side of an assignment
    - not necessarily modifiable
    - that‘s what temporaries are
  - **Lvalue**:
    - can be used on left-hand side of an assignment
    - has to be modifiable
    - that‘s what variables are

```
void incr(int&);
int  add (const int&, const int&);

int i = 0;          //  i  is lvalue, 0  is rvalue
incr(i);            //  OK: i  becomes  1
incr(0);            //  Error: 0  is not an lvalue
int& r = i;         //  OK
incr(r);            //  OK: i  becomes  2
int& q = add(i,r)   //  Error: can’t use temporary as lvalue
incr(add(i,r));     //  Error: can’t use temporary as lvalue
```

C++

11

josuttis | eckstein
IT communication



Rvalue References: The Problem

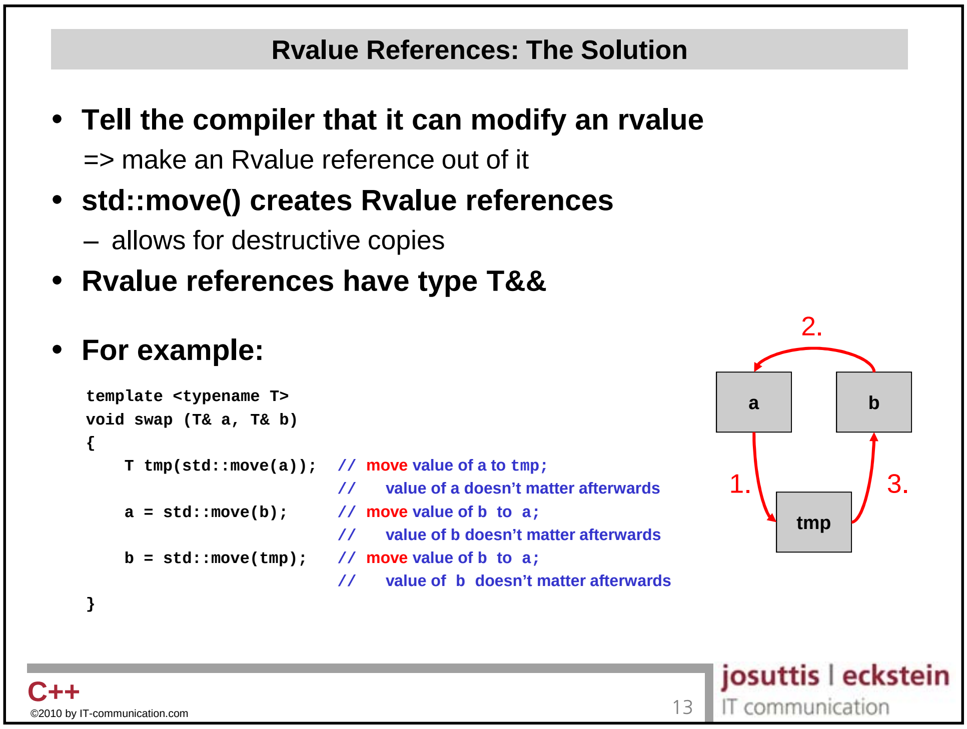
- **Sometimes modifying a value on the right (which could be a temporary) is not a problem**
  - e.g. because it is not used any longer anyway
- **Sometimes it‘s too expensive to make a copy**
  - e.g. when we have to copy all elements in a container
- **For example:**

```
template <typename T>
void swap (T& a, T& b)
{
    T tmp(a); // now, we have copied  the value of a
    a = b;    // now, we have copied  the value of b
    b = tmp;  // now, we have copied  the value of tmp
              // (copied the value of a again)
}
```

C++

12

josuttis | eckstein
IT communication



Rvalue References: The Solution

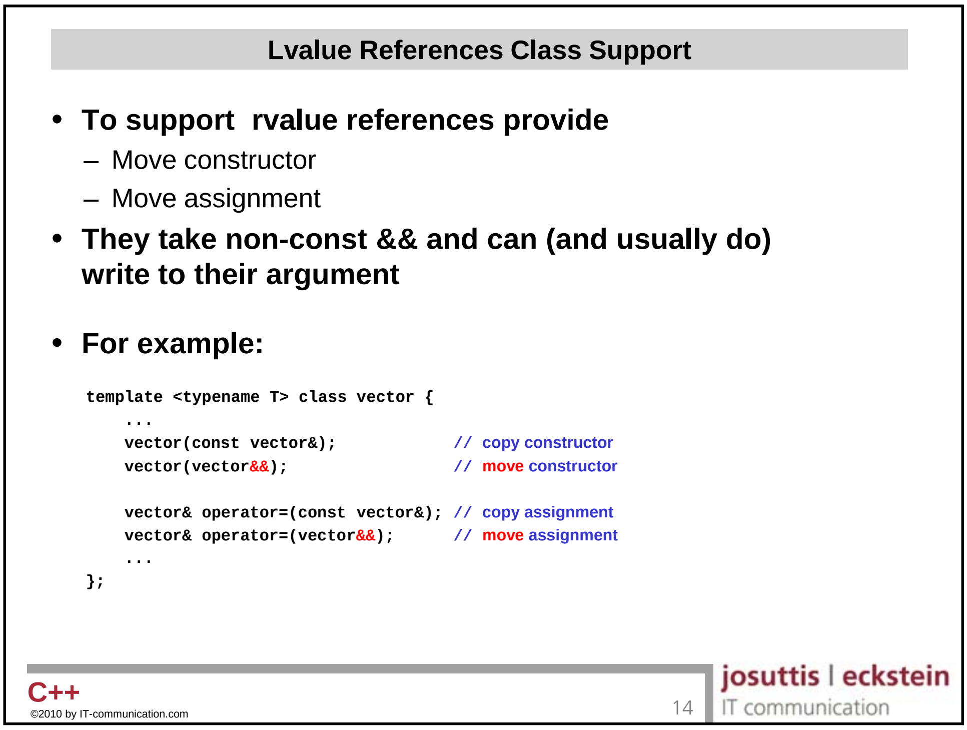
- **Tell the compiler that it can modify an rvalue**
  => make an Rvalue reference out of it
- **std::move() creates Rvalue references**
  – allows for destructive copies
- **Rvalue references have type T&&**

- **For example:**

```
template <typename T>
void swap (T& a, T& b)
{
    T tmp(std::move(a));  // move value of a to tmp;
                          //    value of a doesn't matter afterwards
    a = std::move(b);     // move value of b  to  a;
                          //    value of b doesn't matter afterwards
    b = std::move(tmp);   // move value of b  to  a;
                          //    value of  b  doesn't matter afterwards
}
```

C++

josuttis | eckstein
IT communication



Lvalue References Class Support

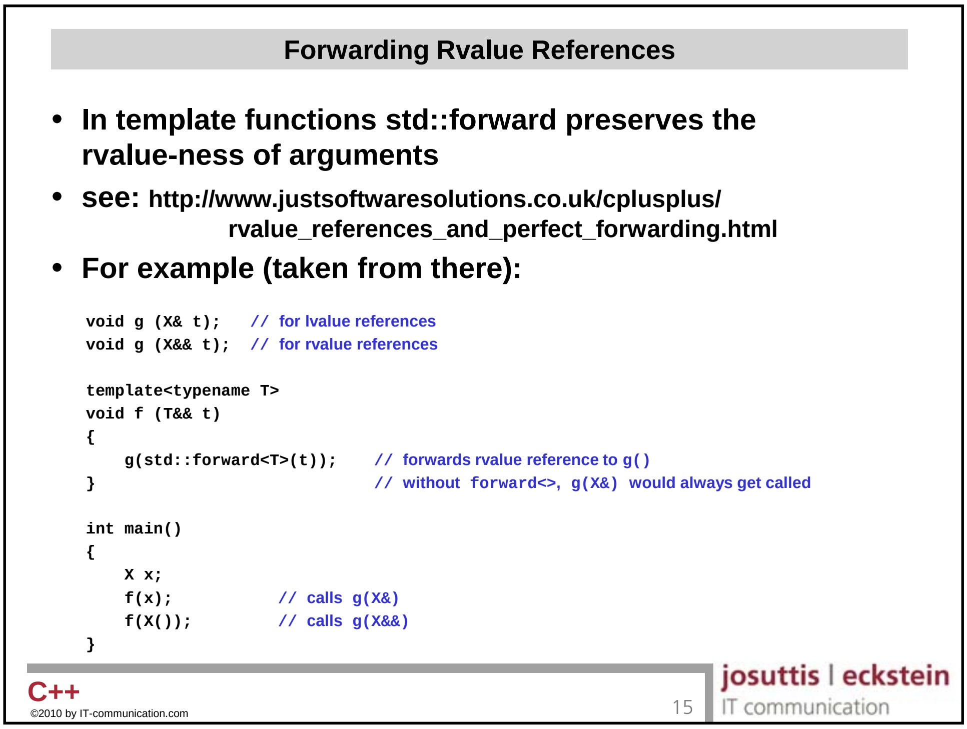
- **To support rvalue references provide**
  – Move constructor
  – Move assignment
- **They take non-const && and can (and usually do) write to their argument**

- **For example:**

```
template <typename T> class vector {
    ...
    vector(const vector&);             // copy constructor
    vector(vector&&);                  // move constructor

    vector& operator=(const vector&);  // copy assignment
    vector& operator=(vector&&);       // move assignment
    ...
};
```

C++

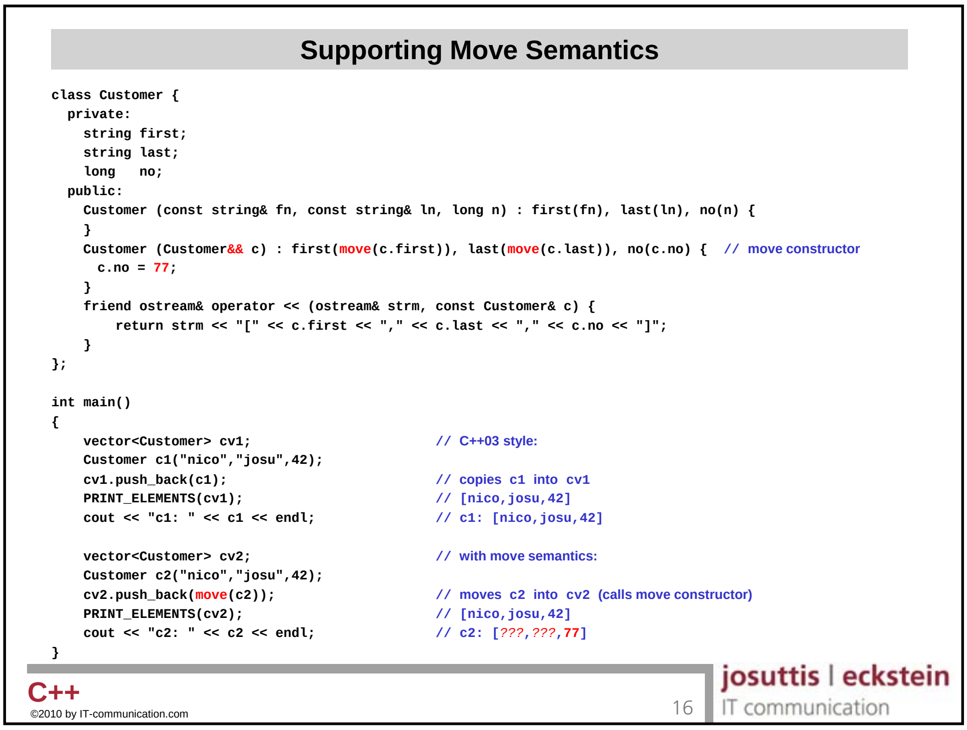
josuttis | eckstein
IT communication



Forwarding Rvalue References

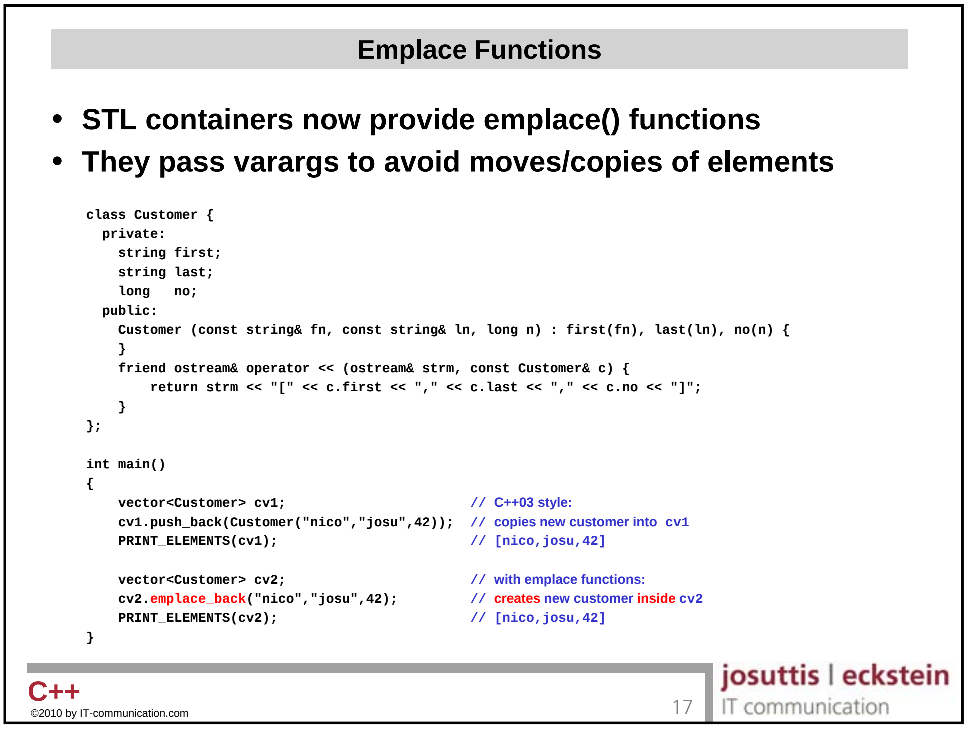
- In template functions std::forward preserves the rvalue-ness of arguments
- see: http://www.justsoftwaresolutions.co.uk/cplusplus/ rvalue_references_and_perfect_forwarding.html
- For example (taken from there):

```
void g (X& t);   // for lvalue references
void g (X&& t);  // for rvalue references

template<typename T>
void f (T&& t)
{
    g(std::forward<T>(t));    // forwards rvalue reference to g()
}                             // without forward<>, g(X&) would always get called

int main()
{
    X x;
    f(x);          // calls g(X&)
    f(X());        // calls g(X&&)
}
```



Supporting Move Semantics

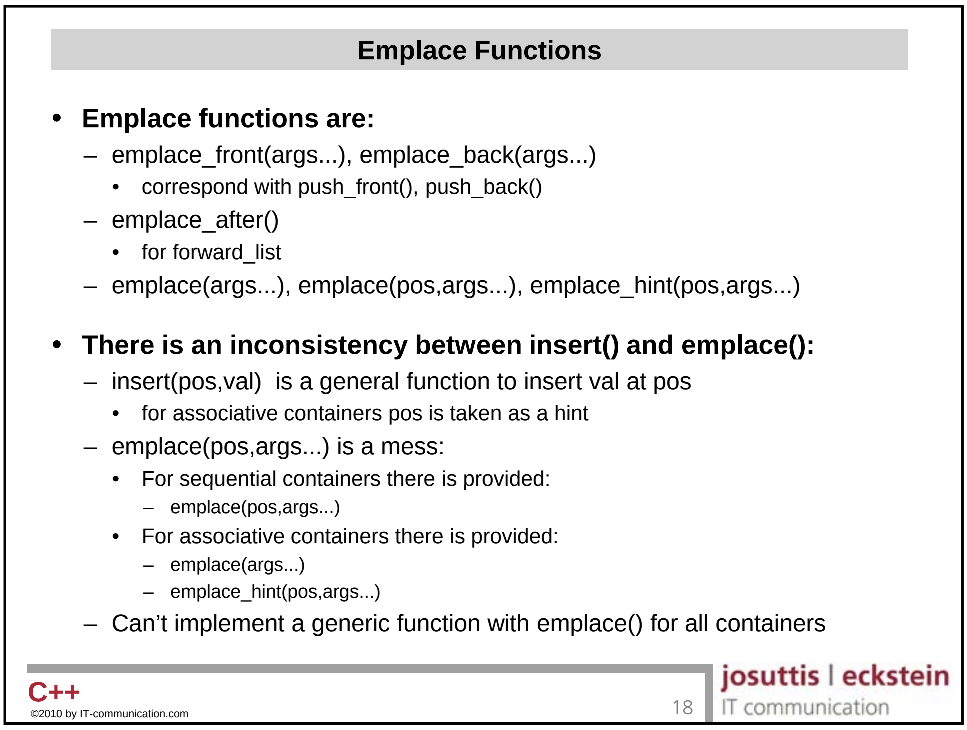
```
class Customer {
  private:
    string first;
    string last;
    long   no;
  public:
    Customer (const string& fn, const string& ln, long n) : first(fn), last(ln), no(n) {
    }
    Customer (Customer&& c) : first(move(c.first)), last(move(c.last)), no(c.no) {  // move constructor
      c.no = 77;
    }
    friend ostream& operator << (ostream& strm, const Customer& c) {
        return strm << "[" << c.first << "," << c.last << "," << c.no << "]";
    }
};

int main()
{
    vector<Customer> cv1;                         // C++03 style:
    Customer c1("nico","josu",42);
    cv1.push_back(c1);                            // copies c1 into cv1
    PRINT_ELEMENTS(cv1);                          // [nico,josu,42]
    cout << "c1: " << c1 << endl;                 // c1: [nico,josu,42]

    vector<Customer> cv2;                         // with move semantics:
    Customer c2("nico","josu",42);
    cv2.push_back(move(c2));                      // moves c2 into cv2 (calls move constructor)
    PRINT_ELEMENTS(cv2);                          // [nico,josu,42]
    cout << "c2: " << c2 << endl;                 // c2: [???,???,77]
}
```



Emplace Functions

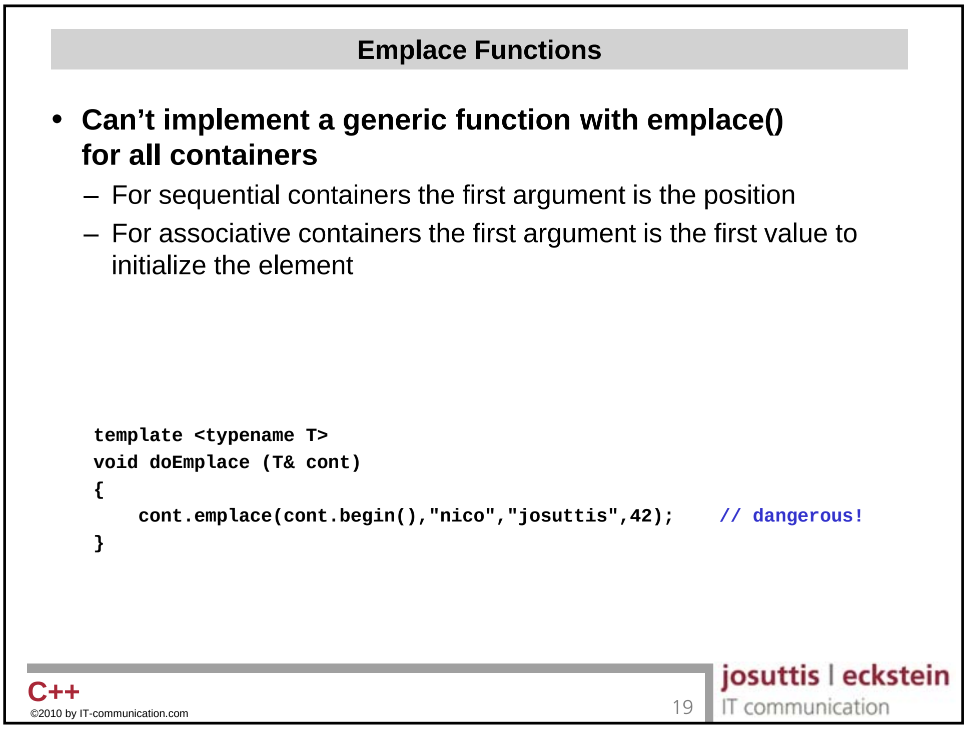
- **STL containers now provide emplace() functions**
- **They pass varargs to avoid moves/copies of elements**

```
class Customer {
  private:
    string first;
    string last;
    long   no;
  public:
    Customer (const string& fn, const string& ln, long n) : first(fn), last(ln), no(n) {
    }
    friend ostream& operator << (ostream& strm, const Customer& c) {
        return strm << "[" << c.first << "," << c.last << "," << c.no << "]";
    }
};

int main()
{
    vector<Customer> cv1;                          // C++03 style:
    cv1.push_back(Customer("nico","josu",42));  // copies new customer into  cv1
    PRINT_ELEMENTS(cv1);                           // [nico,josu,42]

    vector<Customer> cv2;                          // with emplace functions:
    cv2.emplace_back("nico","josu",42);           // creates new customer inside cv2
    PRINT_ELEMENTS(cv2);                           // [nico,josu,42]
}
```



Emplace Functions

- **Emplace functions are:**
  - emplace_front(args...), emplace_back(args...)
    - correspond with push_front(), push_back()
  - emplace_after()
    - for forward_list
  - emplace(args...), emplace(pos,args...), emplace_hint(pos,args...)
- **There is an inconsistency between insert() and emplace():**
  - insert(pos,val) is a general function to insert val at pos
    - for associative containers pos is taken as a hint
  - emplace(pos,args...) is a mess:
    - For sequential containers there is provided:
      - emplace(pos,args...)
    - For associative containers there is provided:
      - emplace(args...)
      - emplace_hint(pos,args...)
  - Can’t implement a generic function with emplace() for all containers

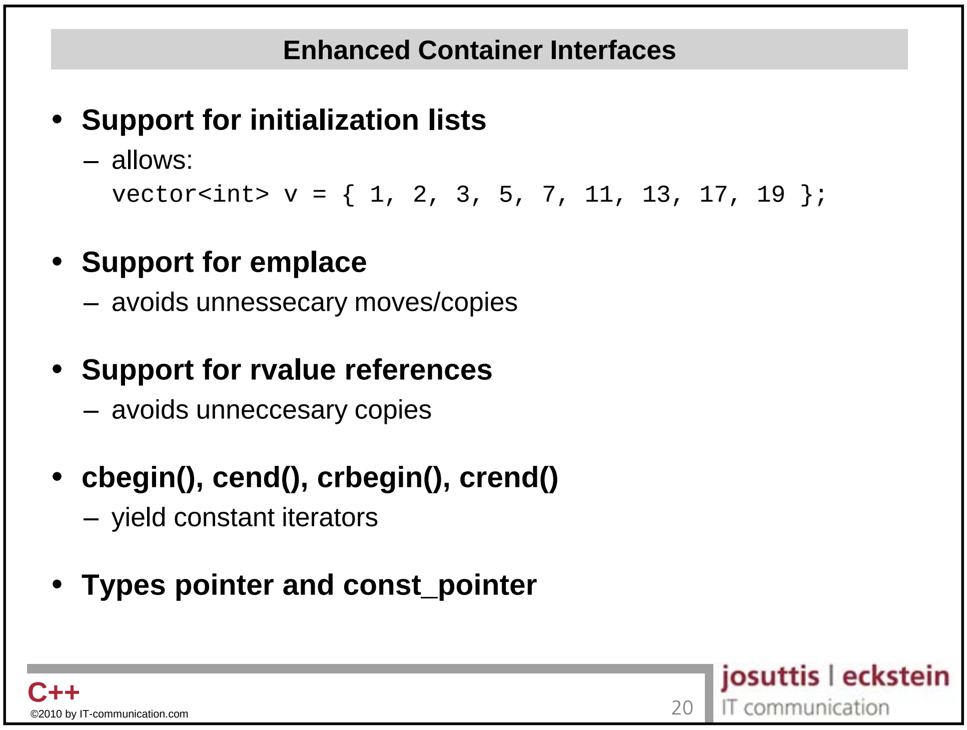


Emplace Functions

- **Can’t implement a generic function with emplace() for all containers**
  - For sequential containers the first argument is the position
  - For associative containers the first argument is the first value to initialize the element

```
template <typename T>
void doEmplace (T& cont)
{
    cont.emplace(cont.begin(),"nico","josuttis",42);    // dangerous!
}
```

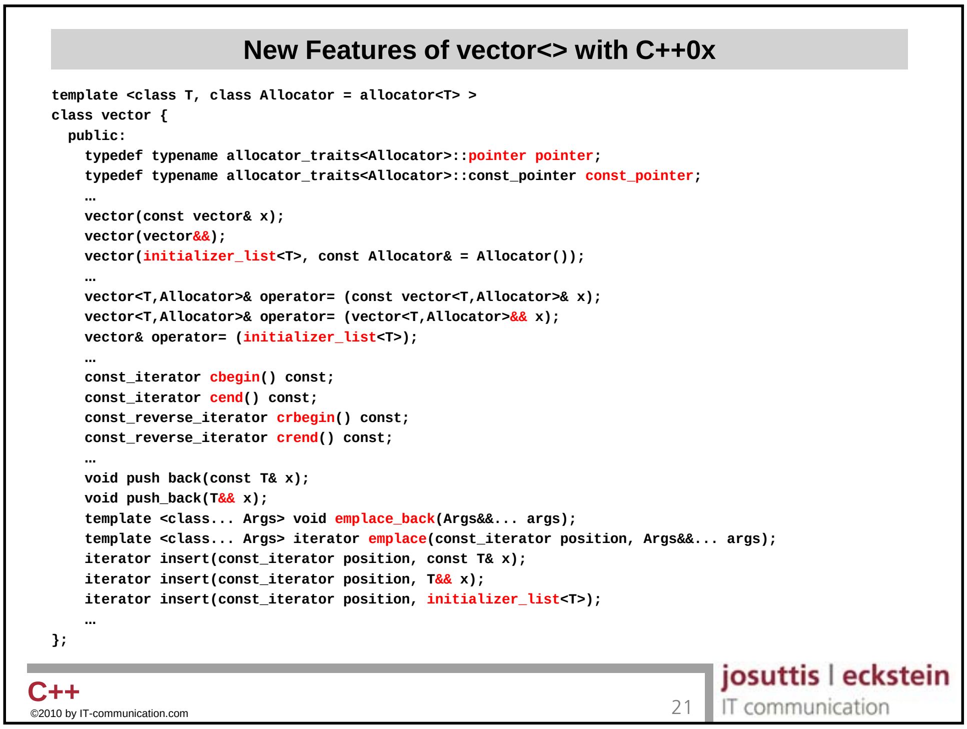
C++
©2010 by IT-communication.com 19
josuttis | eckstein
IT communication



Enhanced Container Interfaces

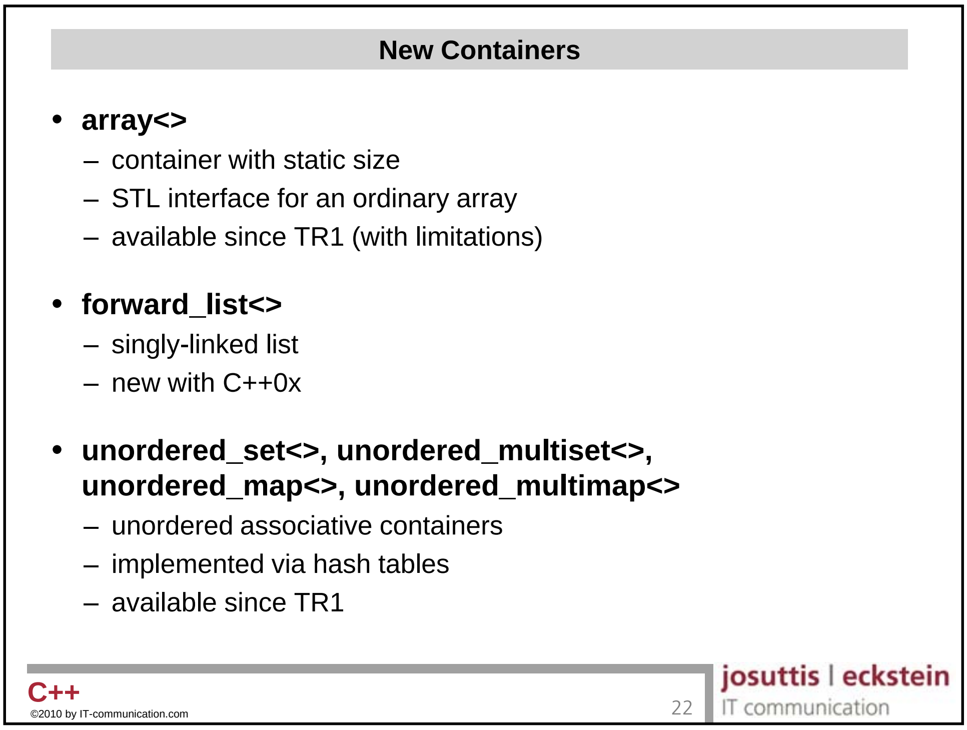
- **Support for initialization lists**
  - allows:
    `vector<int> v = { 1, 2, 3, 5, 7, 11, 13, 17, 19 };`
- **Support for emplace**
  - avoids unnessecary moves/copies
- **Support for rvalue references**
  - avoids unneccesary copies
- **cbegin(), cend(), crbegin(), crend()**
  - yield constant iterators
- **Types pointer and const_pointer**

C++
©2010 by IT-communication.com 20
josuttis | eckstein
IT communication



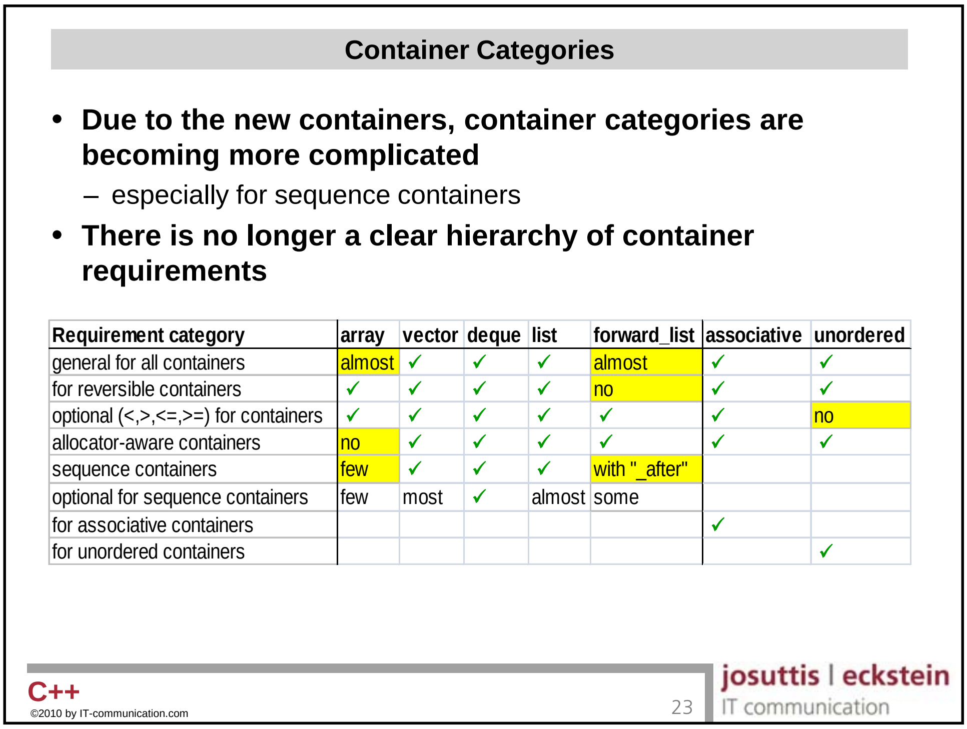
New Features of vector<> with C++0x

```
template <class T, class Allocator = allocator<T> >
class vector {
  public:
    typedef typename allocator_traits<Allocator>::pointer pointer;
    typedef typename allocator_traits<Allocator>::const_pointer const_pointer;
    …
    vector(const vector& x);
    vector(vector&&);
    vector(initializer_list<T>, const Allocator& = Allocator());
    …
    vector<T,Allocator>& operator= (const vector<T,Allocator>& x);
    vector<T,Allocator>& operator= (vector<T,Allocator>&& x);
    vector& operator= (initializer_list<T>);
    …
    const_iterator cbegin() const;
    const_iterator cend() const;
    const_reverse_iterator crbegin() const;
    const_reverse_iterator crend() const;
    …
    void push back(const T& x);
    void push_back(T&& x);
    template <class... Args> void emplace_back(Args&&... args);
    template <class... Args> iterator emplace(const_iterator position, Args&&... args);
    iterator insert(const_iterator position, const T& x);
    iterator insert(const_iterator position, T&& x);
    iterator insert(const_iterator position, initializer_list<T>);
    …
};
```



New Containers

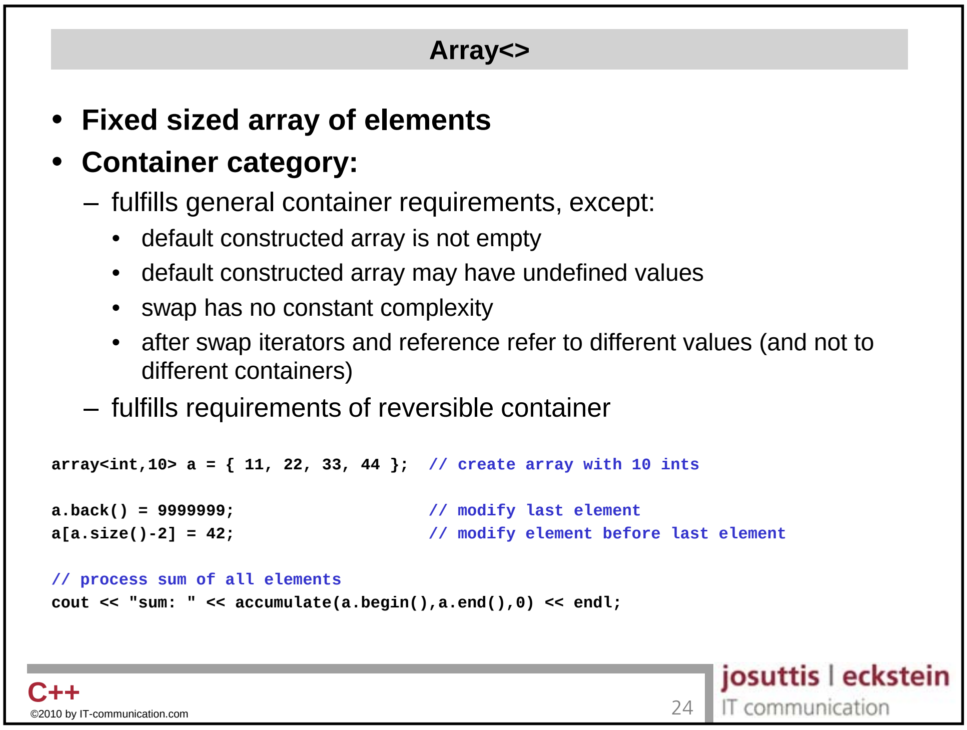
- **array<>**
  - container with static size
  - STL interface for an ordinary array
  - available since TR1 (with limitations)
- **forward_list<>**
  - singly-linked list
  - new with C++0x
- **unordered_set<>, unordered_multiset<>, unordered_map<>, unordered_multimap<>**
  - unordered associative containers
  - implemented via hash tables
  - available since TR1



Container Categories

- **Due to the new containers, container categories are becoming more complicated**
  - especially for sequence containers
- **There is no longer a clear hierarchy of container requirements**

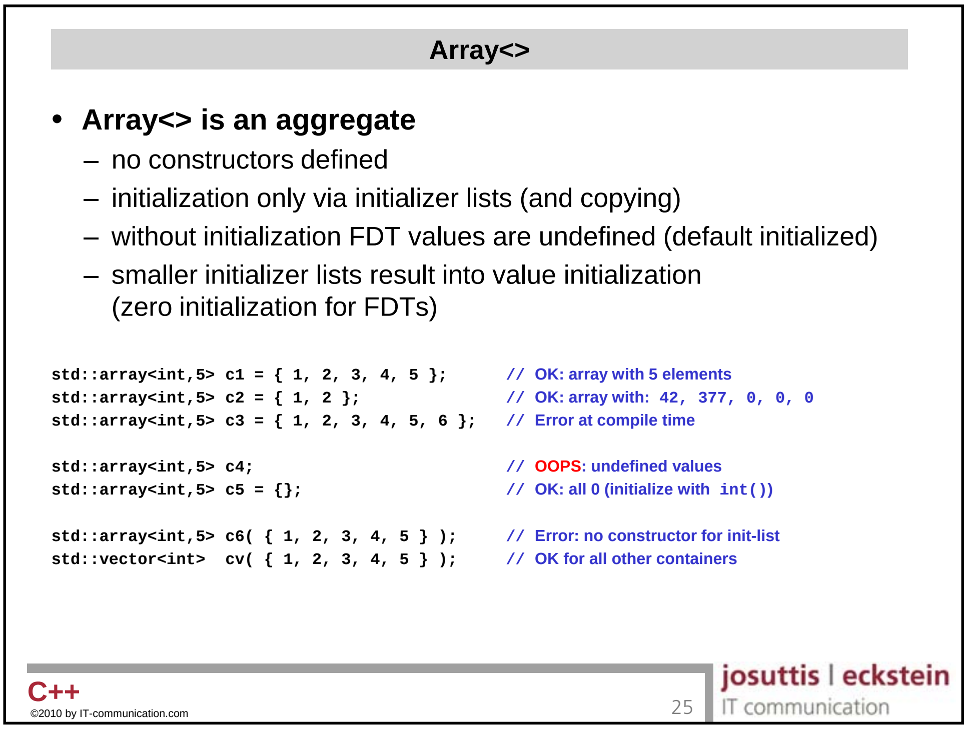
Requirement category	array	vector	deque	list	forward_list	associative	unordered
general for all containers	almost	✓	✓	✓	almost	✓	✓
for reversible containers	✓	✓	✓	✓	no	✓	✓
optional (<,>,<=,>=) for containers	✓	✓	✓	✓	✓	✓	no
allocator-aware containers	no	✓	✓	✓	✓	✓	✓
sequence containers	few	✓	✓	✓	with "_after"		
optional for sequence containers	few	most	✓	almost	some		
for associative containers						✓	
for unordered containers							✓



Array<>

- **Fixed sized array of elements**
- **Container category:**
  - fulfills general container requirements, except:
    - default constructed array is not empty
    - default constructed array may have undefined values
    - swap has no constant complexity
    - after swap iterators and reference refer to different values (and not to different containers)
  - fulfills requirements of reversible container

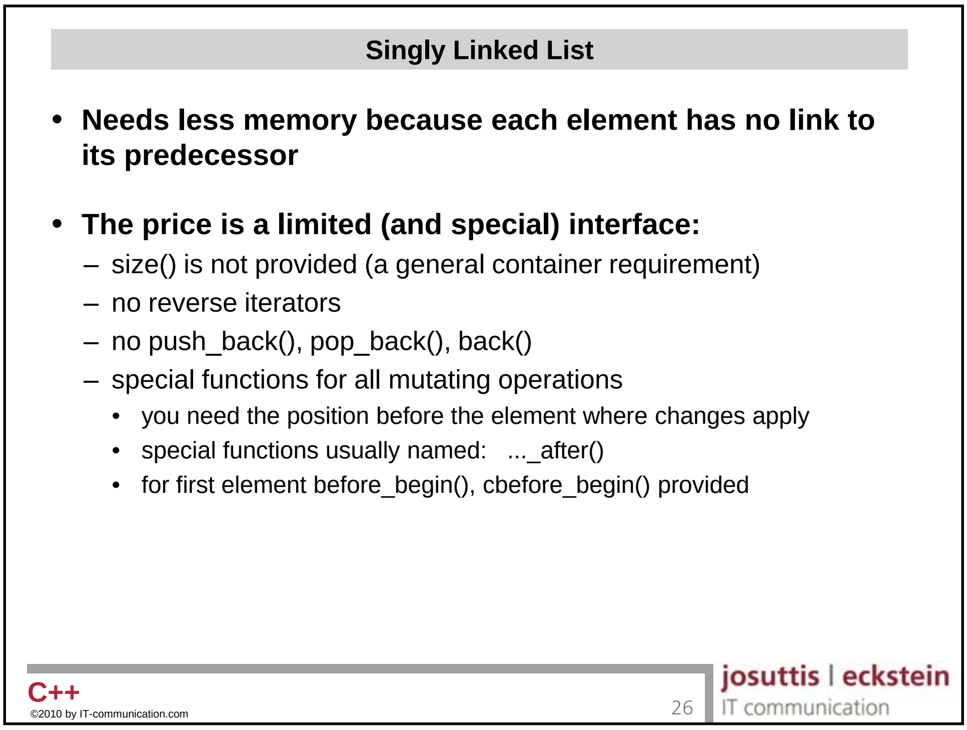
```
array<int,10> a = { 11, 22, 33, 44 };  // create array with 10 ints

a.back() = 9999999;                     // modify last element
a[a.size()-2] = 42;                     // modify element before last element

// process sum of all elements
cout << "sum: " << accumulate(a.begin(),a.end(),0) << endl;
```



Array<>

- **Array<> is an aggregate**
  - no constructors defined
  - initialization only via initializer lists (and copying)
  - without initialization FDT values are undefined (default initialized)
  - smaller initializer lists result into value initialization (zero initialization for FDTs)

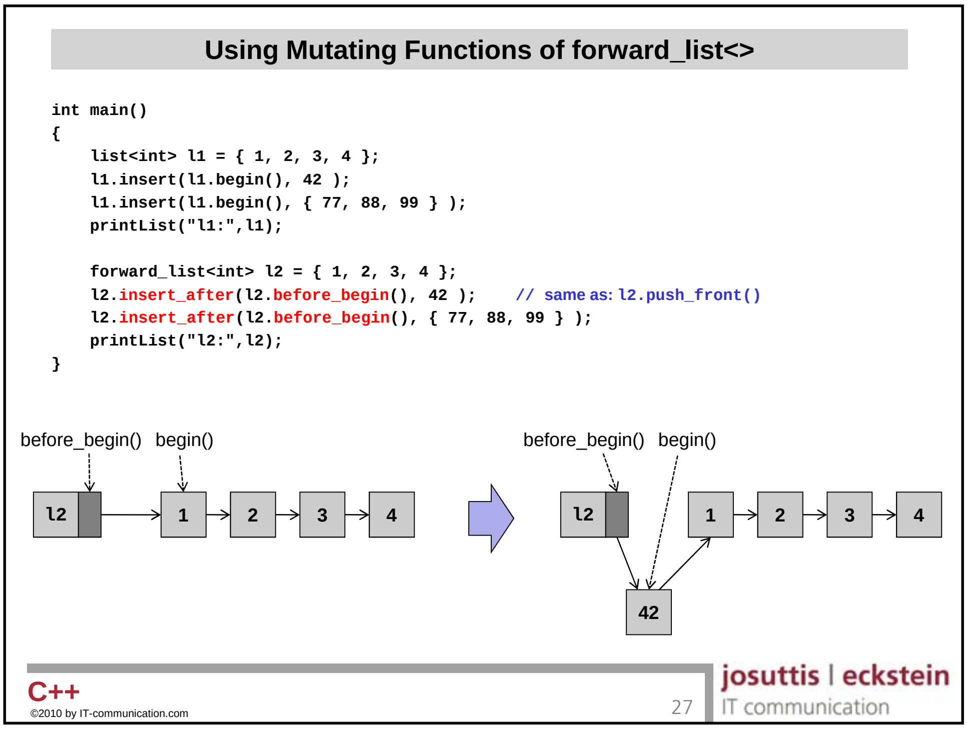
```
std::array<int,5> c1 = { 1, 2, 3, 4, 5 };       // OK: array with 5 elements
std::array<int,5> c2 = { 1, 2 };                 // OK: array with:  42, 377, 0, 0, 0
std::array<int,5> c3 = { 1, 2, 3, 4, 5, 6 };    // Error at compile time

std::array<int,5> c4;                            // OOPS: undefined values
std::array<int,5> c5 = {};                       // OK: all 0 (initialize with  int())

std::array<int,5> c6( { 1, 2, 3, 4, 5 } );      // Error: no constructor for init-list
std::vector<int>  cv( { 1, 2, 3, 4, 5 } );      // OK for all other containers
```



Singly Linked List

- **Needs less memory because each element has no link to its predecessor**

- **The price is a limited (and special) interface:**
  - size() is not provided (a general container requirement)
  - no reverse iterators
  - no push_back(), pop_back(), back()
  - special functions for all mutating operations
    - you need the position before the element where changes apply
    - special functions usually named: ..._after()
    - for first element before_begin(), cbefore_begin() provided



Using Mutating Functions of forward_list<>

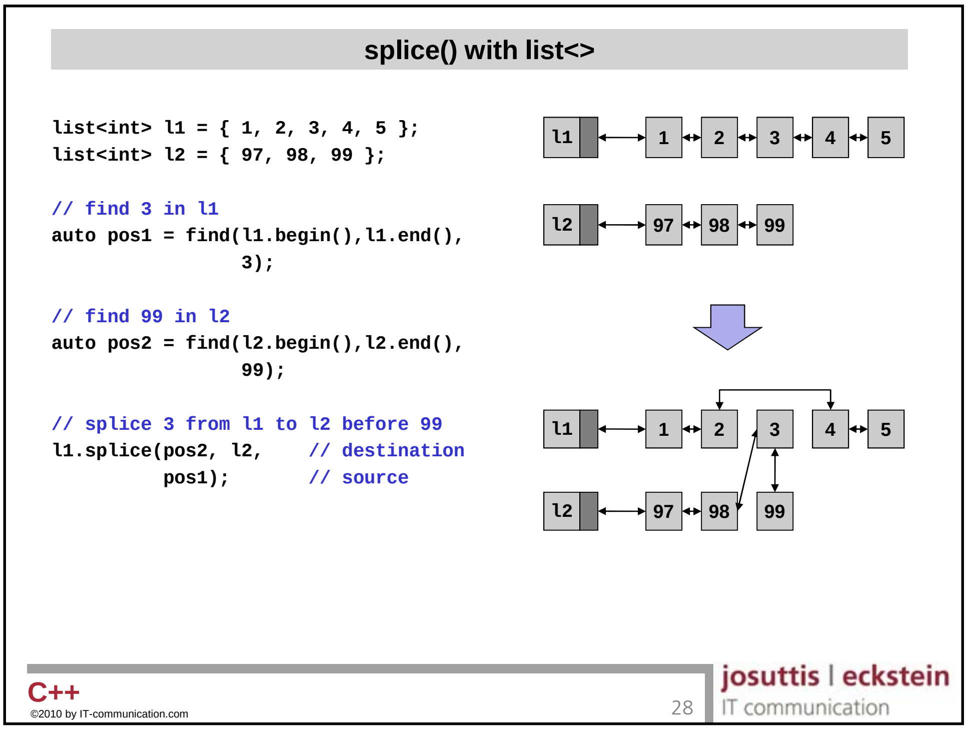
```
int main()
{
    list<int> l1 = { 1, 2, 3, 4 };
    l1.insert(l1.begin(), 42 );
    l1.insert(l1.begin(), { 77, 88, 99 } );
    printList("l1:",l1);

    forward_list<int> l2 = { 1, 2, 3, 4 };
    l2.insert_after(l2.before_begin(), 42 );     // same as: l2.push_front()
    l2.insert_after(l2.before_begin(), { 77, 88, 99 } );
    printList("l2:",l2);
}
```

before_begin()
begin()
l2
1
2
3
4
before_begin()
begin()
l2
1
2
3
4
42

C++

27

josuttis | eckstein
IT communication



splice() with list<>

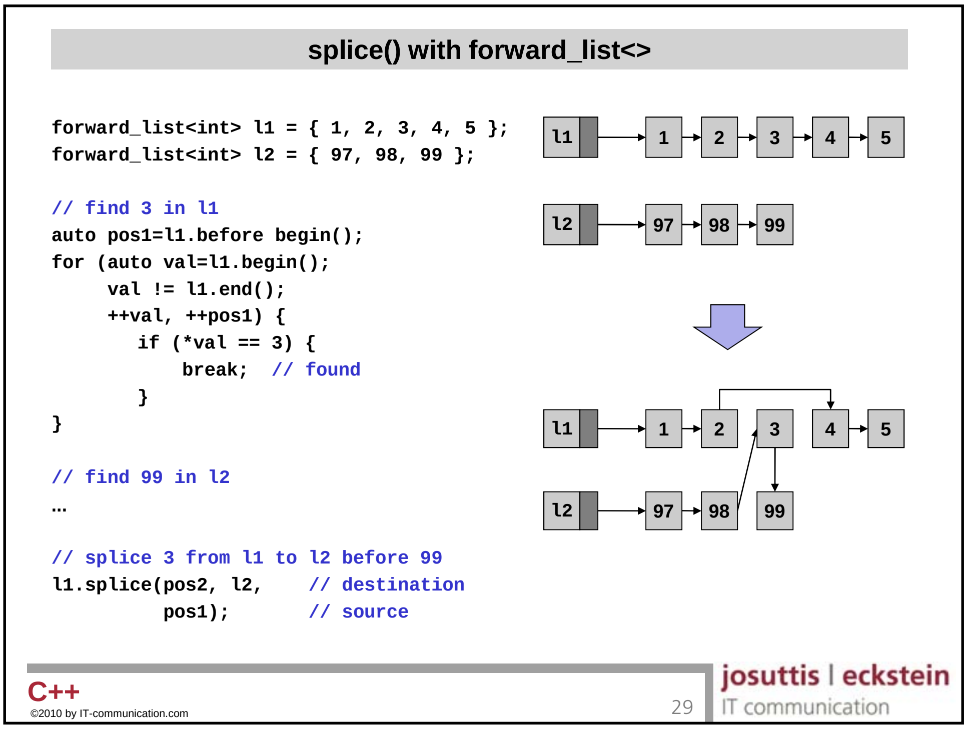
```
list<int> l1 = { 1, 2, 3, 4, 5 };
list<int> l2 = { 97, 98, 99 };

// find 3 in l1
auto pos1 = find(l1.begin(),l1.end(),
                 3);

// find 99 in l2
auto pos2 = find(l2.begin(),l2.end(),
                 99);

// splice 3 from l1 to l2 before 99
l1.splice(pos2, l2,     // destination
          pos1);        // source
```

l1
1
2
3
4
5
l2
97
98
99
l1
1
2
3
4
5
l2
97
98
99

C++

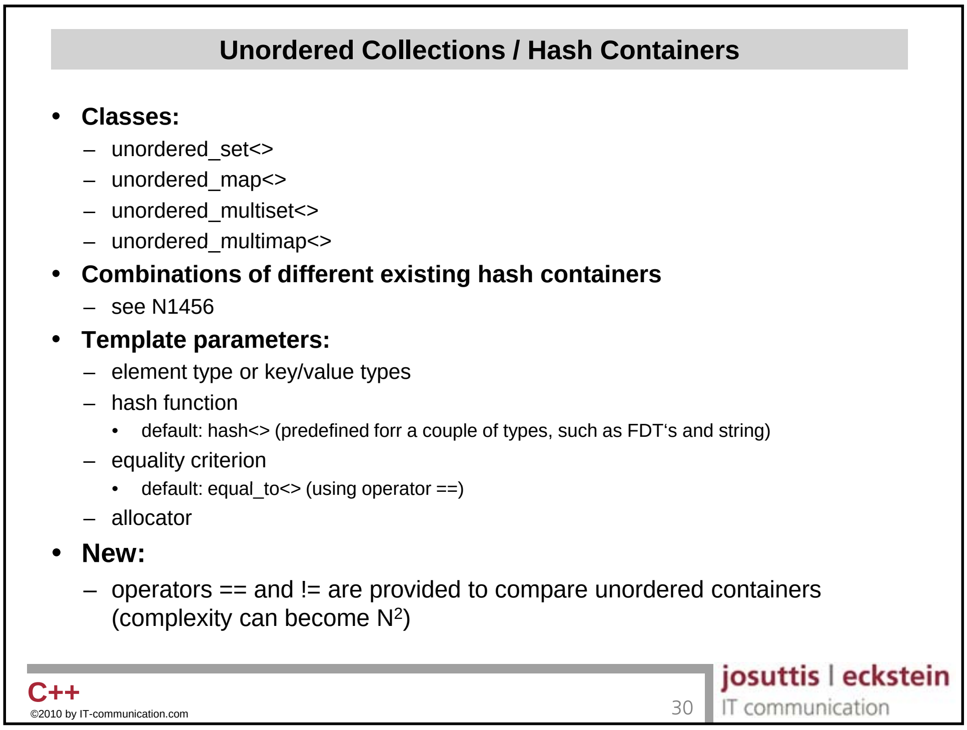
28

josuttis | eckstein
IT communication



splice() with forward_list<>

```
forward_list<int> l1 = { 1, 2, 3, 4, 5 };
forward_list<int> l2 = { 97, 98, 99 };

// find 3 in l1
auto pos1=l1.before begin();
for (auto val=l1.begin();
      val != l1.end();
      ++val, ++pos1) {
         if (*val == 3) {
             break;  // found
         }
}

// find 99 in l2
…

// splice 3 from l1 to l2 before 99
l1.splice(pos2, l2,    // destination
          pos1);       // source
```



Unordered Collections / Hash Containers

- **Classes:**
  - unordered_set<>
  - unordered_map<>
  - unordered_multiset<>
  - unordered_multimap<>
- **Combinations of different existing hash containers**
  - see N1456
- **Template parameters:**
  - element type or key/value types
  - hash function
    - default: hash<> (predefined forr a couple of types, such as FDT's and string)
  - equality criterion
    - default: equal_to<> (using operator ==)
  - allocator
- **New:**
  - operators == and != are provided to compare unordered containers (complexity can become N^2)

Controlling Hash Containers

- **Predefined:**
  - chaining is the approach for collisions
  - minimum load factor is 0
  - only insert() can shrink number of buckets
- **Implementation specific:**
  - singly or doubled linked list
    - C++0x requires at least forward iterators
  - rehashing strategy
  - growth factor
- **Controlled by programmer:**
  - minimum number of buckets
  - hash function
  - equality criterion
  - maximum load factor

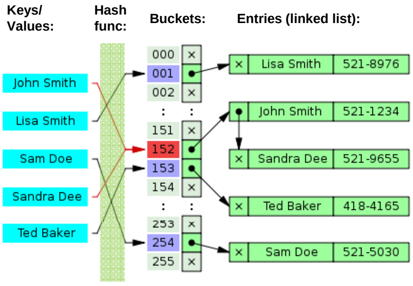
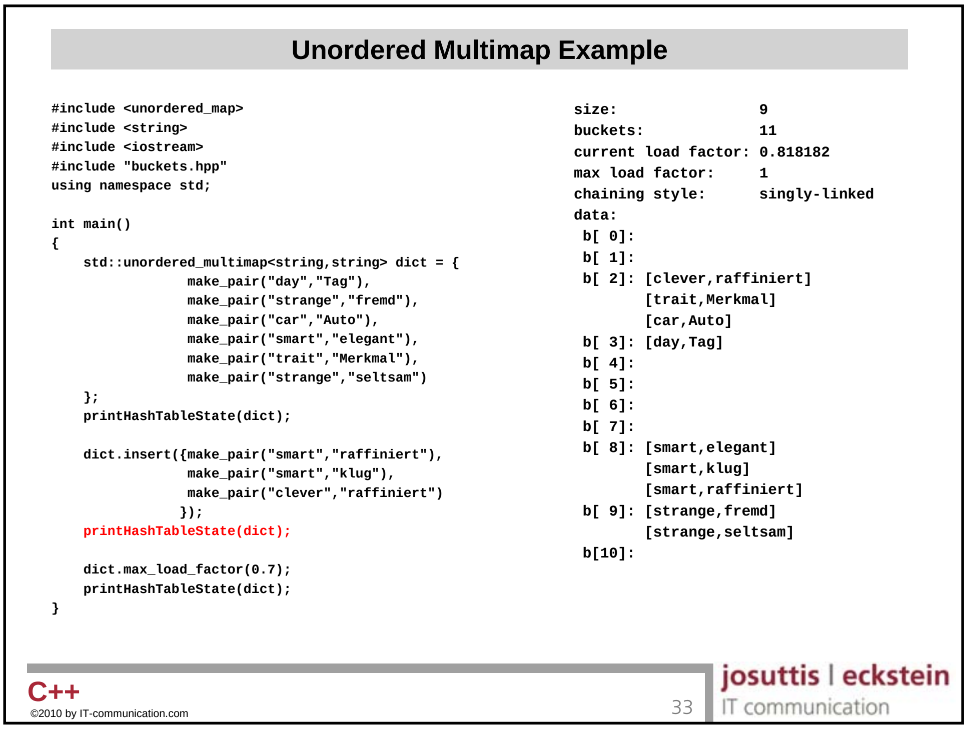


Figure based on Wipipedia

Unordered Multimap Example

```
#include <unordered_map>
#include <string>
#include <iostream>
#include "buckets.hpp"
using namespace std;

int main()
{
    std::unordered_multimap<string,string> dict = {
                 make_pair("day","Tag"),
                 make_pair("strange","fremd"),
                 make_pair("car","Auto"),
                 make_pair("smart","elegant"),
                 make_pair("trait","Merkmal"),
                 make_pair("strange","seltsam")
    };
    printHashTableState(dict);

    dict.insert({make_pair("smart","raffiniert"),
                 make_pair("smart","klug"),
                 make_pair("clever","raffiniert")
                });
    printHashTableState(dict);

    dict.max_load_factor(0.7);
    printHashTableState(dict);
}
```

```
size:                6
buckets:             11
current load factor: 0.545455
max load factor:     1
chaining style:      singly-linked
data:
 b[ 0]:
 b[ 1]:
 b[ 2]: [trait,Merkmal]
        [car,Auto]
 b[ 3]: [day,Tag]
 b[ 4]:
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]: [smart,elegant]
 b[ 9]: [strange,fremd]
        [strange,seltsam]
 b[10]:
```



Unordered Multimap Example

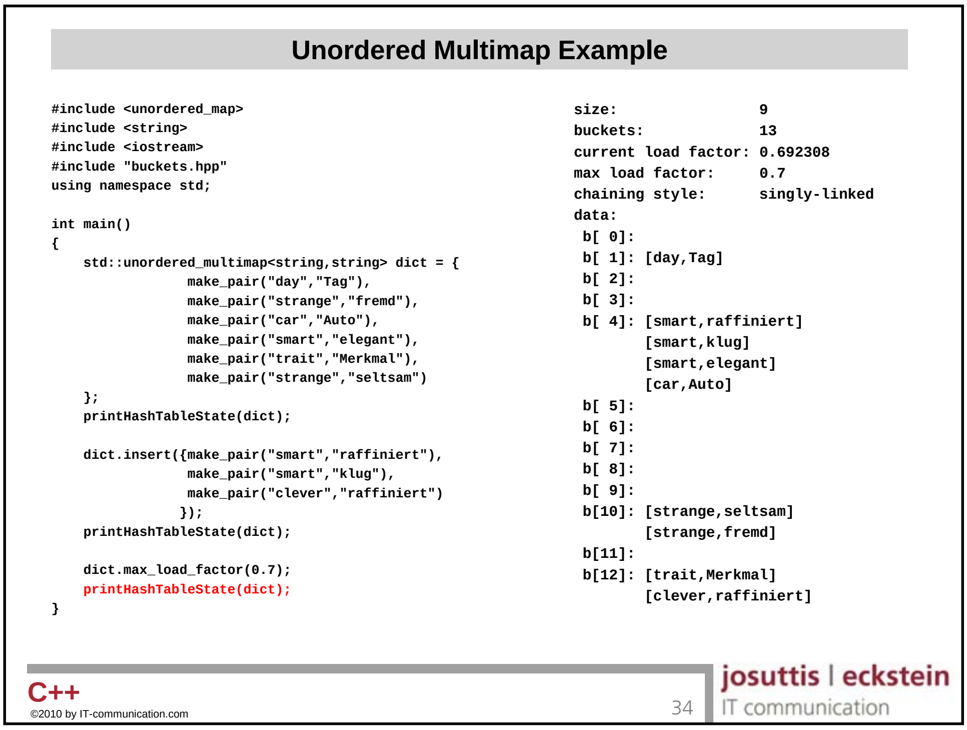
```
#include <unordered_map>
#include <string>
#include <iostream>
#include "buckets.hpp"
using namespace std;

int main()
{
    std::unordered_multimap<string,string> dict = {
                 make_pair("day","Tag"),
                 make_pair("strange","fremd"),
                 make_pair("car","Auto"),
                 make_pair("smart","elegant"),
                 make_pair("trait","Merkmal"),
                 make_pair("strange","seltsam")
    };
    printHashTableState(dict);

    dict.insert({make_pair("smart","raffiniert"),
                 make_pair("smart","klug"),
                 make_pair("clever","raffiniert")
                });
    printHashTableState(dict);

    dict.max_load_factor(0.7);
    printHashTableState(dict);
}
```

```
size:                9
buckets:             11
current load factor: 0.818182
max load factor:     1
chaining style:      singly-linked
data:
 b[ 0]:
 b[ 1]:
 b[ 2]: [clever,raffiniert]
        [trait,Merkmal]
        [car,Auto]
 b[ 3]: [day,Tag]
 b[ 4]:
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]: [smart,elegant]
        [smart,klug]
        [smart,raffiniert]
 b[ 9]: [strange,fremd]
        [strange,seltsam]
 b[10]:
```

C++

33

josuttis | eckstein
IT communication



Unordered Multimap Example

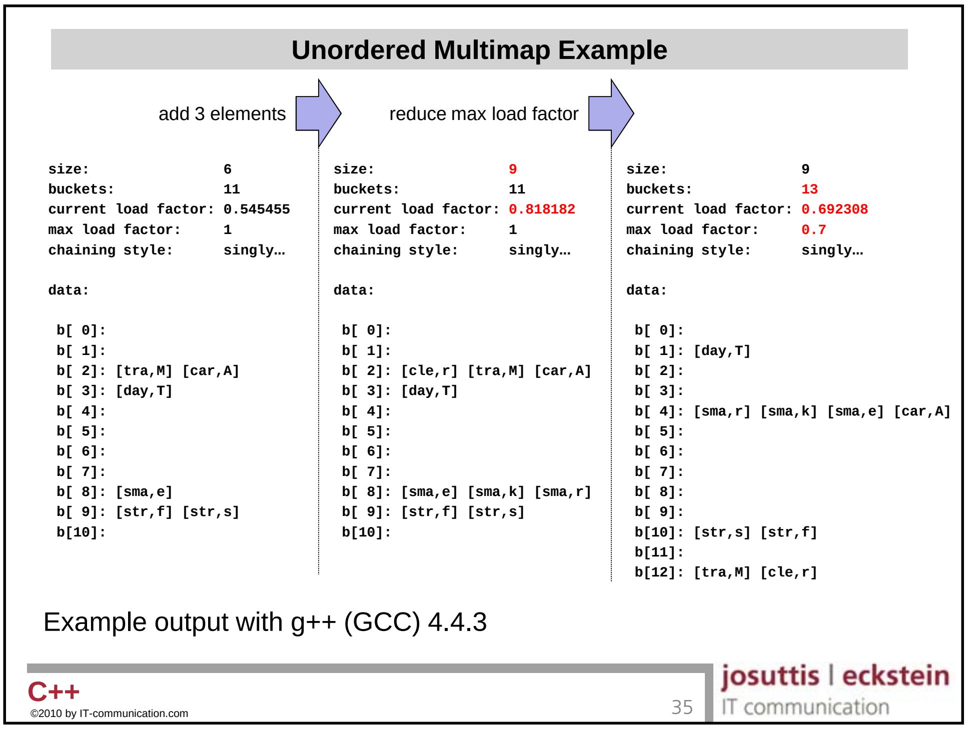
```
#include <unordered_map>
#include <string>
#include <iostream>
#include "buckets.hpp"
using namespace std;

int main()
{
    std::unordered_multimap<string,string> dict = {
                 make_pair("day","Tag"),
                 make_pair("strange","fremd"),
                 make_pair("car","Auto"),
                 make_pair("smart","elegant"),
                 make_pair("trait","Merkmal"),
                 make_pair("strange","seltsam")
    };
    printHashTableState(dict);

    dict.insert({make_pair("smart","raffiniert"),
                 make_pair("smart","klug"),
                 make_pair("clever","raffiniert")
                });
    printHashTableState(dict);

    dict.max_load_factor(0.7);
    printHashTableState(dict);
}
```

```
size:                9
buckets:             13
current load factor: 0.692308
max load factor:     0.7
chaining style:      singly-linked
data:
 b[ 0]:
 b[ 1]: [day,Tag]
 b[ 2]:
 b[ 3]:
 b[ 4]: [smart,raffiniert]
        [smart,klug]
        [smart,elegant]
        [car,Auto]
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]:
 b[ 9]:
 b[10]: [strange,seltsam]
        [strange,fremd]
 b[11]:
 b[12]: [trait,Merkmal]
        [clever,raffiniert]
```

C++

34

josuttis | eckstein
IT communication



Unordered Multimap Example

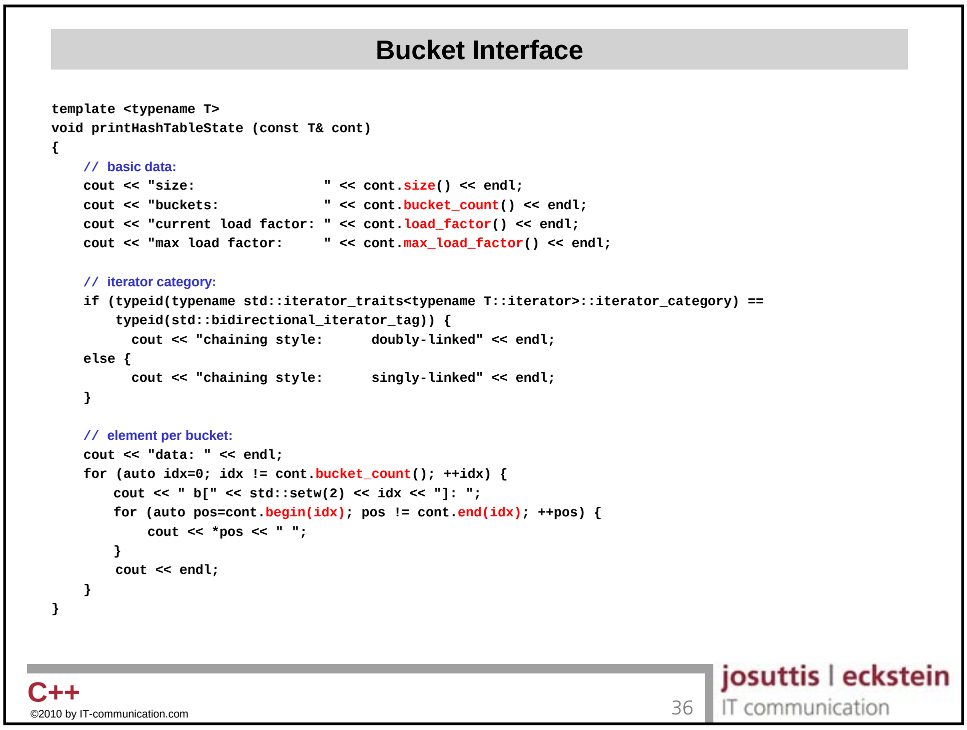
add 3 elements → reduce max load factor →

```
size:                 6
buckets:              11
current load factor: 0.545455
max load factor:      1
chaining style:       singly…

data:

 b[ 0]:
 b[ 1]:
 b[ 2]: [tra,M] [car,A]
 b[ 3]: [day,T]
 b[ 4]:
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]: [sma,e]
 b[ 9]: [str,f] [str,s]
 b[10]:
```

```
size:                 9
buckets:              11
current load factor: 0.818182
max load factor:      1
chaining style:       singly…

data:

 b[ 0]:
 b[ 1]:
 b[ 2]: [cle,r] [tra,M] [car,A]
 b[ 3]: [day,T]
 b[ 4]:
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]: [sma,e] [sma,k] [sma,r]
 b[ 9]: [str,f] [str,s]
 b[10]:
```

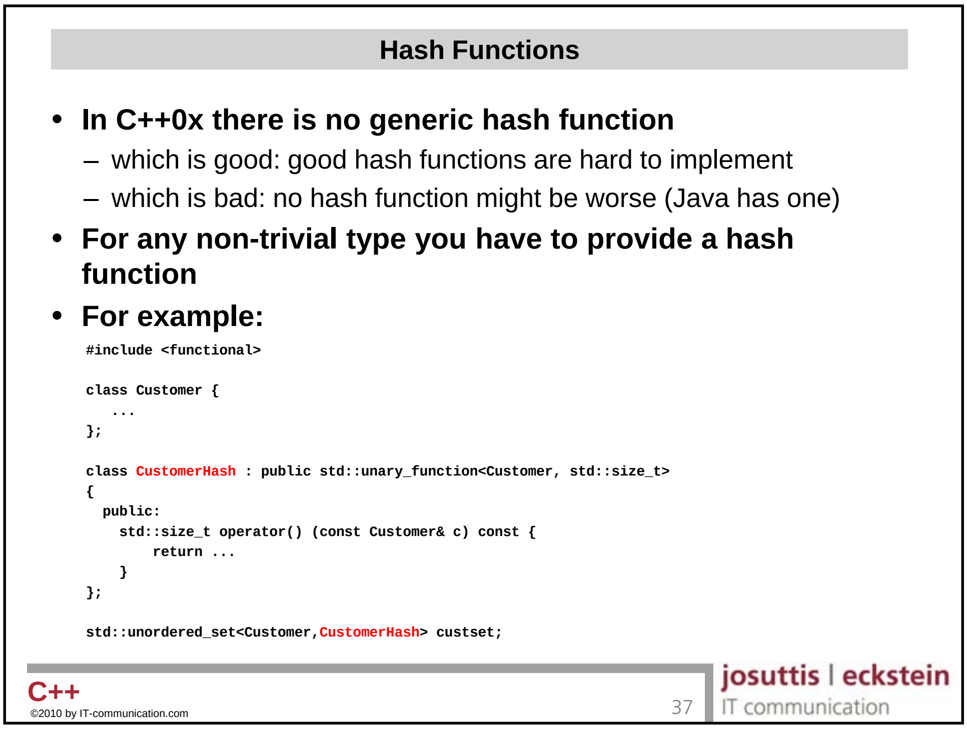
```
size:                 9
buckets:              13
current load factor: 0.692308
max load factor:      0.7
chaining style:       singly…

data:

 b[ 0]:
 b[ 1]: [day,T]
 b[ 2]:
 b[ 3]:
 b[ 4]: [sma,r] [sma,k] [sma,e] [car,A]
 b[ 5]:
 b[ 6]:
 b[ 7]:
 b[ 8]:
 b[ 9]:
 b[10]: [str,s] [str,f]
 b[11]:
 b[12]: [tra,M] [cle,r]
```

Example output with g++ (GCC) 4.4.3



Bucket Interface

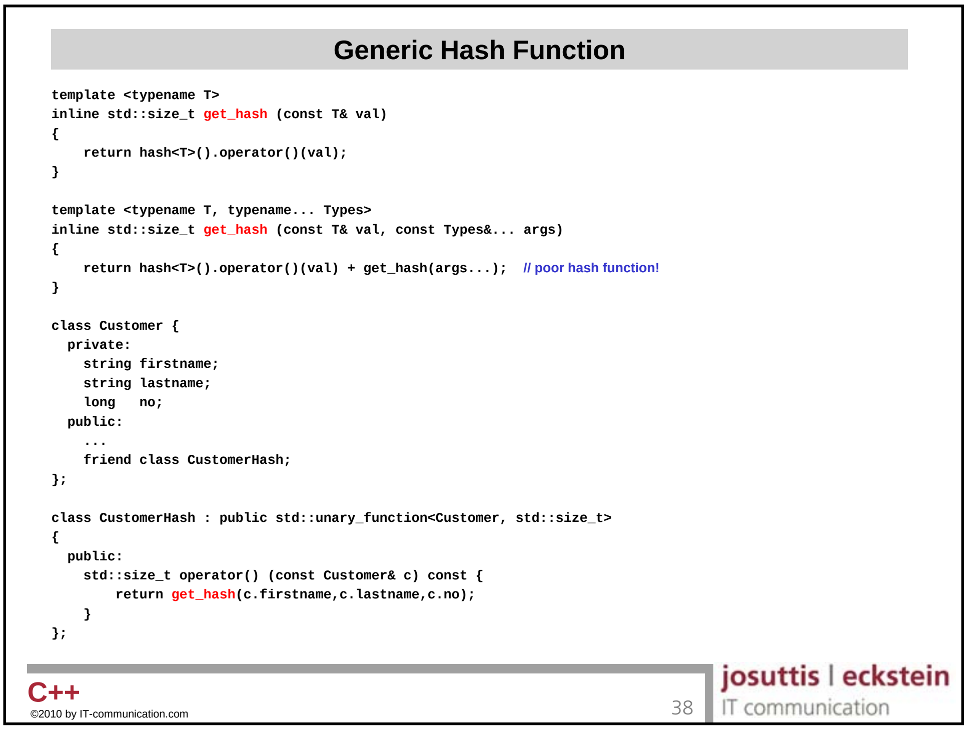
```
template <typename T>
void printHashTableState (const T& cont)
{
    // basic data:
    cout << "size:                " << cont.size() << endl;
    cout << "buckets:             " << cont.bucket_count() << endl;
    cout << "current load factor: " << cont.load_factor() << endl;
    cout << "max load factor:     " << cont.max_load_factor() << endl;

    // iterator category:
    if (typeid(typename std::iterator_traits<typename T::iterator>::iterator_category) ==
        typeid(std::bidirectional_iterator_tag)) {
          cout << "chaining style:      doubly-linked" << endl;
    else {
          cout << "chaining style:      singly-linked" << endl;
    }

    // element per bucket:
    cout << "data: " << endl;
    for (auto idx=0; idx != cont.bucket_count(); ++idx) {
        cout << " b[" << std::setw(2) << idx << "]: ";
        for (auto pos=cont.begin(idx); pos != cont.end(idx); ++pos) {
            cout << *pos << " ";
        }
        cout << endl;
    }
}
```



Hash Functions

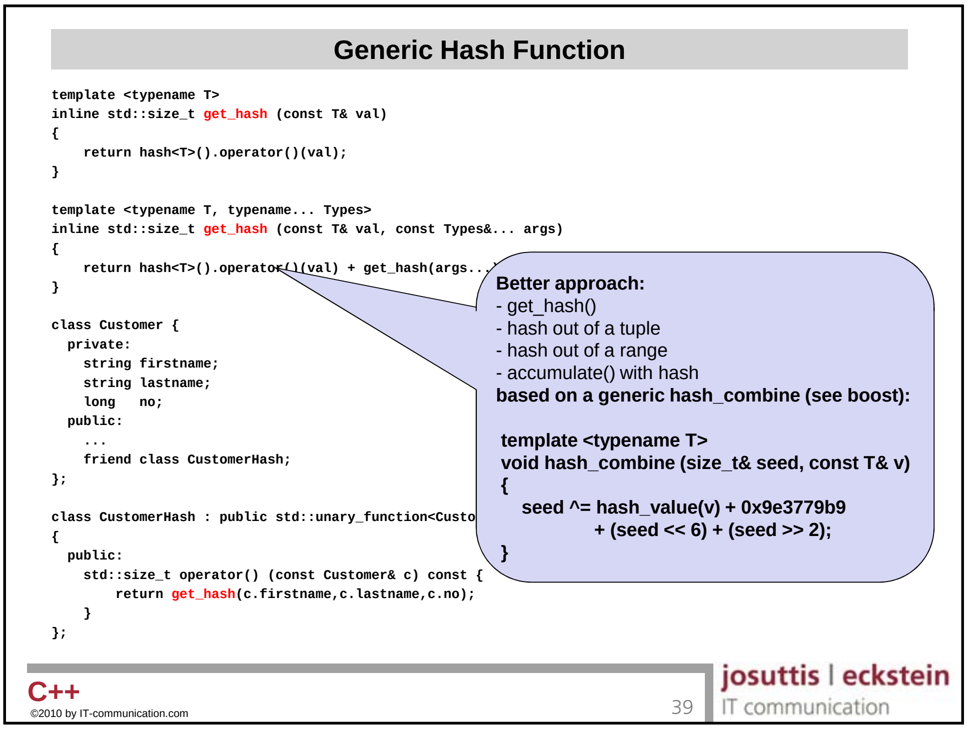
- **In C++0x there is no generic hash function**
  - which is good: good hash functions are hard to implement
  - which is bad: no hash function might be worse (Java has one)
- **For any non-trivial type you have to provide a hash function**
- **For example:**

```
#include <functional>

class Customer {
   ...
};

class CustomerHash : public std::unary_function<Customer, std::size_t>
{
  public:
    std::size_t operator() (const Customer& c) const {
        return ...
    }
};

std::unordered_set<Customer,CustomerHash> custset;
```



Generic Hash Function

```
template <typename T>
inline std::size_t get_hash (const T& val)
{
    return hash<T>().operator()(val);
}

template <typename T, typename... Types>
inline std::size_t get_hash (const T& val, const Types&... args)
{
    return hash<T>().operator()(val) + get_hash(args...);   // poor hash function!
}

class Customer {
  private:
    string firstname;
    string lastname;
    long   no;
  public:
    ...
    friend class CustomerHash;
};

class CustomerHash : public std::unary_function<Customer, std::size_t>
{
  public:
    std::size_t operator() (const Customer& c) const {
        return get_hash(c.firstname,c.lastname,c.no);
    }
};
```



Generic Hash Function

```
template <typename T>
inline std::size_t get_hash (const T& val)
{
    return hash<T>().operator()(val);
}

template <typename T, typename... Types>
inline std::size_t get_hash (const T& val, const Types&... args)
{
    return hash<T>().operator()(val) + get_hash(args...);
}

class Customer {
  private:
    string firstname;
    string lastname;
    long   no;
  public:
    ...
    friend class CustomerHash;
};

class CustomerHash : public std::unary_function<Custo
{
  public:
    std::size_t operator() (const Customer& c) const {
        return get_hash(c.firstname,c.lastname,c.no);
    }
};
```

Better approach:
- get_hash()
- hash out of a tuple
- hash out of a range
- accumulate() with hash

based on a generic hash_combine (see boost):

```
template <typename T>
void hash_combine (size_t& seed, const T& v)
{
   seed ^= hash_value(v) + 0x9e3779b9
            + (seed << 6) + (seed >> 2);
}
```

Q&A

Nicolai Josuttis

www.it-communication.com
josuttis@it-communication.com

Gaussstr. 29
D - 38106 Braunschweig
Germany

Tel.: +49 531 / 129 88 86
+49 700 / 5678 8888
+49 700 / JOSUTTIS

SOA in Practice
SOA 实践指南
SOA in der Praxis
C++ Templates The Complete Guide
The C++ Standard Library A Tutorial and Reference
Nicolai M. Josuttis