C++11: An Overview

science + computing ag

IT-Dienstleistungen und Software für anspruchsvolle Rechnernetze

Tübingen | München | Berlin | Düsseldorf

Bjarne Stroustrup about C++11

Bjarne Stroustrup:

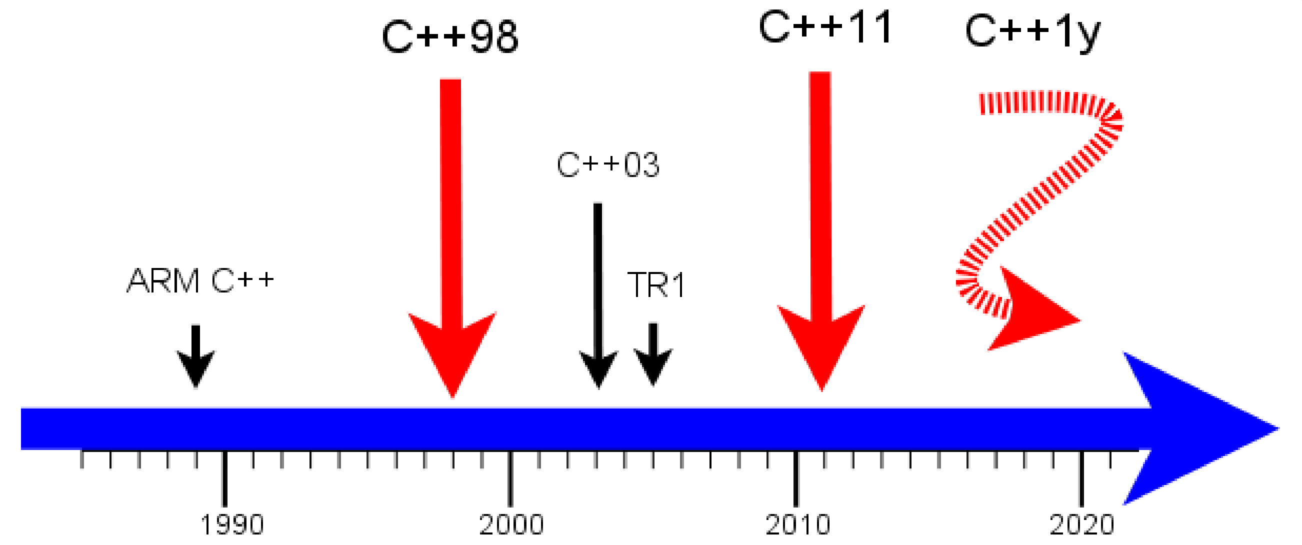
„Surprisingly, C++11 feels like a new language - the pieces just fit together better.“

Overview

(source:http://www.wojcik.net; 2012-02-28)

- the past: C++98
- the present: C++11
  - core language
  - multithreading
  - standard library
- the future: C++1y

Timeline



  - ARM C++: “The Annotated C++ Reference Manual“
- **C++98:** first ISO Standard
  - C++03: technical corrigendum of C++98
  - TR1: technical report 1
- **C++11:** current ISO Standard
- **C++1y:** future ISO Standard

Principles

- Principles of C++:
  - Trust the programmer.
  - You don't have to pay for something you don't need.
  - Don't break existing code.
  - Prefer compile time errors over run time errors.
- Aims for C++11:
  - Is the better programming language
    - for system programming.
    - for the building of libraries.
  - Is easier to teach and to learn.

Deduction of the type with auto

- The compiler determines the type:

```
auto myString= "my String";                  // C++11
auto myInt= 5;                               // C++11
auto myDouble= 3.14;                         // C++11
```

- Get a iterator on the first element of a vector:

```
vector<int> v;
vector<int>::iterator it1= v.begin();        // C++98
auto it2= v.begin();                         // C++11
```

- Definition of a function pointer:

```
int add(int a,int b){ return a+b; };
int (*myAdd1)(int,int)= add;                 // C++98
auto myAdd2= add;                            // C++11
myAdd1(2,3) == myAdd2(2,3);
```

Deduction of the type with decltype

- The compiler determines the type of an expression:

```
decltype("str") myString= "str";                        // C++11
decltype(5) myInt= 5;                                   // C++11
decltype(3.14) myFloat= 3.14;                           // C++11
decltype(myInt) myNewInt= 2011;                         // C++11

int add(int a,int b){ return a+b; };
decltype(add) myAdd= add;  // (int)(*)(int, int)       // C++11
myAdd(2,3) == add(2,3);
```

Deduce the return type of a function

- Example for the new alternative function syntax:

  func(arguments) → return value { functionbody }

- A generic add function with auto and decltype:

```
template <typename T1, typename T2>
auto add(T1 first, T2 second) -> decltype(first + second){
  return first + second;
}
add(1,1);
add(1,1.1);
add(1000LL,5);
```

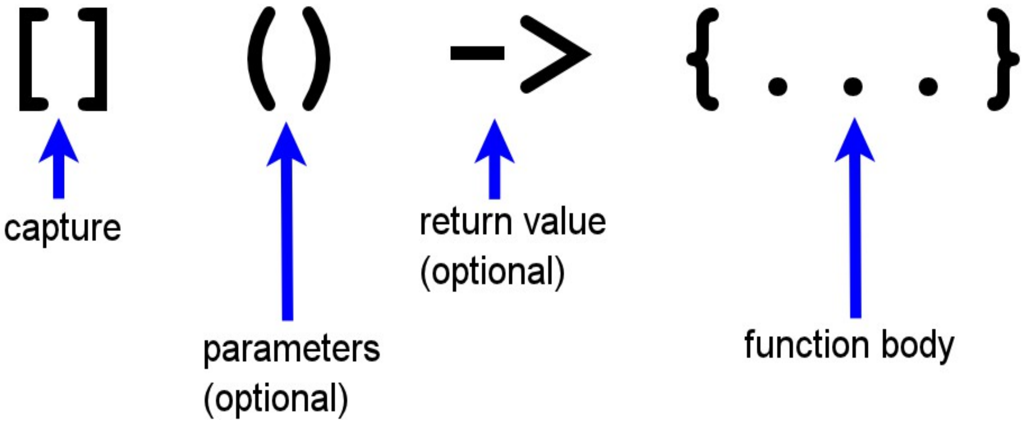
- the result is of type

```
int
double
long long int
```

Lambda functions

- Lambda functions
  - are functions without name.
  - define their functionality right in place.
  - can be copied like data.
- Lambda functions should be
  - concise.
  - self explaining.

Lambda functions: Syntax



[] () -> {...}

capture

parameters (optional)

return value (optional)

function body

- `[]`: captures the used variables per copy of per reference
- `()`: is required for parameters
- `->`: is required for sophisticated lambda functions
- `{}`: may include expressions and statements

Lambda functions

- Sort the elements of a vector:

  ```
  vector<int> vec={3,2,1,5,4};
  ```

  - in C++98 with a function object

    ```
    class MySort{
    public:
      bool operator()(int v, int w){ return v > w; }
    };
    // a lot of code
    sort(vec.begin(),vec.end(),MySort());
    ```

  - in C++11 with a lambda function:

    ```
    sort(vec.begin(),vec.end(),
    [](int v,int w){
       return v > w;
    });
    sort(vec.begin(),vec.end(),[](int v,int w){return v>w;});
    ```

Lambda functions

- Lambda functions can do much more:
  - starting a thread:

    ```
    thread t1([]{cout << this_thread::get_id() << endl;});
    thread t2([]{veryExpensiveFunction();});
    ```

  - Lambda functions are first-class functions:
    - argument of a function:

      ```
      auto myLambda= []{return "lambda function";};
      getLambda(myLambda);
      ```

    - return value of a function:

      ```
      function< string() > makeLambda{
        return []{return "2011";};
      };
      ```

Simple and unified initialization

- Simple data type:

```
int i{2011};
string st= {"Stroustrup"};
```

- Container:

```
vector<string> vec= {"Scott",st,"Sutter"};
unordered_map<string,int> um= {{"C++98",1998},{"C++11",i}};
```

- Array as a member of a class:

```
struct MyArray{
  MyArray(): myData{1,2,3,4,5} {}
  int myData[5];
}
```

- Const heap array:

```
const float* pData= new const float[5]{1,2,3,4,5};
```

The range-based for-loop

- Simple iteration over a container:

```
vector<int> vec={1,2,3,4,5};
for (auto v: vec) cout << v << ",";        // 1,2,3,4,5,

unordered_map<string,int> um= {{"C++98",1998},{"C++11",2011}};
for (auto u:um) cout << u->first << ":" << u->second << "   ";
   // "C++11":2011  "C++98":1998
```

- Modifying the container elements by auto&:

```
for (auto& v: vec) v *= 2;
for (auto v: vec) cout << v << " ,";         // 2,4,6,8,10,

string testStr{"Only for Testing."};
for (auto& c: testStr) c= toupper(c);
for (auto c: testStr) cout << c;  // "ONLY FOR TESTING."
```

Constructor: Delegation

```
class MyHour{
  int myHour_;
  public:
    MyHour(int h){                                       // #1
      if (0 <= h and h <= 23 ) myHour_= h;
      else myHour_= 0;
    }
    MyHour(): MyHour(0){};                               // #2
    MyHour(double h): MyHour(static_cast<int>(ceil(h)){};// #3
};
```

- The constructors #2 and #3 invoke the constructor #1.

Constructor: Inheritance (using)

```
struct Base{
  Base(int){}
  Base(string){}
};
struct Derived: public Base{
  using Base::Base;
  Derived(double){}
};
int main(){
  Derived(2011);         // Base::Base(2011)
  Derived("C++11");      // Base::Base(C++11)
  Derived(0.33);         // Derived::Derived(0.33)
}
```

Requesting methods (`default`)

- Requesting special methods and operators from the compiler:
  - default - and copy - constructor;
    assignment operator, operator new;
    destructor

```
class MyType{
  public:
    MyType(int val) {}                        // #1
    MyType()= default;                        // #2
    virtual ~MyType();
    MyType& operator= (MyType&)
};
MyType::~MyType()= default;
MyType& MyType::operator(MyType&)= default;
```

- #1 suppresses the automatic generation of #2.

Suppress function invocations (`delete`)

- Not copyable classes:

```
class NonCopyClass{
  public:
    NonCopyClass()= default;
    NonCopyClass& operator =(const NonCopyClass&)= delete;
    NonCopyClass (const NonCopyClass&)= delete;
};
```

- A function only accepting `double`:

```
void onlyDouble(double){}
template <typename T> void onlyDouble(T)= delete;
int main(){
  onlyDouble(3);
};
```

➜ `Error: use of deleted function »void onlyDouble(T) [with T = int]«`

Explicit override (`override`)

- Control by the compiler:

```
class Base {
  virtual void func1();
  virtual void func2(float);
  virtual void func3() const;
  virtual long func4(int);
};
class Derived: public Base {
  virtual void fun1() override;                  // ERROR
  virtual void func2(double) override;           // ERROR
  virtual void func3() override;                 // ERROR
  virtual int func4(int) override;               // ERROR
  virtual long func4(int) override;              // OK
};
```

Suppress override (`final`)

- For methods:

```
class Base {
  virtual void h(int) final;
};
class Derived: public Base {
  virtual void h(int);                       // ERROR
  virtual void h(double);                    // OK
};
```

- For classes:

```
struct Base final{};
struct Derived: Base{};                      // ERROR
```

Rvalue references

- rvalue references are special references that can be bind to a rvalue.
- rvalues are
  - temporary.
  - objects without name.
  - objects, of which can not be determined an address.
- rvalue references are defined with 2 and symbols (&&):

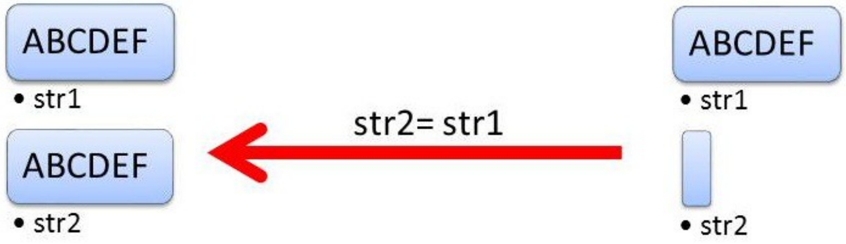
```
MyData myData;
MyData& myDataLvalue= myData;
MyData&& myDataRvalue( MyData());
```

- The compiler can bind lvalue references to an lvalue, rvalue references to an rvalue.
  - Special action can be given for rvalues.
- Use case: move semantic and perfect forwarding.

Move semantic (move)

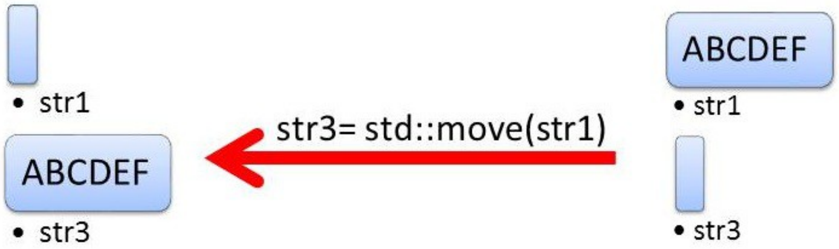
Copy

```
string str1("ABCDEF");
string str2;
str2= str1;
```



Move

```
string str1{"ABCDEF"};
string str3;
str3= std::move(str1);
```



Move semantic

- Advantages:
  - cheap moving of a resource instead of expensive copying:

    ```
    vector<int> myBigVector;
    ....
    vector<int> myBigVector2(move(myBigVector));
    ```
  - Not copyable but moveable objects can be given to or by a function *by value.*
    - Examples: unique_ptr, files, mutexe, promise and future

      ```
      mutex m;
      unique_lock<mutex> uniqueLock(m);
      unique_lock<mutex> uniqueLock2(move(m));
      ```

Perfect forwarding (`forward`)

- Enables to write function templates, which can forward their argument to a further function preserving the lvalue/rvalue items of the arguments.
  - Stroustrup: „ . . . *a heretofore unsolved problem in C++.“*
- Use case: factory function or constructor
- Example: factory function with one argument

```
template <typename T, typename T1>
T createObject(T1&& t1){
  return T(forward<T1>(t1));
}
int myFive2= createObject<int>(5);               // Rvalue
int five=5;
int myFive= createObject<int>(five);             // Lvalue
```

Variadic templates (...)

- Templates, which can get an arbitrary number of arguments
- The ellipse ... denotes the template parameter pack, that can be packed or unpacked.
- Application: std::tuple, std::thread
- Example: a completely generic factory function

```
template <typename T, typename ... Args>
T createObject(Args&& ... args){
  return T(forward<Args>(args)...);
}
string st= createObject<string>("Rainer");
struct MyStruct{
  MyStruct(int i,double d,string s){}
};
MyStruct myStr= createObject<MyStruct>(2011,3.14,"Rainer");
```

More control at compile time (static_assert)

- Has no influence on the run time of the program.
- static_assert can be combined very well with the new type traits library.
- Assert:
  - a 64-bit architecture:

    ```
    static_assert(sizeof(long) >= 8,"no 64-bit code");
    ```

  - an arithmetic type:

    ```
    template< typename T >
    struct Add{
      static_assert(is_arithmetic<T>::value,"T is not arith");
    } ;
    ```

const expressions (constexpr)

- Is an optimisation opportunity for the compiler:
  - Can be evaluated at compile time.
  - The compiler gets a deep insight in the evaluated code.
- Three types:
  - variables:

    ```
    constexpr double myDouble= 5.2;
    ```

  - functions:

    ```
    constexpr fact (int n){return n > 0 ? n * fact(n-1) : 1;}
    ```

  - user defined types:

    ```
    struct MyDouble{
      double myVal;
      constexpr MyDouble(double v): myVal(v){}
    };
    ```

Raw string literales (r“(raw string)“)

- Suppress the interpretation of the string
- Defined with r“(raw string)“ or R“(Raw String)“
- Are practical helper for:
  - paths:

```
string pathOld= "C:\\temp\\newFile.txt";
string pathRaw= r"(C:\temp\newFile.txt)";
```

  - regular expressions:

```
string regOld= "c\\+\\+";
string regRaw= r"(c\+\+)";
```

What I further want to say

- Design of classes:
  - in-class member initialization:

```
class MyClass{
  const static int oldX= 5;
  int newX= 5;
  vector<int> myVec{1,2,3,4,5};
};
```

- Extended data concepts:
  - unicode support: UTF-16 und UTF-32
  - user defined literales: `63_s; 123.45_km; "Hallo"_i18n`
  - the null pointer literal nullptr

Multithreading

C++11's answer to the requirements of the multi-core architectures.

(source: http://www.livingroutes.org, 2012-02-28)

- a standardized threading interface
- a defined memory model

Thread versus task

- thread

```
int res;
thread t([&]{res= 3+4;});
t.join();
cout << res << endl;
```

- task

```
auto fut=async([]{return 3+4;});
cout << fut.get() << endl;
```

aspect	thread	task
communication	shared variable	channel between father and child
thread creation	obligatory	optional
synchronisation	the father is waiting for his child	the `get`-invocation is blocking
exception in the child	child and father terminates	return value of the `get`-invocation

Threads (thread)

- A thread will be parametrized with its working package and starts immediately.
- The father thread:

  ```
  thread t([]{ cout << "I'm running." << endl;});
  ```

  - has to wait for its child:

    ```
    t.join();
    ```

  - needs to separate itself from the child (daemon thread):

    ```
    t.detach();
    ```

- data should be copied per default into child thread:

  ```
  string s{"undefined behavior"};
  thread t([&]{ cout << s << endl;});
  t.detach();
  ```

Thread-local data (thread_local)

- Are unique to a thread
- Behave like static variables

```
void addThreadLocal(string const& s){
  thread_local threadLocalStr("Hello from ");
  threadLocalStr += s;
  cout << threadLocalStr << endl;
}

thread t1(addThreadLocal,"t1");
thread t2(addThreadLocal,"t2");
```

- Result: Hello from t1
  Hello from t2

Protection of data (mutex)

- Shared variables have to be protected against race conditions.
- **race condition**: Two or more threads use a shared variable at the same time, and at least one of them is a write access.
- A mutex (**mut**ual **ex**clusion)
  - ensures the mutual exclusion
  - exists in C++11:
    - in recursive and not recursive way.
    - with and without relative and absolute time.

Deadlocks with mutexes

science + computing
A Bull Group Company

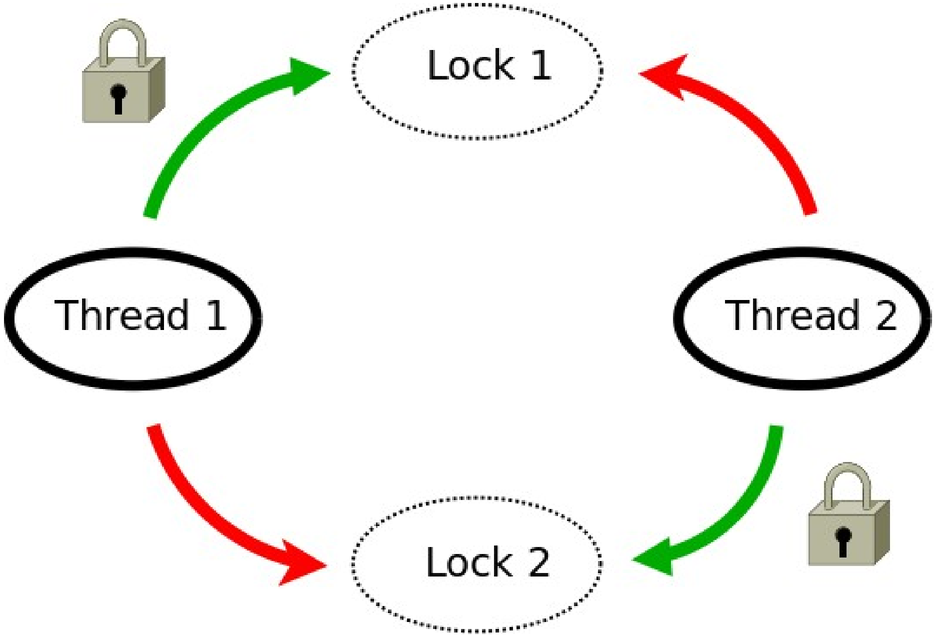
Exception:

- mutex in use:

```
mutex m;
m.lock();
sharedVar= getVar();
m.unlock();
```

- Problem: An exception in getVar() can result in a deadlock.

Acquire the lock in different order:



➔ Use lock_guard and unique_lock.

RAII with lock_guard and unique_lock

- lock_guard and unique_lock manage the lifetime of their mutex according to the RAII idiom.
- lock_guard:

```
mutex mapMutex;
{
  lock_guard<mutex> mapLock(mapMutex);
  addToMap("white",0);
}
```

- unique_lock for the more advanced use
  - Set or release explicit the lock.
  - Move or swap the lock.
  - Tentative or delayed locking.

Initialisation of shared variables

- Variables, that are read-only, have only to be initialised in a secure manner.
  - The expensive locking of the variable is not necessary.
- C++11 offers 3 opportunities:

  1) constant expressions:

  ```
  constexpr MyDouble myDouble;
  ```

  2) call_once and once_flag:

  ```
  void onlyOnceFunction(){ .... };
  once_flag= onceFlag;
  call_once(onceFlag,onlyOnceFunction)
  ```

  3) static local variables:

  ```
  void func(){ ... static int a=2011; ...}
  ```

Condition variables (notify_one, wait)

- a sender – a receiver:

```
mutex proVarMutex;
condition_variable condVar;
bool dataReady;
```

thread 1: sender

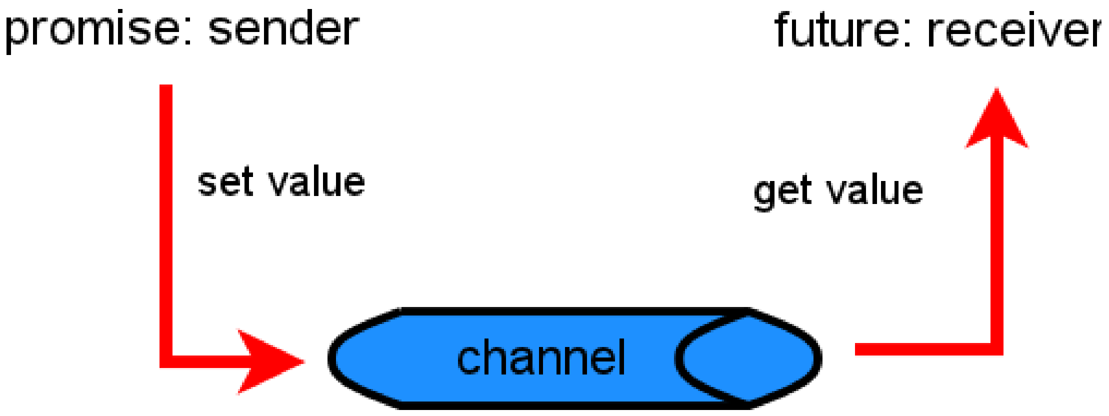
```
          lock_guard<mutex> sender_lock(protVarMutex);
          protectedVar= 2000;
          dataReady= true;
          condVar.notify_one();
```

thread 2 : receiver

```
          unique_lock<mutex> receiver_lock(protVarMutex);
          condVar.wait(receiver_lock,[]{return dataReady;});
          protectedVar += 11;
```

- a sender - many receivers (notify_all and wait)

Promise and future as data channel



- The promise
  - sends the data
  - can serve many futures
  - can send values, exceptions and notifications
- The future
  - is the data receiver
  - the `get`-invocation is blocking

Promise and future in use

```
a=2000;
b=11;
```

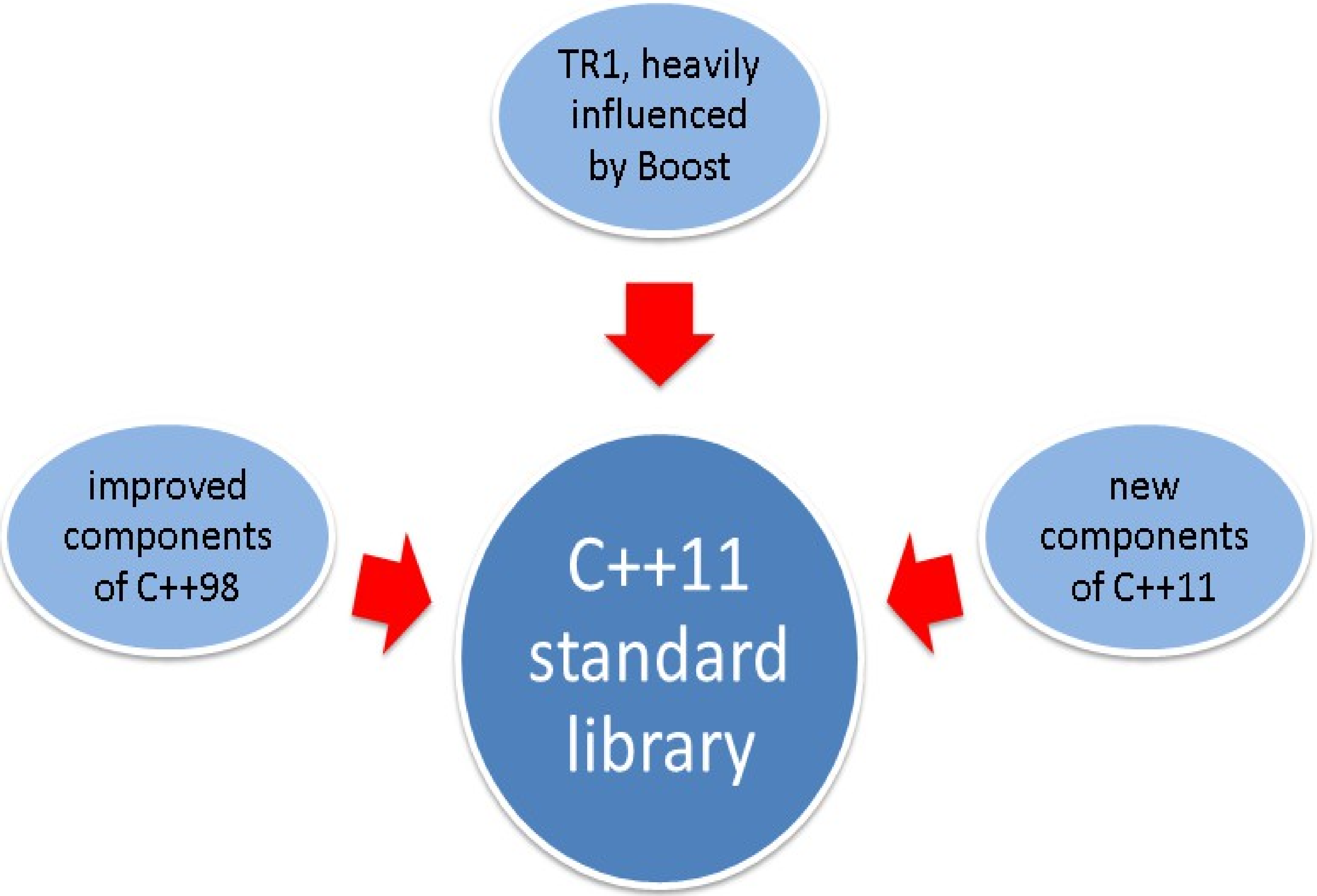
- Implicitly by async

```
future<int> sum= async([=]{ return a+b; });
sum.get();
```

- Explicitly by future and promise

```
void sum(promise<int>&& intProm,int x,int y){
  intProm.set_value(x+y);
}
promise<int> sumPromise;
future<int> futRes= sumPromise.get_future();
thread sumThread(&sum,move(sumPromise),a,b);
futRes.get();
```

Influences on the new standard library



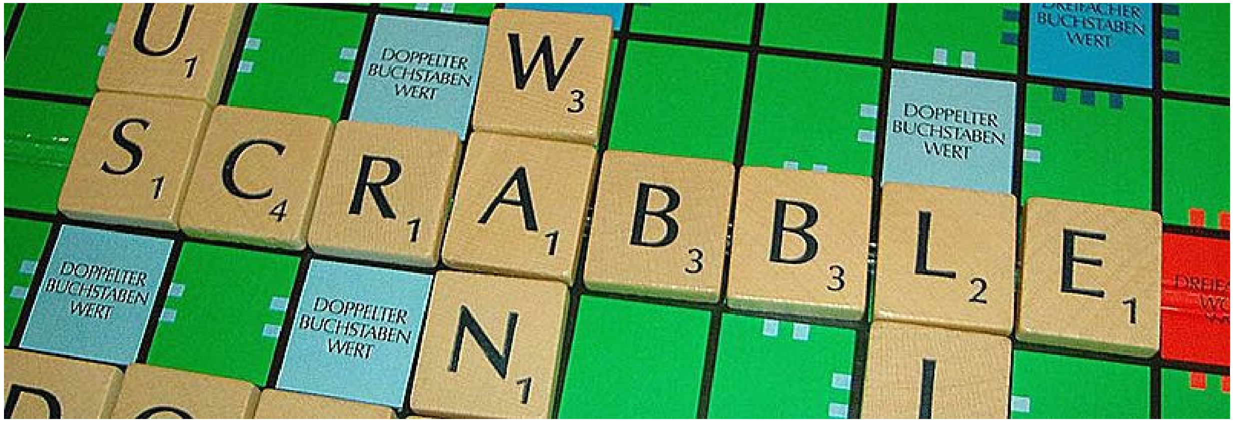
TR1, heavily influenced by Boost

improved components of C++98

C++11 standard library

new components of C++11

Regular expressions



- is a formal language for describing text patterns
- is the tool for text manipulation:
  - is a text equal to a text pattern?
  - search for a text pattern in a text
  - substitute a text pattern in a text
  - iterate over all text patterns in a text

Regular Expressions: Example

- Search for the first occurrence of a number in a text:

```
string text("abc1234def");
string regExNumber(r"(\d+)");
smatch holdResult;
if ( regex_search(text,holdResult,regExNumber) )
    cout << holdResult[0]   << endl;
    cout << holdResult.prefix() << endl;
    cout << holdResult.suffix() << endl;
```

➔ Result: 1234
abc
def

Regular Expressions

- Iterate over all numbers in a text:

```
string text="Der bisherige Standard C++98 ist nach 13
Jahren am 12. August 2011 durch den neuen Standard C++11
abgelöst worden."

regex regNumb(r"(\d+)");

sregex_token_iterator it(text.begin(),text.end(),regNumb);

sregex_token_iterator end;

while (it != end) cout << *it++ << " ";
```

➔ Result: 98 13 12 2011 11

Type traits

- Enable at compile time:
  - type queries (`is_integral<T>,is_same<T,U>`)

```
template <typename T>
T gcd(T a, T b){
  static_assert(is_integral<T>::value,"not integral");
  if( b==0 ) return a;
  else return gcd(b, a % b);
}
```

  - type transformations (`add_const<T>`)

```
typedef add_const<int>::type myConstInt;
cout << is_same<const int,myConstInt>::value << endl;
```

→ Result: true

→ Code, which is self-tuning

Random numbers

- combines a random number generator with a random number distribution:
    - random number generator:
        - creates a stream of random numbers between a minimum and maximum value
        - examples: Mersenne Twister, random_device (*/dev/urandom*)
    - random number distribution:
        - maps the random numbers on the distribution
        - examples: uniform, normal, poisson and gamma distribution
- Throw of a dice:

```
random_device seed;
mt19337 numberGenerator(seed());
uniform_int_distribution<int> six(1,6);
cout << six(numberGenerator)  << endl;            // 3
```

Time utilities

- Elementary component of the new multithreading functionality:
- Examples:
  - Put the actual thread for 100 milliseconds to sleep:

    ```
    this_thread::sleep_for( chrono::millisecond(100) );
    ```

  - performance measurement in seconds:

    ```
    auto begin= chrono::system_clock::now();
    //  a lot to do
    auto end= chrono::system_clock::now() - begin;
    auto timeInSeconds= chrono::duration<double>(end).count();
    ```

Reference wrapper (reference_wrapper)

- reference_wrapper<T> is a copy constructible and assignable wrapper around an object of type T&.
  - behaves as a reference, but can be copied
- New use cases:
  1. classes containing references can be copied:

```
struct Copyable{
  Copyable(string& s): name(s){}
  // string& badName; will not compile
  reference_wrapper<string> name;
};
```

  2. references can be used inside containers of the STL:

```
int a=1, b=2, c=4;
vector<reference_wrapper<int>> vec={ref(a),ref(b),ref(c)};
c = 3;
```

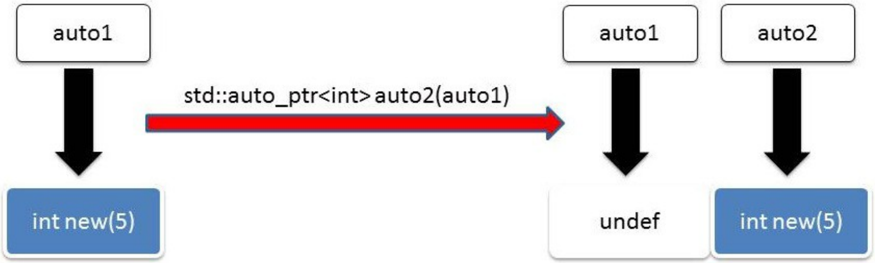
  - Result: vec[2] == 3

Smart pointer: lifecycle management

name	C++ standard	description
auto_ptr	C++98	- owns exclusive the resource - moves the resource silently while copying
unique_ptr	C++11	- owns exclusive the resource - an not be copied - can manage not copyable objects (threads,locks, files, …)
shared_ptr	C++11	- has a reference counter for his resource - manage automatically the reference counter - deletes the resource, if the reference count is 0

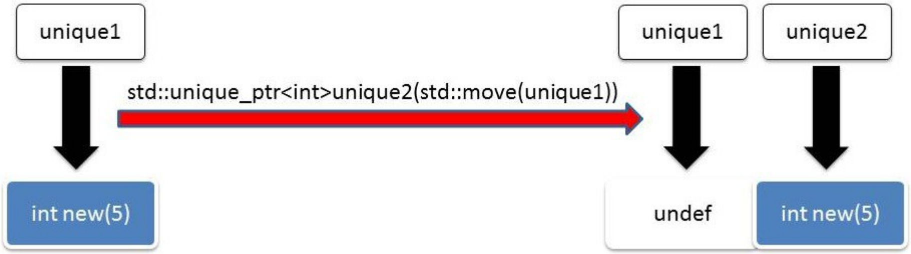
Smart pointer: *copying*

auto_ptr



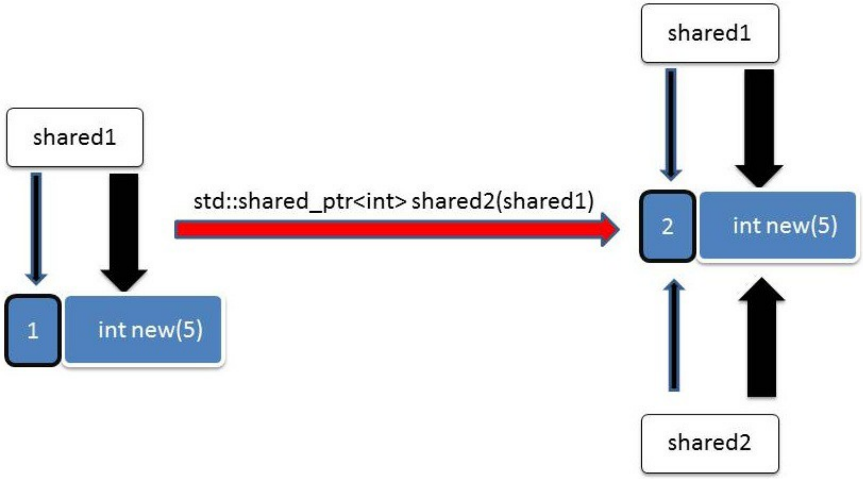
std::auto_ptr<int> auto2(auto1)

unique_ptr



std::unique_ptr<int>unique2(std::move(unique1))

shared_ptr



std::shared_ptr<int> shared2(shared1)

Smart pointer in use

```
shared_ptr<int> sharedPtr(new int(5));          // refcount == 1
{
    shared_ptr<int> localSharedPtr(sharedPtr);  // refcount == 2
}                                               // refcount == 1
shared_ptr<int> globalSharedPtr= sharedPtr;     // refcount == 2
globalSharedPtr.reset();                        // refcount == 1




unique_lock<int> uniqueInt(new int(2011));
unique_lock<mutex> uniqueInt2(uniqueInt);       // error !!!
unique_lock<mutex> uniqueInt2(move(uniqueInt));
vector<std::unique_ptr<int>> myIntVec;
myIntVec.push_back(move(uniqueInt2));
```

New container (tuple and array)

- tuple:
  - heterogeneous container of fixed length
  - extension of the container pair from C++98:

    ```
    tuple<string,int,float> tup=("first",1998,3.14);
    auto tup2= make_tuple("second",2011,'c');
    get<1>(tup)= get<1>(tup2);
    ```
- array:
  - homogeneous container of fixed length
  - combines the performance of a C array with the interface of a C++ vector:

    ```
    array<int,8> arr{{1,2,3,4,5,6,7,8}};
    int sum= 0;
    for_each(arr.begin(),arr.end(),[&sum](int v){sum += v;});
    ```

New container (hash tables)

- consists of (key,value) pairs
- also known as dictionary or associative array
- unordered variant of the C++ container map, set, multimap and multiset
- 4 variations:

name	has value	more equal keys
unordered_map	yes	no
unordered_set	no	no
unordered_multimap	yes	yes
unordered_multiset	no	yes

- comparison of the C++11 with the C++98 containers:
  - very similar interface
  - keys unordered
  - constant access time

New container (hash tables)

```
map<string,int> m {{"Dijkstra",1972},{"Scott",1976}};
m["Ritchie"] = 1983;
for(auto p : m) cout << '{' << p.first << ',' << p.second << '}';

cout << endl;

unordered_map<string,int> um { {"Dijkstra",1972},{"Scott",1976}};
um["Ritchie"] = 1983;
for(auto p : um) cout << '{' << p.first << ',' << p.second << '}';
```

- Result: {Dijkstra,1972}{Ritchie,1983}{Scott,1976}
  {Ritchie,1983}{Dijkstra,1972}{Scott,1976}

bind and function

- Feature for the functional programming:
  - bind allows to easily build functions object.
  - function binds the function objects from bind.

    ```
    int add(int a, int b){ return a+b;};
    function< int(int)> myAdd= bind(add,2000,_1);
    add(2000,11) == myAdd(11);
    ```
- Both libraries are due to the core language extension nearly superfluous:
  - For bind you can use lambda functions.
  - For function you can use auto:

    ```
    auto myAddLambda= [](int v){ return add(2000,v); };
    add(2000,11) == myAddLambda(11);
    ```

C++1y

Predictions are difficult, especially when they concern the future.

(source: www.nato.int; 2012-02-28)

- time frame for C++1y: 2017
- extension of the library:
  - Technical Report with file system
- content
  - constrained templates (2022)
  - multithreading
    - STM (2022)
    - asynchronous IO
  - modules
  - libraries

C++11: An overview

science + computing
A Bull Group Company

Vielen Dank für Ihre Aufmerksamkeit.

Rainer Grimm
science + computing ag
www.science-computing.de

phone +49 7071 9457-253
r.grimm@science-computing.de