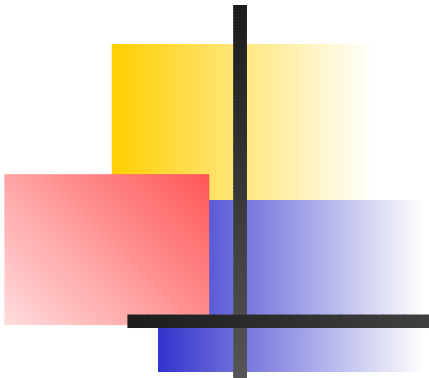
From Java to C++

There and never back again

Roland Lezuo

tbfg@se-linux.inso.tuwien.ac.at

183 Institut für Rechnergestützte Automation - INSO



Overview

- Programming languages
- First C++ program
- Concepts supported by C++
- Advanced C++

Perfect language

- easy to learn and use
- can do everything with it
- code once run everywhere
- uses platform specific features
- offers high level abstraction
- is efficient
- is stylish

Perfect language cont'd

Of course: There is no such thing as a perfect language! (Right not even C++ is perfect)
Wishlist:

- orthogonallity of features
- opt-in of each feature
- static typesafety?
- no syntax?
- bindings for other languages?

C++ usage

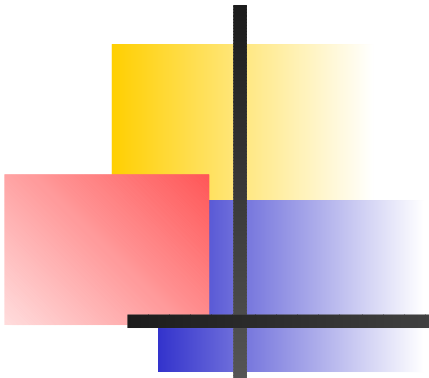
- kernels (MacOS X)
- middleware (Ice, google)
- large applications (openoffice.org)
- games
- embedded (but: stack usage)

Goodbye world in Java

```
class MemLeak
{
  // although there is no such thing as a memory leak in java, this leaks
  // like a barrel without bottom.
  private static java.util.Vector buffer = new java.util.Vector();

  public static void main(String args[])
  {
    for (;;) {
      java.lang.Integer i = new java.lang.Integer(1);
      buffer.add(i);
    }
  }
}
```

Hello world in C++

```
#include <iostream>
#include <cstdlib>

int main(int argc, char* argv[])
{
        std::cout << "Hello World!" << std::endl;
        exit(EXIT_SUCCESS);
}
```



The differences

C++ is an extension of C and mostly backwards compatible.

- classes are an extension and not mandatory
- main outside of a class definition
- everything outside of a class definition is static
- return value is status for operation system and a must
- int main(void) is possible

Declaration and definition

In C++ declaration and definition are separated. Declarations are in so called header files only contain the informations needed to use the class, the definition is in a separate file including the header and implementing the declarations made there.

$\Rightarrow$ Writing a .h file is software architecture
$\Rightarrow$ Writing a .cpp file is programming.

Headers in C++

```
#ifndef myclass_hh_
#define myclass_hh_
        class myClass   {
                private:
                        int _var;
                public:
                        myClass(int);
                        ~myClass() { }
                        int getVar() { return _var; }
                        void setVar(int var=0);
        };
#endif
```

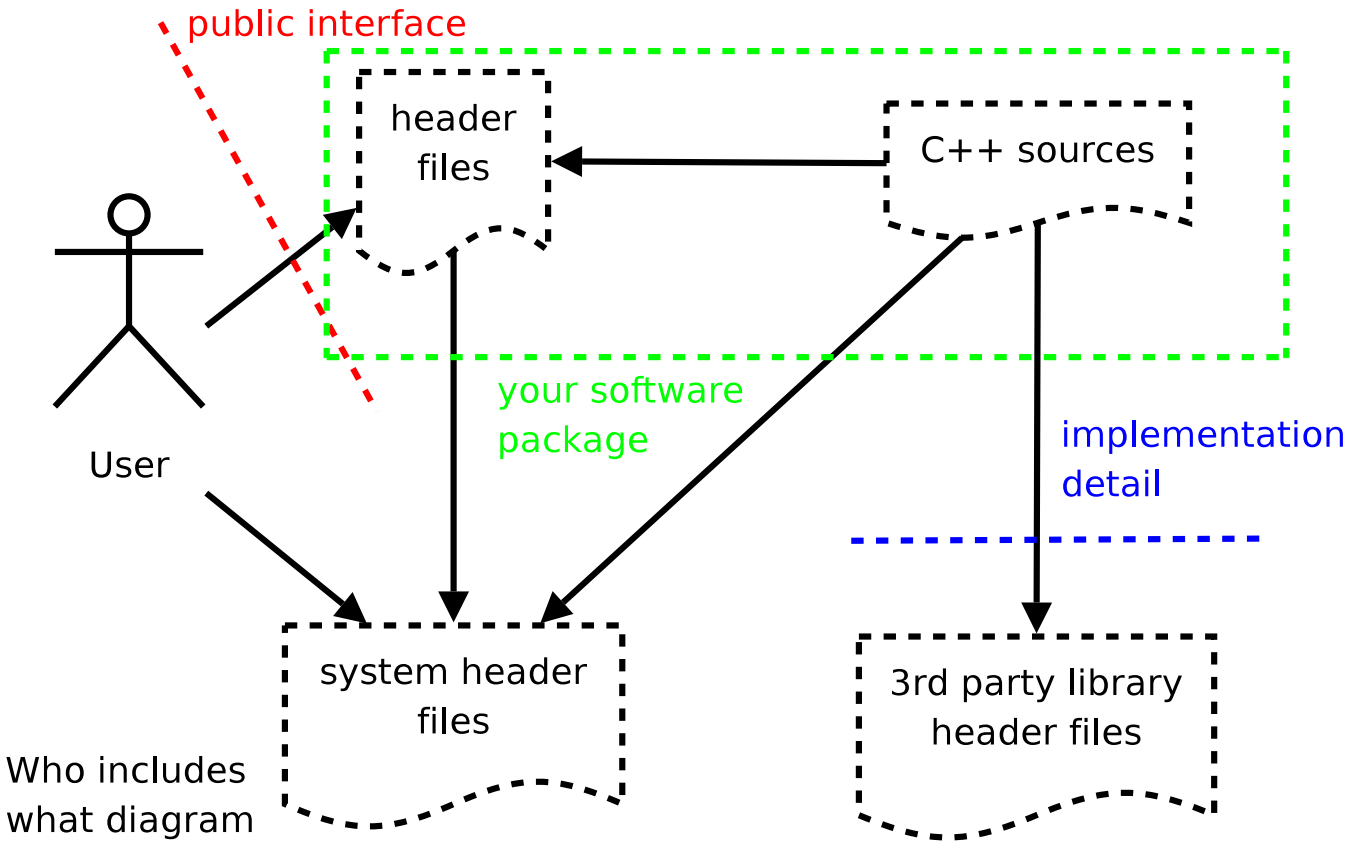
Headers in C++ cont'd

- include guardians needed when included in more than one source file
- inline function will be generated without creating a method, does not always work
- there will be default constructors and destructors if you don't specify them
- headers define interface to class and shall hide implementation details

Implementation in C++

```
#include "myclass.hh"
#include <cstdlib>

myClass::myClass(int seed)
{
        srand(seed);
        _var = rand();
}
void myClass::setVar(int var)
{
        _var = var;
}
```

Include essence



Second most important single slide!

Function overloading

Different functions having the same name, resolved at compile time.

- return value not included to differ functions
- Problem: name mangling, use one compiler for whole source tree

Polymorphic functions

Class child is subclass of parent and overloads a method. Using a pointer to the parent call but having a child instance needs ability to resolve right function a run time.

- vtable implementation
- additional cost at runtime for each call

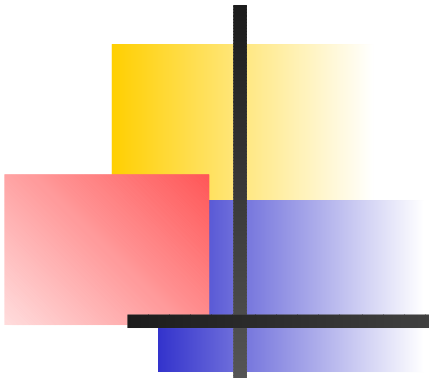
In C++ you can control these costs, in Java not.

Virtual functions

```
#ifndef virtual_hh
#define virtual_hh
class myVirtualClass    {
        public:
                virtual bool polymorphicMember();
                void nonPolymorphicMember();
};
#endif
```

A method is called pure virtual if there is not implementation for this method. You have to mark methods as pure virtual explicitly.

```
virtual int method() = 0;
```



Inheritance

C++ has complex inheritance support. Supports multiple inheritance and non public inheritance.

- public inheritance is most important and does not change access specifiers of base class
- Java like interfaces with pure virtual classes in C++
- methods to be overloaded must be virtual!
- you **have to** supply a virtual destructor for base classes!

Example

```
#ifndef inher_hh
#define inher_hh
#include <string>
class myBase    {
        public:
                virtual ~myBase() { };
                virtual bool abstract();
};
class javaInterfaceLike {
        public:
                virtual ~javaInterfaceLike { }
                virtual int pureVirtual() = 0;
                virtual bool myBool(std::string) = 0;
};
class myClass : public myBase, public javaInterfaceLike {};
#endif
```

Objectsemantics

C++ objects differ from Java objects in the following aspects:

- C++ objects may be created on the stack!
- operator= normally copies an object, in Java you just have a reference
- C++ objects created with operator new must be deleted manually (no garbage collection)
- destructor is called when object is destroyed, guaranteed (no garbage collection :)

Perhaps most important single slide!

C++ pointers

Pointers are a way to access objects indirectly. A pointer points to an object. The content of a pointer is a memory address where an objects exists. There is the NULL pointer pointing to no valid object.

```
int i = 42;
int j = 23;
int* pi;

pi = &i;                  // &...addressoperator
*pi = 5;                  // *...dereference operator
pi += sizeof(int);        // *pi == 23 now (pointing to j)
pi = 0;                   // pi is a NULL pointer now
```

operator new

```
int* pi;

pi = new(int);  // create new object on heap
*pi = 23;
pi = new(int);  // memory leak!
*pi = 42;
delete pi;      // freeing memory
```

C++ reference

```
int i = 23;
int& ri = i;    // every change of ri changes i now!
int& ui;        // uninitialized reference not allowed!

int function(std::string& reference, std::string copy);

std::string str("42 towels");
function(str, str);     // ok, string gets copied once
function(str, "test");  // ok, "test" gets copied
function("test", str);  // error, "test" not const
```

Use reference when you want to return values from function, use const ref if you want to avoid coping data.

```
int function(const std::string&);
```

Stack vs. Heap

```
void function(void)     {
  Object* objr;
  {  Object obj;         // constructor called, on stack
     obj.doSomething(); // call method using . !
     objr = new(Object);// constuctor called, on heap
  }                     // destructor called
  objr->doSomething();  // call method using -> !
  delete objr;          // destructor called, don't forget delete!
}
```

- Access stack objects using .
- -> is a shortcut for (*object).method()
- operator new[] needs delete[] not delete

Exceptions

```
class myException { };
class myClass   {
        public:
                myClass();
                ~myClass() throw();
                void complex() throw(int, myException);
};
```

- ctor may throw every exception
- dtor is defined to not throw exceptions, dtor **MUST NOT** throw exceptions.
- complex may throw int and myException, any other type will cause the program to abort with an exception violation

Catching exceptions

```
try     {
        complex();
} catch (int& i)        {
} catch (myException& e)        {
} catch (...)   {
        // this can't happen, why?
}
```

- You don't have to catch (opposite to Java) anything
- You **should** catch references of the objects...

C++ Summary #1

- C++ is an evolution of C and there you don’t have to use OO techniques
- C++ overloadable methods must be declared **virtual**
- when creating objects on heap using **new** you must free the memory using **delete**
- you must dereference a pointer when accessing its content (*)
- references and addressoperator are the same but are different! (&)
- unexpected exception abort the program

Constructors

There is no super, you link constructors, and even initialize members via linked constructors: see!

```
class myClass : public myBase   {
        private: int i, std::string s;
        public:
                myClass() : myBase(), i(911), s("help") { }
};
```

- You **HAVE TO** call ctors in correct order (same as declaration)
- This code is exception safe!

Prevent object copy

You can prevent the copying of objects by declaring all ctors of a class to be private.

```
class NonCopy   {
        private:
                NonCopy();
        public:
                static NonCopy& instance();
};
```

Copy ctor

You have to define a so called copy ctor to handle copying of classes with pointer members!

```
class CopyCtor  {
        private:
                int *p;
        public:
                CopyCtor(const CopyCtor& tocopy)
                {
                        p = new int;
                        *p = *(tocopy.p);
                }
};
```

Third most important single slide! (actually a consequence of the objectsemantics, but better be explicit in an introduction lecture :)

4+1 casts

C++ knows of 4+1 casts, they are:

- dynamic_cast<ToType>(FromType)
  checks at runtime if cast was successfully, returns 0 if not
- static_cast<ToType>(FromType)
  no runtime checks, use with more caution
- reinterpret_cast<ToType>(FromType)
  overrule compiler's opinion about types
- const_cast<ToType>(FromType)
  remove const from expression
- (ToType)FromType c-style

const everywhere

```
const int* const getMember() const;
int const* const getMember() const;
```

Above lines are semantically equivalent. The meaning of each const from left to right:

- return value pointer to const int
- the pointer itself is constant
- this method does not alter object, and may be called on const objects!

Const correctness

The compiler checks if all const constraints hold and may optimize lot of things.

- call by reference although call by value signature
- directly use value for int const instead of memory access

Problem: data not related to "logic" if locking whole objet $\Rightarrow$ volatile

Operator overloading

In C++ operators are just methods with ”funny” names and therefore you can overload them.

You may not overload:

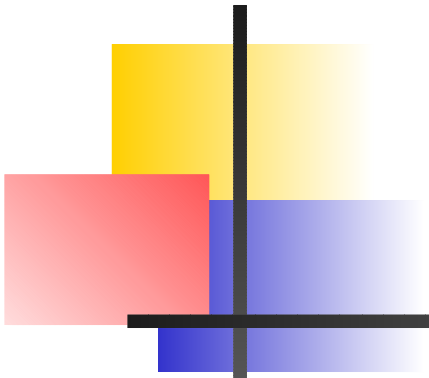
```
::    sizeof    ?:    .
```

You may overload:

```
+     -     *     /     =     <     >     +=
-=    *=    /=    <<    >>    <<=   >>=   ==
!=    <=    >=    ++    --    %     &     ^
!     |     ~     &=    ^=    |=    &&    ||
%=    []    ()    new   delete      ->
```

Namespaces

```
namespace myNamespace {                    //declaration
        class myClass {
                void dummy();
        };
}

void myNamespace::myClass::dummy() {...}//definition

namespace myNamespace {                    //scoped namespace
        myClass::dummy() {...}
}

using namespace myNamespace;    //including namespace globally
myClass::dummy() {...}
```



C++ Summary #2

- don’t forget to write copy ctor when using pointer member field
- use the 4 casts, not c-style casts!
- don’t forget about const to clarify interfaces
- operators may be overloaded
- use namespace, don’t include everything into global namespace

Generic programming

In generic programming types are treated like variables and used as arguments for algorithms. C++ supports generic programming with templates which are evaluated at compile time. Template arguments are written between ”<” and ”>”.

```
template <class T>
T muladd(T a, T b, T c) {
        return a + b*c;
}
```

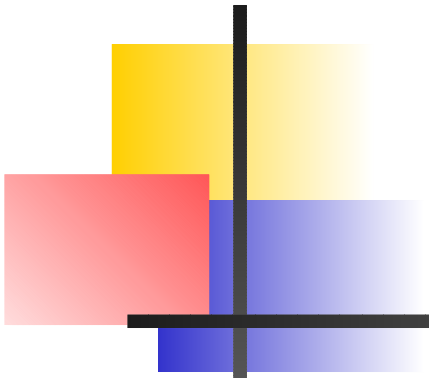
Of course + and * must be defined for used T

Generic programming

One can now use the template like this:

```
int ai=23, bi=42, ci=5;
muladd<int>(ai, bi, ci);

float af=2.3, bf=4.2, cf=0.5;
muladd<float>(af, bf, cf);
```

The compiler generates code at compile time, this means generic algorithm cost space but not runtime.



STL

C++ standard library mostly consists of the STL, a library heavily using templates, hence the name Standard Template Library. The STL offers **typesafe** containers and algorithm. Uses concept of iterators.
Deeper STL is beyond the scope of this lecture, sorry!

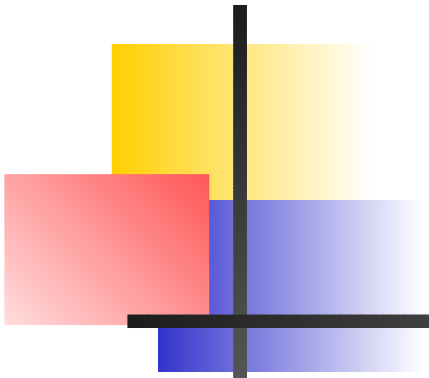
STL example

```
#include <vector>
#include <iostream>

class outInt    {
 public:
  void operator()(const int& i) {std::cout << i << "␣";}
};

int main(void)  {
        std::vector<int> vi;
        outInt oi;      // function object!

        vi.push_back(23);
        vi.push_back(42);
        for_each(vi.begin(), vi.end(), oi);
}
```

STL exmaple 2

Object are either completely constructed or not. There are no half constructed objects. This may be difficult to achieve if exceptions may occur, the auto_ptr helps here. They are garbage collectors, but don’t allow to copy objects.

```
myClass::myClass() throw(int) {
        try     {
                auto_ptr<T1> p1(new T1());
                auto_ptr<T2> p2(new T2());
        }catch (...)    {
                throw 23;       // error code
        }
}
```

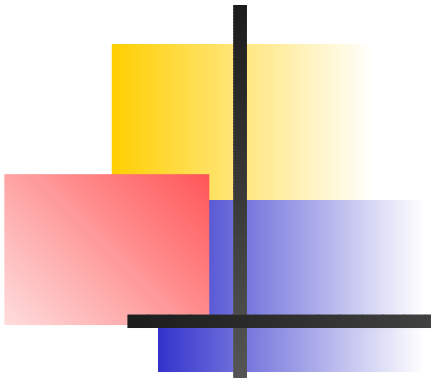


Boost

`www.boost.org` hosts a bunch of high quality C++ libraries.

- smart_ptr - Automatic garbage collection for C++
- spirit - An in source EBNF parser generator
- program_options - Commandline parsing
- regex - Regular expression library

Fourth most important single slide!



That’s all folks!

Questions?