A Brief Introduction to C++ for Java Developers

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy
s.benedettini@unibo.it

Artificial Intelligence 2011

Method

- A C++ introduction for Java developers
  - We will focus on the differences between the two languages
- Very few new theoretical concept
- Enough C++ to use libraries presented in this AI course
- Knowledge of Java and OOP principles is given for granted

Outline

What is C++

- Multiparadigm programming language
  - Procedural
  - Imperative
  - Object-Oriented Programming
  - Functional Programming
  - Generic Programming
  - Metaprogramming
- Compiled
- Statically typed
- Strongly typed (can argue about that)

Motivations

Why C++ is cool

- Lots of stuff available
- It is *very* powerful (more than Java)
- Metaprogrammable
- It is “expert friendly”

Why C++ sucks

- C heritage
- Complex and verbose syntax
- Compilers are stupid
- “Feels” old
  - What? Headers? Are we still in the 70’s?
- No automatic memory management (so what?)
- It is too “expert friendly”

Tools of the Trade

Which platform?

- Language is standardized (C++ $\neq$ Microsoft!)
- Many compilers available
  - each one introduces non-standard extensions
  - which we will neither use or mention
- This course will be Linux-centered
  - GCC

Which tools?

- Any text editor is good
- Eclipse and Netbeans plug-ins are available
- Several specific IDEs as well
- A build system is not required, but it doesn't hurt
  - Autotools (good luck with those!)
  - CMake

References

- Standard library documentation
- Books:
  - The C++ Programming Language
  - Thinking in CPP (suggested reading)
  - The C++ Annotations
  - Many others ranging from introductory to advanced topics
- If you have (and you will) have any doubts, ask guys at Stack**overflow** or browse through the answers
- Your favourite compiler documentation
- Your favourite utility library (mine is Boost)

Outline

Outline

Structure of a Program

- A C++ program code is organized into:
  - Source files (**.cpp**, **.cc**, **.cxx**)
  - Header files (**.hpp**, **.hh**, **.hxx**)
- The one and only entry point of a stand-alone program is its `main` function
- Libraries don't need a `main`
- Header files usually contain declarations of program elements:
  - Global variables
  - Functions
  - Struct and classes
- Headers expose the public interface of a module
- Source files contain definitions (implementations)

“Hello, World!”

Example

Revisited “Hello, World” C++ application

```
#include <iostream>

int someGlobalVar;

void say_it(char str[]) {
  std::cout << "Hello, World! " << str << std::endl;
}

int main(int argc, char* argv[]) {
  someGlobalVar = argc;
  say_it(argv[1]);
  std::cout << "Argument count is " << someGlobalVar
      << std::endl;
  return 0;
}
```

“Hello, World!” example breakdown

- Header inclusion allows one to use functionalities from a different module or library
  - To use the standard stream library

```
#include <iostream>
```

- Global variables and free functions reside outside classes

```
int someGlobalVar;
void say_it(char str[]) { ... }
```

- To write something to standard output
  - Special syntax (employs operator overload)
  - Works for all basic types
  - `cout` is an object representing buffered standard output
  - `endl` puts and end of line in the stream and flushes it
  - Both in `std` namespace

```
std::cout << "Hello, World! " << str << std::endl;
```

- C++ has no first class strings
  - A literal string is an *immutable* array of `chars`

```
char aLiteralString[] = "Hello, World!";
```

Declaration and definition

- Each name you use in a program has to be declared
- Declaration binds a name to a type
- Different kinds of definitions
  - **Variables:** the compiler allocates storage
  - **Functions:** a function body is provided
  - **Classes:** a class body is provided
- **One Definition Rule:** multiple declarations but only one definition allowed

Example

```
int f(int x); // function declaration
class MyClass; // class declaration
int g(double x) { // fun. declaration and definition
  int y = x * x; // var. declaration and definition
  return y;
}
int f(int x) { return 2 * x; } // function definition
class MyClass { /* ... */ }; // class definition
```

Variables

- A variable is the name of a stack memory location which contains an object of a certain type
  - Roughly, a variable is the name for a bunch of bytes
  - Its type tells the compiler how to interpret those bytes
  - A variable *is* the object it refers to
- A variable is tied to a scope
  - Whenever a variable is defined storage space is reserved
  - Whenever a variable goes out of scope that space is re-claimed

```
Type varName = /* initializer expression */;
int x1, x2 = 100, x3; // all share the same type
```

Example

```
{ // beginning of a scope
  double x; // x contains garbage (not 0!)
  x = 9.5; // now x contains 9.5
} // here x's storage is reclaimed
```

Built-in Types

- Roughly the same as Java
  - `boolean` is called `bool`
  - A nonzero integral type is interpreted as `true` while `0` is `false`
- Has `signed` and `unsigned` versions of integral types
  - `int` and `unsigned int`, `short` and `unsigned short`, etc...
- Usual coercion rules apply
- Additionally:
  - Pointers and arrays
  - References
  - Function types, member function types (we won't cover these)
- C++ lacks first-class strings
  - Only string literals and `chars`

```
char s[] = "A string";
char aSpace = ' ', aTabulation = '\t';
```

- Arrays *are not* objects

Expressions, Operators and Statements

- Same as Java

  Operators: arithmetic, logic, bitwise, ternary operator

  Branches: `if`, `switch`/`case`, `goto`

  Loops: `while`, `for`, `break`, `continue`

- Assignment is actually an *expression* that returns the value being assigned

```
if (int* a = (int*) malloc(sizeof(int) * 10)) {
  do_stuff_with_array(a);
} else {
  std::cerr << "Couldn't allocate memory" << std::
      endl;
}
```

Functions

- Basic mechanism of encapsulation and reuse
- Three kinds of functions:

  Free functions: not tied to an object; syntactically defined outside a class

  Non-static member functions: called methods in Java

  Static member functions: called static methods in Java

Function Default Arguments

- Functions can have default arguments
- The must come last in the argument list
- Must be specified in the declaration and *not* in the definition
- If during invocation user leaves some arguments out these are bound to the default value

Example

```
int my_sum(int x, int y, int z = 10, int w = 100) {
  return x + y + z + w;
}

// my_sum(1) // this is an error!
my_sum(1, 2) // calls my_sum(1, 2, 10, 100)
my_sum(1, 2, 3) // calls my_sum(1, 2, 3, 100)
my_sum(1, 2, 3, 4) // calls my_sum(1, 2, 3, 4)
```

Notice that you can't assign `w` without assigning `z` because argument passing is positional

Pointers

- A pointer is the address of a memory location
  - Compare to a variable, which is the name of a memory location
- A pointer type is denoted by the pointee type and a star:

```
int* pointerToAnInt;
Object* pointerToAnObject;
```

- Two new dual operators:

  Address-of (&): takes the address of a variable

  Dereference (*): "follows" the pointer and obtain the pointed to object (pointee)

  - `&x` has the type of x "plus a star at the end"
  - `*x` has the type of x "minus a star at the end"

```
int x = 10;
int* ptrX = &x;
int** ptrPtrX = &ptrX;
assert( *ptrX == 10 );
assert( *(&x) == 10 ); // cancel each other out
```

"Famous Mind-bending Pointer Example"™

- Why does it work?

```
void swap(int* a, int* b) {
  int temp = *a;
  *a = *b;
  *b = temp;
}

int x = 10, y = 20;
swap(&x, &y);
assert( x == 20 && y == 10 );
```

- Usually you are not (and you should not be) interested in a pointer's actual value
  - Let's forget about pointer arithmetic
  - In C++ there are a better options: references

Static Arrays

- Practically an indexed variable
- A contiguous area of memory
- Each element is initialized with its default initializer

```
int a[3] = {10, 20, 30};
int b[] = {9, 10, 11}; // size is inferred
assert( a[0] == b[1] );
int x = get_int_value();
int c[x]; // contains garbage
```

- An array variable is an immutable pointer (cannot point to something else)

```
std::cout << a << ' ' << &a << std::endl // print
    the same string twice
assert( a[0] == *a);
assert( *(b + 2) == b[2] ); // ptr. arithmetic is
    ugly
```

Dynamic Memory

- When you don't know how many things you are dealing with, you need dynamic memory
- In C you call `malloc` to allocate chunks of memory on the heap

```
float* ptrFloat = (float*) malloc(sizeof(float));
```

- Not type-safe and error prone
- C++ elegant solution: `operator new`

```
double* ptrDouble = new double;
MyObjectType* o = new MyObjectType;
```

- Rationale: the compiler knows how big are your objects
- Objects can also be initialized

```
double* pd = new double; *pd = 3.14;
double* p2d = new double(3.14);
assert( *pd == *p2d );
```

- No `null` keyword; use `0` or `NULL` macro instead

Dynamic Arrays

Foreword

- Don't use arrays in C++
  - No bound checks
  - Must be manually deleted (not true for static ones)
  - They are not objects
  - Difficult to pass around
- Learn to use a `std::vector` instead

Usage

- Use `operator new` to allocate more than one object in a contiguous memory chunk
- Compare C and C++ versions of the same code:

```
int* arrayInC = (int*) malloc(sizeof(int) * 100);
int* arrayInCpp = new int[100];
```

Manual Memory Management

- Space allocated on the stack is automatically reclaimed when variables/static arrays go out of scope
- Space allocate in the heap must be manually freed
- Enter `operator delete` and `operator delete[]`

  delete reclaims memory allocated via `new`

  delete[ ] reclaims memory allocated via `new[]`

```
int* pi = new int(100);
delete pi;
int* arr = new int[100];
delete[] arr; // dimension is not needed
// delete arr; // is undefined behaviour
```

- Other undefined/unsafe behaviours
  - Deleting a pointer not returned by a `new`
  - Double delete
  - Although deleting a null pointer is a no-op

const

- `const` is a type decorator much like `*` is for pointers
- A `const` variable is immutable
  - The memory region it represents cannot be be modified
  - Immutability at the *bit level*
  - `const` variables cannot be assigned

```
const int aConst = 100; // before type annotation
// ++aConst; aConst += 10; aConst = 99; // errors
int const anotherConst = -12; // after type
    annotation
```

- If you know you will not change a variable, make it `const`
- I can always convert from a non-`const` type to a `const` type and vice versa

const pointer types

- Pointer to const object:
  - Pointee cannot be modified

```
int x = 100;
const int* ptr1 = new int(10);
int const* ptr2 = &x;
// *ptr1 = 12; ++*ptr1; // errors
ptr2 = ptr1; // can change pointer
*ptr2 = 101; // can't change x through pointer
```

- const pointer to object:
  - Pointer cannot be modified (can be still deleted)

```
int x = 20;
int* const ptr = new int(10);
// ptr = &x; // error
*ptr = x; // can change pointee
```

- Beware of conversions

```
const int x = 10;
const int* px = &x;
// int* px2 = &x; // could change x through pointer
// int* px3 = px; // same as above
```

References

- Basically, references are `const` pointers
- Syntactically, references behave like variables

```
int x = 100;
int& rx = x;
++rx;
assert( x == 101 );
```

- Equivalent to:

```
int* const px = &x;
++*px;
assert( x == 101 );
```

- If you assign a reference you change the object it refers to

```
rx = 200;
assert( x == 200 && rx == 200 );
```

- Ampersand is a type decorator just like `const` and `*`

"Famous Mind-bending Pointer Example"™ Take Two

- References are especially useful as function arguments
- Explain why it works:

```
void swap(int& a, int& b) {
  const int temp = a;
  a = b;
  b = temp;
}

void inc(int& x) { ++x; }

int x = 10, y = 20;
swap(x, y);
assert( x == 20 && y == 10 );
inc(x); inc(y);
assert( x == 21 && y == 11 );
```

Outline

Compilation Stages

- Compilation is subdivided in three stages
  1. Preprocessing
  2. Compilation
  3. Linking

Preprocessing + Compilation

Input – Translation unit: a source file and the headers it includes

Output – Object file: contains native code + metadata

Linking

- Takes a bunch of object files and produces:
  - An executable
  - A library (static or dynamic)
- Resolves names (translates names to addresses)

Headers Demystified

- Headers contain the public interface of a library
  - Declarations of types, variables and functions
  - Do not contain definitions — remember ODR!
  - `inline` functions and templates are an exception

Example

- Suppose you write a library containing:

```
unsigned int my_crc32(char* buffer) { /*...*/ }
```

- Problem: users must repeat that function prototype whenever they use `my_crc32` in their code
- Solution:
  - Library writers produce headers with all the required declarations
  - A text preprocessor automatically copy-pastes header contents into source files (`#include` directive)

C++ Preprocessor

- We will use only a small subset of preprocessor features
- Many archaic preprocessor uses are superseded by safer C++ features
  - Compile time constants
  - Generic code

File inclusion: `#include` directive

```
#include <iostream> // relative to include
    directories
#include "mylib/mymodule.hpp" // relative to CWD
```

Include guards: to include an header only once per translation unit

```
// file: someheader.hpp
#ifndef SOMEHEADER_HPP_
#define SOMEHEADER_HPP_
// header code
#endif
```

Outline

Outline

Code Organization with Namespaces

- Unlike Java, there is no concept of package in C++
- Yet you can organize names into hierarchies of namespaces

```
namespace mylib {
int myfun(int x); // function declaration
class MyType {}; // class definition
}
```

- `myfun` and `MyType` live inside `mylib` namespace
- `mylib::myfun` and `mylib::MyType` are their qualified names
- Namespaces can be nested

```
namespace mylib { namespace util {
unsigned int my_crc32(char* buf) { /* ... */ }
} }
```

- Namespaces can span files and translation units
  - In the previous snippet I re-opened `mylib` namespace, possibly in a different file
  - `mylib::util::my_crc32` is its qualified name

`using` directive and Overloads

- Similar to Java `import` directive
- Useful to save typing
- `using namespace Ns;`
  - All names in `Ns` are available unqualified
  - Same as `import Ns.*;` in Java
- `using Ns::somename;`
  - Only `Ns::somename` is available unqualified
  - Same as `import Ns.somename;` in Java
- Watch for ambiguities!

```
namespace ns1 { int f(); void g(double d); }
namespace ns2 { int f(); void g(int n); }
double f();

using namespace ns1;
using ns2::f; using ns2::g;
// f(); // call of 'f' is ambiguous
ns1::f(); // must disambiguate
::f(); // calls 'f' in the default namespace
g(3.14); // calls ns1::g
g(100); // calls ns2::g
```

A Small Example

```
// file: mylib.hpp
namespace mymath {
  double* poly_zeros(double coeff[]);
  namespace signal {
    double* fft(double v[]);
} }

// file: mylib.cpp
namespace mymath {
  double poly_zeros(double coeff[]) { /* ... */ }
}
double* mymath::signal::fft(double v[]) { /* ... */ }

// file: application.cpp
#include "mylib.hpp"
int main(int argc, char* argv[]) {
  using namespace mymath;
  double someVector[] = /* ... */;
  double* zeros = poly_zeros(someVector);
  double* dft = signal::fft(someVector);
}
```

Outline

OOP Overview in C++

Commonalities

- Almost all OOP concepts in Java are directly mapped to C++
  - Inheritance
  - Abstract classes/methods
  - Interfaces (more or less)

Differences

- No automatic memory management makes these operations critical:
  - Construction
  - Destruction
  - Copy and assignment
- Objects can live on the stack or on the heap
- There is no common Object root
  - No `toString`, `clone`, `equals`, `hash` methods for free!
- Multiple inheritance

The `class` Construct

Example

A C++ counter — take 1

```
class Counter {
public:
  Counter() { val = 0; } // default ctor
  Counter(int val) { this->val = val; }
  int getVal() { return val; }
  void setVal(int val) { this->val = val; }
  void inc() { ++val; }
private:
  int val;
  // int val = 10; // error illegal C++; Java only
};
```

Comments on Counter Example

- Visibility declaration refer to group of members
- `class` and `struct` are synonyms except:
  - `class` members have default `private` visibility
  - `struct` members have default `public` visibility
- `this` is a (`const`) pointer to the current object
- Data members (i.e., `val`) cannot be default initialized
- Still this definition is not idiomatic C++
  - Methods are inline
  - No copy-constructor or destructor (actually not needed)
  - It's not `const`-correct

Object Manipulation

- Syntax changes if accessing a member from a pointer or a variable/reference
  - Biggest difference from Java
- Dot operator: access from a variable/reference
- “Arrow” operator: access from a pointer

```
void inc_n(Counter& c, int n) {
  for (int i = 0; i < n; ++i)
    c.inc();
}

Counter* c = new Counter(10);
c->inc(); c->inc();
assert( c->getVal() == 12 );
c->setVal(-10);
inc_n(*c, 10);
assert( c->getVal() == 0 );
```

const-correctness – Designer Perspective

- An accessor method (i.e., “getter”) should not modify an object state
- In C++ we can enforce that with `const` keyword
- In a method marked `const`:
  - You cannot modify data members
  - You cannot invoke non-`const` methods
  - `this` is a pointer to a `const` object
- `const` becomes part of the method signature

```
class Counter {
  // ...
  int getVal() const {
    // ++val; setVal(10); inc(); // errors
    return val;
  }
  // ...
};
```

const-correctness – User Perspective

- User cannot invoke non-const methods
- User cannot modify data members

```
using namespace std;
void print(const Counter& c) {
  // c.setVal(100); // error
  cout << "Counter(" << c.getVal() << ")" << endl;
}
void inc_n(Counter& c, int n) { /* ... */ }

Counter* c = new Counter(10);
print(*c); // prints ‘‘Counter(10)’’
const Counter* c2 = new Counter(20);
print(*c2); // prints ‘‘Counter(20)’’
// inc_n(*c2, 10); // error
c2 = c; // notice memory leak here!
// inc_n(*c2, 10); // error
```

Object Initialization – Problems

- The meaning of the following code is clear:

```
int x = 10;
int y;
int z = x;
y = x;
```

  - A variable represents a memory location
  - Therefore, assignment is a bit copy
  - Also, storage for x and y is reclaimed before exiting local scope

- But what is the interpretation of this code?

```
Counter a = /* initialization */
Counter b; // what is this?
Counter c = a; // and this?
b = a; // assignment
```

- Objects = Data + Behaviour
  - We have to give meaning to the previous expressions

Object Initialization – Default Constructor

- Constructor without arguments
- It is synthesized for you if you don't provide any constructor
  - Just like Java
- Handles default initialization

```
struct A {};
struct B {
  B() { std::cout << "Hello!" << std::endl; }
  B(int x) {}
};
struct C { C(int x) {} };

A a;
B b; // prints ‘‘Hello!’’
// C c; // error: C has not default constructor
```

Object Initialization – Custom Constructor

- The typical constructor with arguments
  - Again, just like Java
  - But supports default arguments, like any function
- Special “functional initialization” syntax

```
using namespace std;
struct A {
  A(int x, int y) { cout << x << ' ' << y << endl; }
};

A a(10, 20); // prints ‘‘10 20’’
// A a2(30); // error: A has not constructor A(int)
A a2 = A(1, 2); // this syntax is also acceptable
```

Object Initialization – Copy Constructor

- Has a specific signature: `T(const T&)`
- A default copy constructor is synthesized for you if you don't provide one
  - Recursively invokes each member copy constructor
  - Basic types are bit-copied
- Handles initialization from another object

```
struct A {
  A(int x) { this->x = x; cout << "A(" << x << ")"
      << endl; }
  A(const A& otherObject) {
    x = otherObject.x + 1;
    cout << "A copy-ctor " << x << endl;
  }
  int x;
};
void f(A a) {}
A a(0);
A a2 = a;
A a3(a);
f(a);
```

Prints:

A(0)

A copy-ctor 1

A copy-ctor 2

A copy-ctor 3

Object Destruction – Destructor

- The destructor is a method called automatically before an object storage is reclaimed
  - When a variable goes out of scope
  - When a `delete` operation is performed
- It is a “special” (like a constructor) parameterless method

```
struct A {
  A() { cout << "A-ctor" << endl; }
  ~A() { cout << "A-dtor" << endl; }
}

{
  cout << "Entering scope\n";
  A a;
  cout << "Exiting scope\n";
}
cout << "Outside scope" << endl;
```

Prints:

Entering scope

A-ctor

Exiting scope

A-dtor

Outside scope

Outline

Inheritance

```
struct Base {
  Base(int x) { cout << "B(" << x << ")" << endl; }
  void method() {}
};
class Derived : public Base {
  public:
    Derived(int x) : Base(x * x) {
      cout << "Derived(" << x << ")" << endl;
    }
};
struct Derived2 : Derived {}; // no public required

Derived d(2); // prints ‘‘Base(4) Derived(2)’’
d.method(); // inherits methods
Base b(d); // copy ctor: Derived ‘‘is a’’ Base
```

- Minor syntactic differences from Java
  - Colon instead of `extend` keyword
  - Different syntax for calling base class constructor
- Remember to call base class constructor!
  - Compiler does that for you in synthesized constructors

Polymorphic Methods

- In Java methods are polymorphic by default
- In C++ you have to explicitly state which methods are polymorphic
- In C++ we have polymorphic dispatch only if:
  - Called method is `virtual`
  - Receiver is a pointer or a reference
- Rule: a base class destructor should always be `virtual`

```
struct Base {
  void f() { cout << "Base" << endl; }
  virtual g() { cout << "Base" << endl; }
};
struct Derived : Base {
  void f() { cout << "Derived" << endl; }
  void g() { cout << "Derived" << endl; }
};

Base* b = new Derived;
b->f(); // prints ‘‘Base’’
b->g(); // prints ‘‘Derived’’
```

Pure Virtual Functions

- Pure Virtual Functions in C++ = Abstract Methods in Java
- Like in Java, a class with pure virtual functions (abstract methods) cannot be instantiated
- Minor syntactic difference

```
struct Base {
  virtual void polymorphic() const;
  void nonPolymorphic();
  virtual abstract() = 0;
};
```

Interface

- C++ doesn't have first class interfaces
- Can be emulated via multiple inheritance and pure virtual functions
  - Interface as a design pattern

```
struct StreamInterface {
  virtual void open() = 0;
  virtual char read() = 0;
  virtual void write(char) = 0;
  virtual void close() = 0;
};
```

Outline

Operator Overload

Foreword

- We are interested in overloading operator only from a user perspective
- We will see how to implement “Stream operator”
  - To write stuff to standard output
  - Idiomatic C++ alternative to Java `toString`

Interpretation

- We can use custom types with C++ operators
- Operator application is translated to method call
- Special hooks are provided to customize operator behaviour
- Normal C++ precedence rule apply
- You can’t define new operators

Operator Overload Interpretation

- Operator application is translated to method call
  - Suppose class A has overloaded plus and times operator

```
A a, b, c;
a + b * c;
a.operator +(b.operator *(c));
```

- To customize operator behaviour, implement special methods
  - In the previous example, a valid (but useless) definition might be

```
struct A {
  A operator+(A a) { return *this; }
  A operator*(A a) { return *this; }
};
```

Overloaded Stream Operator for Classes

- To make an object “streamable”

```
#include <iosfwd>
struct A {
  // class definition
  friend std::ostream& operator<<(std::ostream& out,
       const A& a) {
    // write object 'a' to stream 'out'
    return out; // return the stream
  }
}
```

- `friend` is like *friend stereotype* in UML
- This expression now compiles

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

A a = // ...
cout << "String representation of A is: " << a <<
    endl;
```

An (Almost) Idiomatic Counter Class

Counter declarations

```
// file: Incrementable.hpp
struct Incrementable {
  virtual ~Incrementable() {};
  virtual void inc() = 0;
}

//file: Counter.hpp
#include <iosfwd>
#include "Incrementable.hpp"
struct Counter : Incrementable {
  Counter();
  Counter(int val);
  int getVal() const;
  void setVal(const int val);
  void inc();
  friend std::ostream& operator<<(std::ostream&, const
       Counter&);
private:
  int val; };
```

An (Almost) Idiomatic Counter Class

Counter definitions

```
//file: Counter.cpp
#include "Counter.hpp"
Counter::Counter() { val = 0; }

Counter::Counter(const int val) { this->val = val; }

int Counter::getVal() const { return val; }

void Counter::setVal(const int val) { this->val = val
    ; }

void Counter::inc() { ++val; }

std::ostream& operator<<(std::ostream& out, const
    Counter& c) {
  return out << "Counter(val=" << c.getVal() << ")";
}
```

Templates

- Mechanism to write generic code
- Bears some similarities to Java Generics
- Allows one to write code parametrized w.r.t some types
  - Generic functions
  - Generic classes

```
template<class T> struct MyClass; // class template
template<class T, class U> struct MyType; // struct
    template
template<class T> int fun(T* a, int x); // template
    function
```

- Type parameters must be instantiated

A Pair Template Example

- How to store arbitrary objects into a fixed size container?
  - Remember that there is no common supertype
- Use templates!

```
template<class T, class U> struct Pair {
  T first;
  U second;
  Pair(const T& t, const U& u) {
    first = t; second = u;
  }
}

Pair<int, double> p(100, 0.98);
Pair<int, int> p2(0, 0);
p2.first = p.first + 100;
```

A Brief Introduction to C++ for Java Developers

Stefano Benedettini

DEIS, *Alma Mater Studiorum* University of Bologna, Campus of Cesena, Italy
s.benedettini@unibo.it

Artificial Intelligence 2011