Introduction to C++ for Java users

Leo Liberti

DIX, École Polytechnique, Palaiseau, France

I semester 2006

Preliminary remarks

Teachers

Leo Liberti: liberti@lix.polytechnique.fr.
TDs: Pietro Belotti belotti@elet.polimi.it,
Claus Gwiggner gwiggner@lix.polytechnique.fr

Aim of the course

Introducing C++ to Java users

Means

Develop a simple C++ application which performs a complex task

http://www.lix.polytechnique.fr/~liberti/teaching/c++/fromjava/

Course structure

Timetable

Lecture: Wednesday 20/9/06 14:00-15:30, Amphi Pierre Curie
TDs: 21-22/9: 8-10, 10:15-12:15, 13:45-15:45, 16-18, S.I. 34-35
25/9/06: 8-10, 10:15-12:15, S.I. 34-35

Course material (optional)

- Bjarne Stroustrup, *The C++ Programming Language*, 3rd edition, Addison-Wesley, Reading (MA), 1999
- Stephen Dewhurst, *C++ Gotchas: Avoiding common problems in coding and design*, Addison-Wesley, Reading (MA), 2002
- Herbert Schildt, *C/C++ Programmer's Reference*, 2nd edition, Osborne McGraw-Hill, Berkeley (CA)

Course contents

Syllabus

1. Introduction
   - Programming style issues
   - Main differences
   - Development of the first program
2. Memory management
   - Pointers
   - Memory allocation/deallocation
3. Standard Template Library
   - Input and output
4. Classes and templates
   - Inheritance and embedding
   - Interfaces
   - Templates

Course contents

Application (developed during TDs)

- **WET** (WWW Exploring Topologizer)
- Graph representation of the World Wide Web
- Explores local neighbourhood of a given URL
- Outputs the graph in a format that can be displayed graphically

Didactical value

- Sufficiently complex software architecture, easy code
- Coded during the practicals as a series of separate exercises

Indentation

- **Absolutely necessary for the programmer / maintainer**
- **ONE STATEMENT PER LINE**
- After each opening brace {: new line and tab (2 characters)
- Each closing brace } is on a new line and "untabbed"

```
double x, y, z, epsilon;
...
if (fabs(x) < epsilon) {
  if (fabs(y) < epsilon) {
    if (fabs(z) < epsilon) {
      for(int i = 0; i < n; i++) {
        x *= y*z;
      }
    }
  }
}
```

Comments

- **Absolutely necessary for the programmer / maintainer**
- One-line comments: introduced by //
- Multi-line comments: /* ... */
- Avoid over- and under-commentation
- Example of over-commentation

```
// assign 0 to x
double x = 0;
```

- Example of under-commentation

```
char buffer[] = "01011010 01100100";
char* bufPtr = buffer;
while(*bufPtr &&
   (*bufPtr++ = *bufPtr == '0' ? 'F' : 'T'));
```

C++/Java: main differences

- Java is a *byte-compiled* language, C++ is *fully compiled* ($\Rightarrow$ C++ is faster)
- Java *requires* the use of classes, C++ may also be used in "old fashion" procedural style
- In Java, no code is ever outside classes; in C++ some code (namely, the `main()` function) must be outside classes
- C++ lets you access memory directly through *pointers*, Java has no pointer mechanism worthy of note
- C++ has a more fine-grained memory management (allocation/deallocation)
- C++ programs usually employ classes/algorithms from the Standard Template Library (STL)
- Some differences in class inheritance
- C++ employs *templates* for generic programming (Java has the `Object` data type)

Building

- The translation process from C++ code to executable is called *building*, carried out in two stages:
  1. *compilation*: production of an intermediate object file (`.o`) with unresolved external symbols
  2. *linking*: resolve external symbols by reading code from standard and user-defined libraries
- Can perform both in one go with the GNU command `c++`:
  `c++ -o helloworld helloworld.cxx`
- Can perform separately: `c++ -c helloworld.cxx` (produces `helloworld.o`),
  `c++ -o helloworld helloworld.o` (useful for combining multiple object files into one executable)

Debugging

- GNU/Linux debugger: `gdb`
- Graphical front-end: `ddd`
- Designed for Fortran/C, not C++
- Can debug C++ programs but has troubles on complex objects (use the "insert a print statement" technique when `gdb` fails)
- Memory debugger: `valgrind` (to track pointer bugs)
- In order to debug, compile with -g flag:

```
c++ -g -o helloworld helloworld.cxx
```

- More details during labs

Packaging and Distribution

- For big projects with many source files, a `Makefile` (detailing how to build the source) is essential
- Documentation for a program is **absolutely necessary** for both users and maintainers
- Better insert a minimum of help within the program itself (to be displayed on screen with a particular option, like `-h`)
- A `README` file to briefly introduce the software is usual
- There exist tools to embed the documentation within the source code itself and to produce Makefiles more or less automatically
- UNIX packages are usually distributed in tarred, compressed format (extension `.tar.gz` obtained with the command `tar zcvf` *directoryName*`.tar.gz` *directoryName*

The first C++ program

```
/*********************************************
* Name:       helloworld.cxx
* Author:     Leo Liberti
* Source:     GNU C++
* Purpose:    hello world program
* Build:      c++ -o helloworld helloworld.cxx
* History:    060818 work started
*********************************************/

#include<iostream>

int main(int argc, char** argv) {
  using namespace std;
  cout << "Hello World" << endl;
  return 0;
}
```

The first C++ program

- Each executable program coded in C++ must have one function called **main()**
  `int main(int argc, char** argv);` **outside all classes**
- The `main` function is the entry point for the program
- It returns an integer *exit code* which can be read by the shell that launched the program
- The integer `argc` contains the number of arguments on the command line
- The array of character arrays `**argv` contains the arguments: the command `./mycode arg1 arg2` gives rise to the following storage:

  `argv[0]` is a `char` pointer to the char array `./mycode`
  `argv[1]` is a `char` pointer to the char array `arg1`
  `argv[2]` is a `char` pointer to the char array `arg2`
  `argc` is an `int` variable containing the value 3

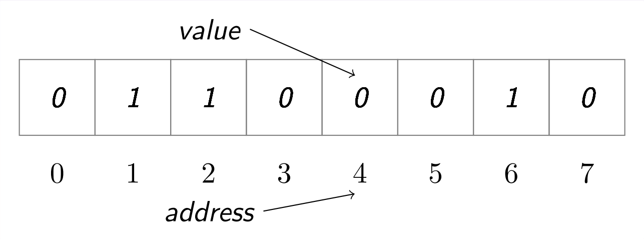
The first C++ program

- C++ programs are stored in one or more text files
- Source files: contain the C++ code, extension `.cxx`
- Header files: contain the declarations which may be common to more source files, extension `.h`
- Source files are compiled
- Header files are included from the source files using the *preprocessor directive* `#include` (like `import` in Java)

```
#include<standardIncludeHeader>
#include "userDefinedIncludeFile.h"
```

Memory

- Usual representation for memory: indexed array of cells where values can be stored



- unit of measure: **bit** (Binary digIT) — can hold a 0 or a 1
- 8b (bit) = 1B (byte), 1024 B = 1 KB (Kilobyte), 1024 KB = 1MB
- sometimes find 1KB = 1000 B and 1MB = 1000 KB ($= 1 \times 10^6$ bytes)

Pointers

- retrieve the address of a variable:

  pointerName = &variableName ;

```
int* pi;
pi = &i;
```

- retrieve the value stored at an address:

  *variableName = *pointerName ;*

```
int j;
j = *i;
```

- using pointers as arrays:

```
const int bufferSize = 10;
char buffer[bufferSize] = "J. Smith";
char* bufPtr = buffer;
while(*bufPtr != ' ') {
  bufPtr++;
}
std::cout << ++bufPtr << std::endl;
```

Pointer warnings

- Pointers allow you to access memory directly, hence can be very dangerous
- Attempted memory corruption results in "segmentation fault" error and abort, or garbage output, or unpredictable behaviour
- Most common dangers:
  1. writing to memory outside bounds

```
char buffer[] = "LeoLiberti";
char* bufPtr = buffer;
while(*bufPtr != ' ') {
   *bufPtr = ' ';
   bufPtr++;
}
```

  2. deallocating memory more than once
- Pointer bugs are usually very hard to track

Using object pointers

- Suppose `myObject` is a pointer to a `MyClass` object, and that `MyClass` has a method `void MyClass::update(void);`
- Two equivalent ways to call this method:
  1. `(*ttsPtr).update();`
  2. `ttsPtr->update();`
- Prefer second way over first

The stack and the heap

- Executable program can either refer to near memory (the *stack*) or far memory (the *heap*)
- Accessing the stack is **faster** than accessing the heap
- The stack is **smaller** than the heap
- Variables are allocated on the stack `double myDouble;`
- Common bug (but hard to trace): **stack overflow**
  `char veryLongArray[1000000000];`
- Memory allocated on the stack is deallocated automatically at the end of the scope where it was allocated (closing brace })
- Memory on the heap can be accessed through *user-defined memory allocation*
- Memory on the heap must be deallocated explicitly, otherwise *memory leaks* occur, exhausting all the computer's memory
- Memory on the heap **must not be deallocated more than once** (causes unpredictable behaviour)

User-defined memory allocation

- Operator `new`: allocate memory from the heap

  pointerType pointerName* = `new` *pointerType* ;

  ```
  MyClass* myObject = new MyClass;
  ```

- Operator `delete`: release allocated memory

  `delete` *pointerName*;

  delete myObject;

- Commonly used with arrays in a similar way:

  pointerType pointerName* = `new` *pointerType* [*size*] ;

  ```
  double* positionVector = new double [3];
  ```

  `delete []` *pointerName* ;

  ```
  delete [] positionVector;
  ```

- **Improper user memory management causes the most difficult C++ bugs!!**

The STL

- Collection of generic classes and algorithms
- Born at the same time as C++
- Well defined
- Very flexible
- Reasonably efficient
- Use it as much as possible, do not reinvent the wheel!
- Documentation: `http://www.sgi.com/tech/stl/`
- Contains:
  - Classes: `vector`, `map`, `string`, I/O streams, ...
  - Algorithms: `sort`, `swap`, `copy`, `count`, ...

vector example

```
#include<vector>
#include<algorithm>
...
using namespace std;
vector<int> theVector;
theVector.push_back(3);
theVector.push_back(0);
if (theVector.size() >= 2) {
   cout << theVector[1] << endl;
}
for(vector<int>::iterator vi = theVector.begin();
      vi != theVector.end(); vi++) {
   cout << *vi << endl;
}
sort(theVector.begin(), theVector.end());
for(vector<int>::iterator vi = theVector.begin();
      vi != theVector.end(); vi++) {
   cout << *vi << endl;
}
```

map example

```
#include<map>
#include<string>
...
using namespace std;
map<string, int> phoneBook;
phoneBook["Liberti"] = 3412;
phoneBook["Baptiste"] = 3800;
for(map<string,int>::iterator mi = phoneBook.begin();
      mi != phoneBook.end(); mi++) {
   cout << mi->first << ":  " << mi->second << endl;
}
cout << phoneBook["Liberti"] << endl;
cout << phoneBook["Smith"] << endl;
for(map<string,int>::iterator mi = phoneBook.begin();
      mi != phoneBook.end(); mi++) {
   cout << mi->first << ":  " << mi->second << endl;
}
```

Streams

- Data "run" through *streams*
- Stream types: input, output, input/output, standard, file, string, user-defined

outputStreamName << *varName or literal . . .* ;

```
std::cout << "i = " << i << std::endl;
```

inputStreamName >> *varName* ;

```
std::cin >> i;
```

```
stringstream buffer;
char myFileName[] = "config.txt";
ifstream inputFileStream(myFileName);
char nextChar;
while(inputFileStream && !inputFileStream.eof()) {
   inputFileStream.get(nextChar);
   buffer << nextChar;
}
cout << buffer.str();
```

Object onto streams

- Complex objects may have a complex output procedure
- **Example**: suppose we have a class called `TimeStamp` which reads the system clock (method `update()`), and produces the time when asked (method`get()`)

  ```
  class TimeStamp {...};
  ```
- We create an object of this class

  ```
  TimeStamp theTimeStamp;
  ```
- We would like to be able to

  `cout << theTimeStamp << endl;` and get

  Thu Sep 7 12:23:11 2006 as output
- Solution: overload the << operator

  ```
  std::ostream& operator<<(std::ostream& s, TimeStamp& t)
     throw (TimeStampException);
  ```

Object onto streams II

```
#include <ctime>
std::ostream& operator<<(std::ostream& s, TimeStamp& t)
   throw (TimeStampException) {
   using namespace std;
   time_t theTime = (time_t) t.get();
   char* buffer;
   try {
      buffer = ctime(&theTime);
   } catch (...)  {
      cerr << "TimeStamp::updateTimeStamp():  "
            "couldn't print system time" << endl;
      throw TimeStampException();
   }
   buffer[strlen(buffer) - 1] = '\0';
   s << buffer;
   return s;
}
```

Example: nested inheritance

- Consider a corporate personnel database
- Need class Employee;
- Certain employees are "empowered" (have more responsibilities): need class Empowered : public Employee;
- Among the empowered employees, some are managers: need class Manager : public Empowered;
- `Manager` contains public data and methods from `Empowered`, which contains public data and methods from `Employee`

Example: nested inheritance

```
class Employee {
  public:
    Employee();
    ~Employee();
    double getMonthlySalary(void);
    virtual void
        getEmployeeType(void);
};
```

←

```
class Empowered : public Employee {
  public:
    Empowered();
    ~Empowered();
    bool isOverworked(void);
    virtual void
        getEmployeeType(void);
};
```

↑

```
class Manager : public Empowered {
  public:
    Manager();
    ~Manager();
    bool isIncompetent(void);
    virtual void
        getEmployeeType(void);
};
```

Example: nested inheritance

It is possible to write a function such as:

```
void use(Employee* e) {
   e->getEmployeeType();
}
```

and then call it using Employee, Empowered, Manager objects:

```
Employee e1;
Empowered e2;
Manager e3;
Employee* e1Ptr = &e1; // all pointers to Employee base class
Employee* e2Ptr = &e2;
Employee* e3Ptr = &e3
use(e1Ptr); // output:  Employee
use(e2Ptr); // output:  Empowered
use(e3Ptr); // output:  Manager
```

Being or having an object?

- Consider example of a salary object:

```
class Salary {
    Salary();
    ~Salary();
    void raise(double newSalary);
    ...
};
```

- Might think of deriving `Employee` from `Salary` so that we can say `theEmployee.raise();` to raise the employee's salary
- Technically, nothing wrong
- Architecturally, very bad decision!
- Rule of thumb:
  derive B from A only if B can be considered as an A
- In this case, better embed a `Salary` object as a data field of the `Employee` class

Pure virtual classes

- Java equivalent: `interface`
- All classes derived from the interface implement the interface methods as declared in the interface
- Can guarantee the formal behaviour of all derived objects
- In C++, an interface is known as a *pure virtual class*: a class consisting only of method declarations and no data fields
- A pure virtual class has no constructor — no object of that class can ever be created (only objects of derived classes)
- A pure virtual class may have a virtual destructor to permit correct destruction of derived objects
- All methods (except the destructor) are declared as follows:
  returnType methodName(*args*) `= 0;`
- All derived classes **must** implement all methods

Pure virtual classes

```
class EmployeeInterface {
 public:
   virtual ~EmployeeInterface() { }
   virtual void getEmployeeType(void) = 0;
};
```

```
class Employee : public virtual EmployeeInterface {...};
class Empowered : public Employee, public virtual EmployeeInterface {...};
class Manager : public Empowered, public virtual EmployeeInterface {...};
```

```
void use(EmployeeInterface* e) {...}
...
use(&e1); // output: Employee
use(&e2); // output: Empowered
use(&e3); // output: Manager
```

- Code behaves as before, but clearer architecture
- `public virtual` inheritance: avoids having many copies of `EmployeeInterface` in `Empowered` and `Manager`

User-defined templates

- *Situation*: action performed on different data types
- *Possible solution*: write many functions taking arguments of many possible data types.
- *Example*: swapping the values of two variables

```
void varSwap(int& a, int& b);
void varSwap(double& a, double& b);
...
```

- Potentially an unlimited number of objects ⇒ invalid approach
- Need for *templates*

```
template<class TheClassName> returnType functionName(args);
```

```
template<class T> void varSwap(T& a, T& b) {
   T tmp(b);
   b = a;
   a = tmp;
}
```

User-defined templates

Behaviour with predefined types:

```
int ia = 1;
int ib = 2;
varSwap(ia, ib);
cout << ia << ", " << ib << endl; // output:  2, 1
```

```
double da = 1.1;
double db = 2.2;
varSwap(da, db);
cout << da << ", " << db << endl; // output:  2.2, 1.1
```

User-defined templates

Behaviour with user-defined types:

```
class MyClass {
  public:
    MyClass(std::string t) :  myString(t) { }
    ~MyClass() { }
    std::string getString(void) { return myString; }
    void setString(std::string& t) { myString = t; }
  private:
    std::string myString;
};
```

```
MyClass ma("A");
MyClass mb("B");
varSwap(ma, mb);
cout << ma << ", " << mb << endl; // output:  B, A
```

Internals and warnings

- Many hidden overloaded functions are created **at compile-time** (one for each argument list that is actually used)
- Very difficult to use debugging techniques such as breakpoints (which of the hidden overloaded functions should get the breakpoints?)
- **Use sparingly**
- But use the Standard Template Library as much as possible (already well debugged and very efficient!)