Object Oriented Programming
USTH, Master ICT, year 1

Aveneau Lilian

lilian.aveneau@univ-poitiers.fr
XLIM/ASALI/IG, CNRS, Computer Science Department
University of Poitiers

2017/2018

Lecture 1 – Introduction

- Generalities about this TU
- Generalities about C++
- Incompatibilities between C++ and C
- Conversational Input-Output
- First program

Objectives

- Object Oriented Programming
  - Basic notions: Classes, Objects ...
  - UML: notation, *etc.*
- C++ language
  - Differences (+ and -) with respect to C language
  - Class, Constructor, Destructor
  - Simple inheritance, multiple
  - Polymorphism
  - Class/Method patterns
  - Exceptions
  - STL: Standard Template Library
  - ...

Some clarifications ...

Student work

- Lecture
  - Add your own notes and comments
- Labwork
  - Do as much exercises as possible
  - Finish them all before examinations

Divers

- We assume you already know C language
- This lecture concerns only C++
- Some books:
  - B. Eckel, "Thinking in C++, v1 + v2"
  - Any book about C++ and design patterns

OOP – 1

Programming problem

Produce software of quality:

- exactitude, robustness
- extensible, reusable
- portable, efficiency

Problem for large projects (some humans/year)

Structured programming

- Natural limit: the Wirth equation
  - Algorithms + Data structures = Program
- Structures programs, ameliorates robustness and exactness
- Impossible to reuse or adapt the code
  - Break module if data structure is changed
  - The fault to Wirth equation!

OOP – 2

What OOP offers

- The notion of *object*
  - OOP equation: Methods + Data = Object
- Data encapsulation, pure OOP
  - Users cannot directly access to data: they must use methods (send a message)
  - Interest: object seen through its method specifications (contract)
  - We speak about data abstraction
  - Increase software quality: facilitate maintenance and reutilisation
- Concept of *class*
  - Generalization of the notion of data type
  - Describes the set of objects having a common structure and same methods
  - Object: a variable of type class, aka instance

OOP – 3

What OOP offers (end)

- Inheritance
  - Define new class from an existing class, by adding new data or methods
  - New class inherits properties and behavior of its ancestors
  - New class specializes the old ones
  - Can be done as many times as necessary
- Polymorphism
  - Derived class can redefine (modify) some inherited methods from base class
  - Thus possibility to consider different objects in a common way
  - Improve extensibility of programs: adding new objects into pre-established scenarios

Careful about C++

Built upon C language, do not demand a strict application of the OOP principles ...

C++ specifics

Differences with ANSI-C

- Function definition
- Operator `const` meanings
- Through pointers compatibility
- Stream notion for user input/ouput

C++ adding

- New comments
- Variable declaration are free
- New notion: reference (value transmission of arguments)
- Over definition of functions (same name, different signatures)
- Different dynamic memory management (`new` and `delete`)
- Inline functions (improve preprocessor ones)

C++ and OOP – 1

Class notion

Includes:

- Description of data structure (data members)
- Methods (function members)

An instance can be created:

- In usual declarations
- Dynamic allocation with `new` operator

Must expose one or more constructors:

- Function member, launched at object creation
- More or less complex, depending on class nature!

Must also present one (at most) destructor!

C++ and OOP – 2

Other OOP special features

- Operator overdefinition (but programs are no more readable)
- Implicit or explicit conversion also for classes
- Simple and multiple Inheritance
- Polymorphism (via virtual functions)
- Stream and IO operators (<< and >>)
- Friend functions

Function definitions with C++

Goal of this section

Describe what C++ does not allow regarding old C ...

With C, 2 ways

```
/* Kerninghan and Ritchie */
double fexple (u, v)
int u;
double v;
{
  ...
}
```

```
/* ANSI C*/
double fexple (int u, double v)
{
  ...
}
```

C++

Permits only the second way

Remark: `int` return value may be silent (discouraged)

C++ function prototypes

Using C

To use previously undefined function, we may:

- Not declaring it (implicitly, return value is `int`)
- Only indicating its return type (*eg.* `double fexple;`)
- Declare it using its prototype: `double fexple(int,double);`

Using C++

All functions prototype must be declared before to be used

Attention to implicit conversions

Compiler will try to convert (potentially with some deterioration) the arguments to stick one of previously defined prototypes (overloading)

Remark

May name prototypes arguments to ease comments/documentation

Function arguments and return value

Similarities: arguments and return value may

- Not exist
- Be a *scalar* value of fundamental data-type
- Be a *structure* value of abstract data-type

Remarks:

- C++ extends structure behavior to classes
- Difference between structure and array (pointer)

Differences between C and C++

Concern only the syntax of header and prototypes following 2 cases:

1. Functions without argument
   - Using C, we use `void`; using C++ we use nothing
2. Functions without return value
   - Using C, we may use `void`; using C++ we MUST use `void`

The const qualifier

Introducted into ANSI C, it exists some differences with C++ ...

Range

- Automatic local variable: no difference
- Global variable: C++ limits to source file containing (idea: to replace the use of `#define`)

Usage into expression

Constant expression: value is calculated during compile
With the following code:

```
const int p = 3;
```

Expression `2 * p * 5` is constant in C++, not in C
Utility: to declare arrays

C++ conceived to limit at the maximum preprocessor usage

Compatibility between void* and other pointers

Implicit conversions disappear!

```
void* gen;
int *adi;
...
gen = adi; /* always licit */
adi = gen; /* licit in C, not in C++ */
```

Only void* conversion is authorized! We write:

```
adi = (int*)gen; // Explicit conversion, cast operator
```

Why this difference

- Passage from void* address to data-type pointer: associate type to an address
- To force compiler to modify starting address (to respect alignment constraint, eg. SSE data-type __m128)

Generalities

C procedures can still be used: include `<cstdio>` and `<cstdlib>`

C++ offers another possibility:

- Simple to use (no more format string)
- Object module size is reduced (less instructions)
- Extensible to user classes

Display: some examples

Uses stream notion: `std::cout`

```
std::cout << "Hello world!" << std::endl; // operator <<, endl for ending line

int n = 25;
std::cout << n << std::endl;              // same operator, with overloading

std::cout << "Value: " << n << std::endl; // << returns cout, it can be reused!
```

All is defined into the namespace `std`, but we may simplify the code by adding "`using namespace std;`" in the beginning of the module

Display, second example

```cpp
#include <iostream>
using namespace std ;
int main() {
  int n=25;
  long p=250000;
  unsigned q=63000;
  char c='a';
  float x = 12.3456789 ;
  double y = 12.3456789e16 ;
  const char * ch = "hello" ;
  int* ad= &n;
  cout<<"value of n   : " <<n <<"\n";
  cout<<"value of p   : " <<p <<"\n";
  cout<<"character c  : " <<c <<"\n";
  cout<<"value of q   : " <<q <<"\n";
  cout<<"value of x   : " <<x <<"\n";
  cout<<"value of y   : " <<y <<"\n";
  cout<<"string ch    : " <<ch<<"\n"; // particular case, display a pointer
  cout<<"address of n : "<<ad<<"\n"; // standard case, hexadecimal display
  cout<<"address of ch: "<<(void*)ch<<"\n"; // idem
}
```

Print format

```
#define D(A) cout << #A << endl; A // Display A as a text, THEN execute A

#include <iostream>
using namespace std;

int main() {
  D(int i = 47;)
  D(float f = 2300114.414159;)
  const char* s = "Is there any more?";

  D(cout.setf(ios::unitbuf);)
  D(cout.setf(ios::showbase);)
  D(cout.setf(ios::uppercase | ios::showpos);)
  D(cout << i << endl;) // Default is dec
  D(cout.setf(ios::hex, ios::basefield);)
  D(cout << i << endl;)
  D(cout.setf(ios::oct, ios::basefield);)
  D(cout << i << endl;)
  D(cout.unsetf(ios::showbase);)
  D(cout.setf(ios::dec, ios::basefield);)
  D(cout.setf(ios::left, ios::adjustfield);)
  D(cout.fill('0');)
  D(cout << "fill char: " << cout.fill() << endl;)
  D(cout.width(10);)
  cout << i << endl;
  D(cout.setf(ios::right, ios::adjustfield);)
  D(cout.width(10);)
  cout << i << endl;
  D(cout.setf(ios::internal, ios::adjustfield);)
  D(cout.width(10);)
  cout << i << endl;
  D(cout << i << endl;) // Without width(10)
```

Print format

```
  D(cout.unsetf(ios::showpos);)
  D(cout.setf(ios::showpoint);)
  D(cout << "prec = " << cout.precision() << endl;)
  D(cout.setf(ios::scientific, ios::floatfield);)
  D(cout << f << endl;)
  D(cout.unsetf(ios::uppercase);)
  D(cout << f << endl;)
  D(cout.setf(ios::fixed, ios::floatfield);)
  D(cout << f << endl;)
  D(cout.precision(20);)
  D(cout << "prec = " << cout.precision() << endl;)
  D(cout << f << endl;)
  D(cout.setf(ios::scientific, ios::floatfield);)
  D(cout << f << endl;)
  D(cout.setf(ios::fixed, ios::floatfield);)
  D(cout << f << endl;)

  D(cout.width(10);)  cout << s << endl;
  D(cout.width(40);)  cout << s << endl;
  D(cout.setf(ios::left, ios::adjustfield);)
  D(cout.width(40);)  cout << s << endl;
}
```

Input from keyboard

Input stream: `std::cin`

Usage of cin

- Operator >>
- Adapted to fundamental data-types
- Adaptable to user classes (overload)
- Return the stream! (composition)

First example

```
#include <iostream>
using namespace std ;
int main() {
  int n;
  float x;
  char t[81];
  do {
    cout << "enter␣an␣integer,␣a␣string␣and␣a␣float:␣" ;
    cin >> n >> t >> x; // composition: read n, return cin, etc.
    cout << "thanks␣for␣" << n << ",␣" << t << "␣and␣" << x << "\n";
  } while (n) ;
  return 0;
}
```

Input problems

Problems and remarks

- Characters reading jump spaces (broad sense)
- De-synchronization, blocking, etc.: use class `istringstream`

```
#include <iostream>
#include <sstream>
using namespace std ;
main() {
  int n;
  bool ok = false ;
  char c;
  string line ;
  do {
    cout << "enter an integer and a character:\n" ;
    getline (cin, line) ; // to read a ligne on keyboard
    istringstream buffer (line) ;
    ok = (buffer>>n>>c); // cin to bool is ok!
  } while (! ok) ;
  cout << "thanks for " << n << " and " << c <<"\n";
}
```

Hello World 1/2

```
// main.c
#include <iostream> // never .h in C++

int main ()
{
  std::cout << "Hello World!" << std::endl;

  return 0;
}
```

Compilation

Like in C, but using g++ :

```
aveneau@localhost> g++ -W -Wall -g -O -c main.c
aveneau@localhost> g++ -W -Wall -g -O -o hello main.o
```

Hello World 2/2

```cpp
/// Hello.h : interface
#ifndef HELLO_H_
#define HELLO_H_
#include <string>
class Hello {
    // class constant, private
    static const std::string m_hello;
public:
    Hello(); // constructor
    ~Hello(); // destructor
    void sayHello (); // method
};
#endif /// HELLO_H_
```

```cpp
/// main.c
#include <iostream> // never .h in C++
#include "Hello.h"

int main ()
{
  Hello hello; // Constructor call

  hello.sayHello (); // Method call

  return 0; // Destructor call
}
```

```cpp
/// Hello.cpp: implantation
#include "Hello.h"
#include <iostream>
/// Definition of the class constant, always in implantation
const std::string Hello::m_hello ( "Hello World!" );
/// Constructor
Hello::Hello() {}
/// Destructor
Hello::~Hello() {}
/// Instance method
void Hello::sayHello () {
    std::cout << Hello::m_hello << std::endl;
}
```