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 2 – The C++ specifics

- Free place for declarations
- References
- Default arguments
- Function overloading
- Heap handling
- The `inline` specification
- Namespace
- New cast operators

Special features overview

- New comments (idem gnu enhancement)
- Free location for declarations
- Notion of “reference”
- Default arguments in function declarations
- Over-definition of functions
- Operators `new` and `delete`
- “inline” functions
- New cast operators
- New boolean data-type `bool` with `true` and `false` special keywords
- Notion of namespace

Free location for declarations

General rules

- More flexible than C: where you want (but still before the first usage)
- Range: still limited to surrounding block
- Expression for scalar variable initialization: anyone

Structured instructions case

- May be declared at the last possible location:

```
for (int i=0 ; ... ; ... ) { // C++ correct, invalid using C language!
   ...
} // "i" is no more known ...
i = 6; // error, except if "i" was previously declared
```

Notion of reference

It is difficult to transmit arguments by address in C: it needs pointers (so, it consists to transmit by value an address)

Classical solution using pointers

```
#include <iostream>
using namespace std;
void swap (int* a, int* b) {
  cout<<"swap("<<*a<<","<<*b<<")"<<endl;
  const int c = *a;  *a = *b;  *b = c;
  cout<<"gives: "<<*a<<","<<*b<<endl;
}
```

```
int main () {
  int n=10, p=20;
  cout<<"before: "<<n<<" and "<<p<<endl;
  swap (&n, &p);
  cout<<"after: "<<n<<" and "<<p<<endl;
  return 0;
}
```

Solution using reference

```
#include <iostream>
using namespace std;
void swap (int &a, int &b) {
  cout<<"swap("<<a<<","<<b<<")"<<endl;
  const int c = a;  a = b;  b = c;
  cout<<"produces: "<<a<<","<<b<<endl;
}
```

```
int main () {
  int n=10, p=20;
  cout<<"before: "<<n<<" and "<<p<<endl;
  swap (n, p);
  cout<<"after: "<<n<<" and "<<p<<endl;
  return 0;
}
```

Remarks

- Process entirely managed by compiler
- How to use reference to pointer? `int* &param`

Reference properties

- Induces indirect threats
  - Unwanted edge effect
- Cast lacks
  - Casting possibilities disappear:
    ```
    void f (int &n);
    float x;
    f(x); // illegal call, no implicit cast
    ```
    Require an argument to be a `lvalue` of the demanded data-type
- Case of a constant argument
  ```
  void fct (int &);
  fct (3); // incorrect: f cannot modify a constant
  const int c = 15;
  fct (c); // incorrect: same reason
  ```
- Case of silent constant argument
  ```
  void fct (const int &);
  fct (3); // correct
  const int c = 15;
  fct (c); // correct
  float x = 1.25;
  fct (x); // correct: a temporarily int variable is built
  ```
  Interesting mainly to send objects as argument!

Return value case

- Return non-local variable

```
int& f() {
  ...
  return n;
}
```

```
...
int p;
...
p = f(); // interest?
```

- Produce a lvalue: useful for operator overloading (eg. [])

```
int & f();
int n;
float x;
...
```

```
...
f() = 2*n + 5; // put 2n+5 to reference
               // given by f()
f() = x;       // idem after cast to int
```

- Possible cast (contrarily to arguments)
  - Warning: false for constant
- Return value and const

```
int n=3;
float x=3.5;
int & f() {
  return 5; // illegal
  return n; // OK
  return x; // illegal
}
```

```
...
const int& f () {
  return 5; // OK, with temp copy
  return n; // OK
  return x; // OK, with cast
            // and then temp copy
}
```

Generalities about reference

- Reference is more general than classical argument
  - Identifier déclaration: `int n; int&p=n;`
  - Here, n and p share same memory space
  - Take care: no pointer on reference arrays
- Reference initialization
  - Mandatory with declaration
  - Reference is not modifiable (constant)
  - Points always to a variable (not `int& n=3;` !)
  - ... except for constant reference! e.g. `const int&n = 3;`
  - Same principle with cast:

```
float x = 3.5f;
const int &n = x; // create a temporary variable (int tmp=3)
cout << "n = " << n << endl; // print 3 (rounded)
n = 6; // compilation error!
cout << "x = " << x << endl; // x not modified, still 3.5 ;-)
```

First example

C's rule: function call with as many arguments as in its signature

C++: values may be automatic ...

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

void fct (int, int=12); /// function declaration with optional arg.

int main () {
  int n=10, p=20;
  fct (n, p); // classical call
  fct (n);    // correct in C++, second argument
              //                 take default value: 12
  return 0;
}

void fct (int a, int b) { /// not necessary to repeat default values
  cout << "call to fct (" << a << "," << b << ");" << endl;
}

/// will print:
// call to fct (10,20);
// call to fct (10,12);
```

NB: call to fct() without parameter will not compile

Second example

```
#include <iostream>
using namespace std;
/// function declaration with two arguments having default values
void fct ( int a=0, int b=12 ) {
  cout << "call to fct (" << a << "," << b << ")" << endl;
}
/// Usage example
int main ()
{
  const int n = 10;
  const int p = 20;
  fct ( n, p );
  fct ( n );
  fct ();
  return 0;
}

/// will print:
// call to fct (10,20)
// call to fct (10,12)
// call to fct (0,12)
```

Default argument's properties

- Always at end of arguments list

```
float fexple ( int = 5, long, int = 3 ); // compilation error
```

- Can use different values, in different declarations

Implementation

- It exists in C for classical operators, like in `a + b`
- In C++, it works also for classes, *eg.* Complex, Vector ...
- More generally, possible for any function
- Discrimination: arguments' type

Implementation in C++

```
#include <iostream>
using namespace std;
/// 2 prototypes with same name
/// Only signatures differ!
void doppelganger (int);
void doppelganger (double);
/// usage example
int main () {
  int n=5;
  double x=2.5;
  doppelganger (n); // call with INT
  doppelganger (x); // call with DOUBLE
  return 0;
}
```

```
/// The first version
void doppelganger (int a)
{
  cout << __FUNCTION__ <<
    " (int a=" << a << ")" <<endl;
}

/// The second version
void doppelganger (double a)
{
  cout << __FUNCTION__ <<
    " (double a=" << a << ")" <<endl;
}
```

Here, no cast is necessary

How the overloaded function is chosen

Rule to determine the called overloaded function

First example

```
void doppelganger (int);    /// I
void doppelganger (double); /// II
char c; float y; long l;
doppelganger (c);   /// call to I, after cast from char to int
doppelganger (y);   /// call to II, after cast from float to double
doppelganger (l);   //  error, no solution for long (ambiguous)
```

Second example

```
void display (char*); /// I
void display (void*); /// II
char *ad1 = ... ;
double *ad2 = ...;
display (ad1); // call I
display (ad2); // call II, after cast to void*
```

Third example

```
void try (int, double); /// I
void try (double, int); /// II
int n, p; double z; char c;
try (n, z); // call I
try (c, z); // call I, after cast to int
try (n, p); // compilation error
```

Overloaded function choice

Fourth example

```
void test (int=0, double=0); /// I
void test (double=0, int=0); /// II
int n; double z;
test (n, z); // I
test (z, n); // II
test (n);    // I
test (z);    // II
test ();     // compilation error, since two solutions
```

Fifth example

```
void thing (int);        // compilation error, cannot distinguish between
void thing (const int); // int and const int (passing by value)
```

Sixth example

```
void thingy (int *);        /// I
void thingy (const int *); /// II
int n=3; const int p=5;
thingy (&n); // I
thingy (&p); // II
```

Seventh example

```
void thingummy (int &);        /// I
void thingummy (const int &); /// II
int n=3; const int p=5;
thingummy (n); // I
thingummy (p); // II
```

Eighth example

```
void thingumbob (int &);        ///  I
void thingumbob (const int &); /// II
int n; float x;
```

```
thingumbob (n); // I
thingumbob (2); // II
thingumbob (x); // II
```

Rule for searching the overloaded function

One argument functions

Search *best* function, using following ordered criteria:

- Exact with both sign attributes and `const` for references or pointers
- Numerical promotion, essentially `char` and `short` → `int` and `float` → `double`
- Standard cast: legal one for assignment, potentially degrading, plus UDC (User Defined Cast)

Search stop at first concluding level. If several solutions inside a same level then compilation error

Many arguments functions

- For each argument, select appropriate functions
- Then do an intersection
  - If several functions at same level: compilation error
  - Idem if no function is found

Link edition and symbol renaming

Overloading works with prototype and modular compiling

Link editor: function choice?

- Renaming (modification) of "external" names of functions
- Use of argument types and internal name

Problem: applies to any function

C function inclusion

Impossible: C++ modifies its name, but not C!

- Use `extern "C"` before its declaration
- Usable in block:

```
#ifdef __cplusplus
extern "C" {
#endif
   void exple (int);
   double thing (int, char, float);
#ifdef __cplusplus
}
#endif
```

The `new` operator

C standard functions are not usable with objects, and are discouraged for others arguments ...

Use examples

```
int *ad1;
ad1 = new int; // allocation similar to: (int*) malloc(sizeof(int));
int *ad2 = new int; // other allocation, during initialization
char *adc;
adc = new char[100]; /// allocate 100 bytes array
```

Syntax and role of `new`

- Two syntax:
  - `new type` returns a type-pointer `type*`
  - `new type[n]`, where `n` is any positive integer expression
- In failure case raise a `bad_alloc` exception

Warning: do not mix up with Java's `new`, that does not work with fundamental types

The `delete` operator

Always free what have been previously allocated using `new` (asap)

Syntax and role

- Classical syntax:
  - `delete address` where address was first allocated using `new`
- Undefined behaviors:
  - multiple freeing
  - bad address

Examples

```
double *adp = new double;
*adp = 1.;
float *adf = new float[10]
for (int i=0; i<10; i++)
  adf[i] = i*(float)*adp;
...
delete adp; /// free  address previously allocated and returned
delete adf; /// same syntax for arrays
```

The new(nothrow) operator

before normalization, in error case new returned NULL

⟹ we can force this old behavior

```
#include <cstdlib> /// exit() declaration
#include <iostream>
using std::cout; /// without including all std, we can declare
using std::endl; /// which parts are visible without using "std::"
int main () {
  long size;
  cout << "wished size: ";
  cin >> size;
  for (int nbloc = 1; ; nbloc++) {
    int *adr = new(std::nothrow) int[size]; /// or "new(nothrow)" if "using namespace std"
    if ( adr == NULL ) {
      cout << "**** not enough memory ****" << endl;
      exit (-1);
    }
    cout << "Allocating block number: " << nbloc << endl;
  }
  return 0;
}
```

Program result

```
wished size: 100000000000
Allocating block number: 1
Allocating block number: 2
...
Allocating block number: 351
nothrow(24975) malloc: *** mmap(size=400000000000) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
**** not enough memory ****
```

Managing memory overflow

Register our callback function instead of classical exception `bad_alloc`

```
#include <cstdlib> /// exit() declaration
#include <new> /// for set_new_handler()
#include <iostream>
using std::cout;
using std::endl;
void overflow () { // called when not enough memory
  cout << "Not enough memory" << endl;
  cout << "Program exit" << endl;
  exit (-1);
}
int main () {
  std::set_new_handler (overflow);
  long size;
  cout << "wished size: ";
  std::cin >> size;
  for (int nbloc=1; ; nbloc++) {
    int*adr = new int[size];
    cout << "allocation block number " << nbloc << endl;
  }
  return 0;
}
```

Program result

```
wished size: 100000000000
allocation block number 1
...
allocation block number 351
new_handler(25301) malloc: *** mmap(size=400000000000) failed (error code=12)
*** error: can't allocate region
*** set a breakpoint in malloc_error_break to debug
Not enough memory
Program exit
```

Macros *vs* inline

Macro-function: pseudo function dealt by preprocessor, based on some rewriting

```
#define SQUARE(_x) (_x)*(_x)
int main () {
  cout << "square(2+1)="
       << SQUARE(2+1) << endl;
  return 0;
}
```

⟹

```
int main {
  cout << "square(2+1)="
       << (2+1)*(2+1) << endl;
  return 0;
}
```

Idea is to optimize the code: avoid instructions (save current state like registers etc.) relying to function call ...

- Function needs less memory but are slower

Inversely: macro-functions generally fail with side-effects

- In C++ we do no more use macros!
- ... but `inline` functions

Square example

```
inline double square (double x) { return x * x ; }
```

The "inline" functions

Another example:

```
#include <cmath> /// sqrt() definition
#include <iostream>
using namespace std;

inline double norme (double vec[3]) {
    double s=0.;
    for (int i=0; i<3; i++)
        s += vec[i]*vec[i];
    return sqrt(s);
}
```

```
int main() {
    double v1[3], v2[3];
    for (int i=0; i<3; i++) {
        v1[i] = double(i);
        v2[i] = double(2*i-1);
    }
    cout<<"norme1: "<<norme(v1)<<endl;
    cout<<"norme2: "<<norme(v2)<<endl;
    return 0;
}
```

Comparison

	Pros	Cons
Macro	- Save time	- Memory space loss - Side-effect risk - No separate compilation
Function	- Save memory - Separate compilation	- Execution time loss
inline function	- Save time	- Memory space loss - No separate compilation

Using namespace

```cpp
#include <iostream>
int      x = 1;
double d = 1.;
using namespace std;
namespace A /// definition
{ /// of named space A
  int x = 2;
  double d = 2.;
  void f(int a) {
    cout<<"A::f("<<a<<")"<<endl;
  }
}
using namespace A; // !!
namespace B /// definition
{ /// of named space B
  int x = 3;
```

```cpp
  double d = 3.;
  void f(double a) {
    cout<<"B::f("<<a<<")"<<endl;
  }
}
int main ()
{
  f(A::x);      // f from A, x from A
  f(B::x);      // f from A, x from B
  B::f(A::d);  // f from B, d from A
  B::f(B::d);  // f from B, d from B
  f(::x);       // f from A, x global
  B::f(::d);   // f from B, d global
  return 0;
}
```

Remarks

- Instruction `using` come after definition, for instance through an inclusion
- Always at global level
- Incremental creation is possible (like `std`)

Usage

- By stipulating namespace, eg. `A::x` or `::x`

```cpp
namespace A {  int n; }
long n; /// synonymous of n
int main () {
  using A::n; // now n is synonymous of A::n !!
  ...
}
void f () {
  ... // here n is synonymous of ::n !!
}
```

- By simplifying writing with `using`, eg. `using A::x;`
- By using `using` direction for namespace

```cpp
namespace A { int n; double x; }
float x;
int main () {
  using namespace A; /// may be local to a given block!
  x = 3.;     // idem A::x
  A::x = 3.; // always usable
  n = 1;      // idem A::n, of course
  ...
}
```

- Always possible to come back to hidden writing
- Ambiguity leads to compilation errors

Others properties

- Inclusion: put a namespace inside another one, etc.

```
namespace A {
  int n;
  namespace B { int n; }
}
```

```
void f() {
  using namespace A;
  n = 3;
  B::n = 4;
  A::B::n = 4;
}
```

```
void g() {
  using namespace A;
  n = 3;
  A::n = 3;
  A::B::n = 4;
}
```

- Transitivity

```
namespace A {
  int n;
  float x;
}
```

```
namespace B {
  using namespace A;
  float y;
}
```

```
void g() {
  using namespace B;
  n = 3;   // idem A::n
  x = 4.;  // idem A::x
  y = 5.;  // idem B::y
}
```

- Aliases

```
namespace TooLongNameSpace {
  int n;  float x;          }
```

```
namespace E TooLongNameSpace;
```

- Anonymous namespace to replace `static`

```
namespace {
  int n;  float x;
}
```

```
void f() {
  ::n = 3;
  ::x = 3.14159f;
}
```

Replace old C versions:

- `const_cast`: between given constant type and the same type with or without `const` and `volatile`
- `reinterpret_cast`: for cast which result depends on implementation (eg. int to pointer)
- `static_cast`: for implementation independent casts (wide meaning)

```
int n = 12;
const int *ad1 = &n;
int *ad2 = const_cast<int*>(ad1);        // idem (int *)
ad1      = const_cast<const int*>(ad2); // idem (const int*)
int m = 12;
const int*const ad3 = &m;     // here constant pointer (like ad1) to constant data
int *ad4 = const_cast<int*>(ad3);        // idem (int*)

long l;
int *adi = reinterpret_cast<int*>(l);    // idem (int*)
l = reinterpret_cast<long>(adi);         // idem (long)

int p = static_cast<int> (l);            // idem (int)
```