INTRODUCTION TO C++

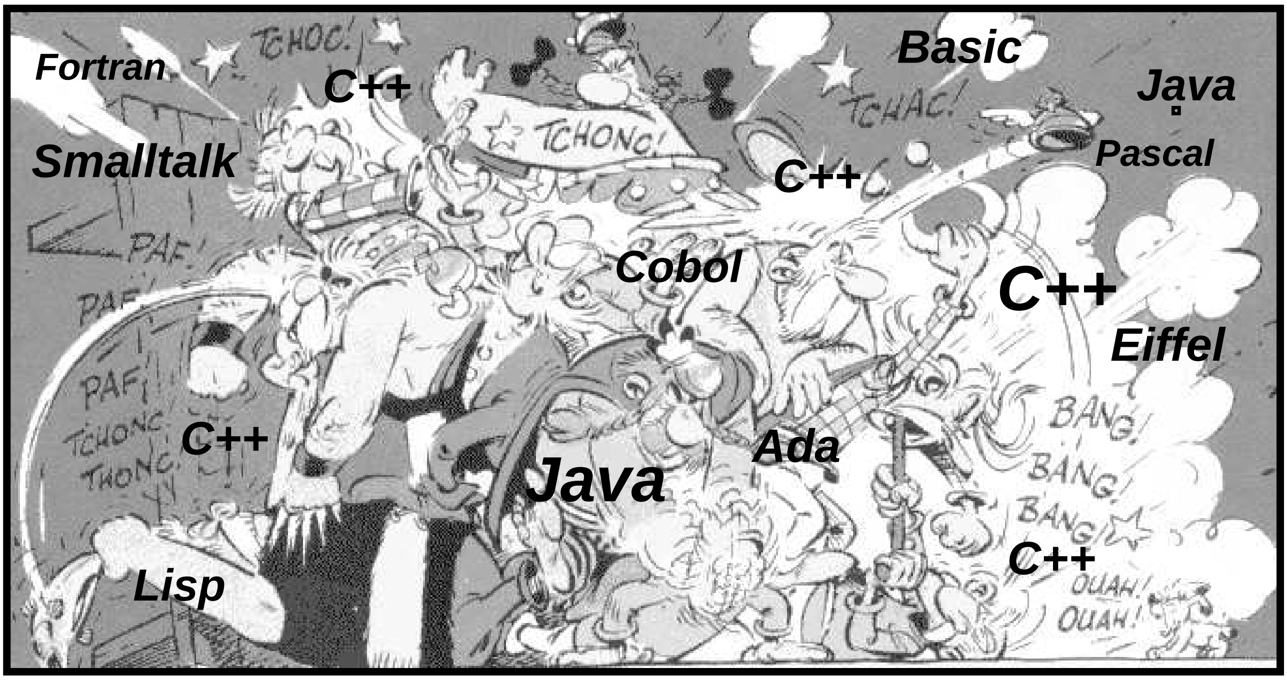
Gérald MASINI
Senior Researcher

L O R I A
ISA Research Group (`http://www.loria.fr/isa`)

Post: B.P. 239, 54506 Vandœuvre-lès-Nancy Cedex, France
E-mail: Gerald.Masini@loria.fr
Tel: (+33) (0)3.83.59.20.68
Fax: (+33) (0)3.83.41.30.79

Programmers often deny C++...
but use it!

Programmers often praise other OOL...
but do not use them!



• Ye Jolly Programmers •

Bits of History: C

- C was designed around 1970 in AT&T Bell Labs
  by Brian W. Kernighan and Dennis M. Ritchie

- For the UNIX OS:
  the kernel was written in C

- C is a low-level language aimed at efficiency:
  - elementary objects (characters, addresses, ...)
  - elementary operations (assignment, comparisons, ...)

- C quickly met with success and became available with all usual OS

- Many dialects:
  ANSI C was defined around 1988

C will be supposed to be your native language!

Bits of History: C++

- C++ was also designed in AT&T Bell Labs
  by Bjarne Stroustrup

- C++ is an extension of C

```
main {
  Language C;
  C++;
}
```

- The first '+' : C with classes (1979)

  C with classes = C + { *classes*, *inheritance*, *friends*, *constructors* and *destructors*, *inline functions*,... }

  But no dynamic binding

- The second '+' (1983–1997)

  C++ = C with classes + { *references*, *overloading*, *dynamic binding*, *multiple inheritance*, *templates*,... }

- *ISO/IEC Final Draft International Standard 14882*
  Standard defined by ANSI X3J16 Committee and ISO WG-21 Working Group (November 14, 1997)

Bjarne Stroustrup's purposes:

- provide object-oriented facilities to C programmers
  inspired by Simula
- preserve C programmers' habits
- make C to C++ transition easy

Why C?

- C is universal
  suited to any application field any programming method (!)
- C is available
  on supercomputers as well as personnal computers
- C is efficient
  low-level language
- C programs are (supposed to be) portable

Books

A lot of books about C++ are available

Four of them are especially recommended:

- Bjarne Stroustrup
  The C++ Programming Language, 3rd Edition
  Addison-Wesley, Reading (MA), USA, 1997

  The C++ Reference Manual

- Bjarne Stroustrup
  The Design and Evolution of C++
  Addison-Wesley, Reading (MA), USA, 1994

  The father of C++ explains his motivations

- Stanley B. Lippman and Joelle Lajoie
  C++ Primer, 3rd Edition
  Addison-Wesley, Reading (MA), USA, 1998

  To learn C++ programming
  very complete (1200 pages!)

- David R. Musser and Atul Saini
  STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library
  Addison-Wesley, Reading (MA), USA, 1996

  To learn using the Standard Template Library

PART 1: CLASSES

Declaration of class Article

looks like a C structure (`struct`)

```
// File Article.h
class Article {
private:  // DATA MEMBERS
  char* _name;
      // Article name
  float _basePrice;
      // Price not including taxes
  int _quantity;
      // Number of items currently in stock
public:  // FUNCTION MEMBERS
  char* name();
      // Get the article name
  float basePrice();
      // Get the price not including taxes
  int quantity();
      // Get the number of items in stock
  float price();
      // Get the price including taxes
  void remove(int n);
      // Withdraw 'n' items from the stock
  void add(int n);
      // Add 'n' items to the stock
};
```

Definition of the function members

```
#include "Article.h"
  // Insert the Article class definition

char* Article::name()
  { return _name; }

float Article::basePrice()
  { return _basePrice }

int Article::quantite()
  { return _quantity; }

float Article::price()
  { return(1.2 * _basePrice); }

void Article::remove(int n)
  { _quantity -= n; }

void Article::add(int n)
  { _quantity += n; }
```

- Member functions are C functions
- The **scope resolution operator** ‘**::**’ is used to attach each function to its class

Friends (either classes or functions)

```
// File Article.h: Declaration of class Article
class Article {

private:
  char* _name;
  float _basePrice;

public:
  friend class LiquorStore;
  friend void Drugstore::sell();
  friend void debug();

  int quantity; // Public data member

  char* name();
  float basePrice();
  float price();
  void remove(int n);
  void add(int n);

};
```

- *Any client* of class Article may get or set the value of *public* data members
- *Friends* may use the private members of *any object* of type Article

Inline functions are macros

```
// File Article.h: Declaration of class Article
class Article {
private:
  char* _name;
  float _basePrice;
  int _quantity;

public:
  char* name() { return _name; }
  float basePrice() { return _basePrice; }
  int quantity() { return _quantity; }
  float price() { return(_basePrice * 1.2); }
  void remove(int n) { _quantity -= n; }
  void add(int n);
};

inline void Article::add(int n) { _quantity += n; }
```

- Efficiency is preserved:

  The compiler expands a function call into the corresponding code (i.e. the body of the inline function)

  C macros (`#define`) only perform text substitution

- Information hiding is preserved:

  Programmers still write function calls to perform operations on objects (here, to access data members)

SUMMARY

- A class defines a type
- A class is not a syntactical unit:
  Declarations of different classes and definitions of functions may be scattered and mixed in several files
- The protection unit is the class:
  A function member of a given class can use the private members of *any object* of that class
- The value of public data members may be changed by any client (e.g. using an assignment)

PART 2: REFERENCES & CONSTANTS

```
#include "Article.h"
main() {
  Article a, b;
  Article& refa = a;  // A reference
  int i, j;
  int* pi,* pj;    // Pointers
  int& refi = i;  // References must be initialized
  int& refj = j;  // at declaration
  ...
  pi = &i; // 'pi' contains the address of 'i'
  pj = pi; // 'pj' contains the same address as 'pi'
  *pj = 10; // 10 is assigned to 'i'
  ...
  refa = b;       // Same effect as a = b
  refi = refj;    // Same effect as i = j
  ...
};
```

- C types (`char`, `int`, `float`, etc.) are available
- A *pointer* is an address
- A *reference* is a new name (an *alias*) for an existing object

  A reference is also represented by a pointer but is used like a variable: Operations performed on the reference are performed on the object

Call by reference

```
void swap1 (int* i, int* j)
/* C style: Exchange values of ‘i’ and ‘j’ */
  {
    int tmp = *i;
    *i = *j;
    *j = tmp;
  }

void swap2 (int& i, int& j)
// C++ style: Exchange values of ‘i’ and ‘j’
  {
    int tmp = i;
    i = j;
    j = tmp;
  }

main() {
  int m = 1, n = 2;
  swap1(&m, &n);  // Now m == 2 and n == 1
  swap2(m, n);    // Now m == 1 and n == 2
}
```

- Call by reference provides write access to the effective parameters without using pointers
- Using references increases understandability of source code, but is also dangerous (it provides write access to parameters in the function body)

Constant variables

```
int in(const int a[], const int s, const int el)
// Does array 'a' contain element 'el'?
// 's' is the size of 'a'
  {
    int i;
    for(i = 0 ; (i < s) && (a[i] != el) ; i++);
    return (i != s);
  }

main() {
  const int size = 1000;
  int table[size];
  int i, inTable;
  ...
  inTable = in(table, size, i);
  ...
}
```

- The value of a constant variable cannot be modified
- Formal parameters declared as constant references:

  `const Type& parameter`

  are useful to pass the address of a large structure whilst forbidding write access within the function

Constant functions

```
class Article {
private:
  char* _name;
  float _basePrice;
  int _quantity;

public:
  char* name() const { return _name; }
  float basePrice() const;
  int quantity() const { return _quantity; }
  float price() const {return (_basePrice * 1.2);}
  void remove(int n) { _quantity -= n; }
  void add(int n) { _quantity += n; }
};

inline float Article::basePrice() const
  { return _basePrice; }
```

- Constant functions increase security:
  They have not write access to data members...

- ...because, within a constant function, the current object is declared as:

  ```
  const Article* const this;
  ```

  (constant pointer to constant data)

⇒ use constant functions to implement access functions

SUMMARY

- A variable implicitely contains an object:
  `x = y` is an assignment of **objects**

- Pointers must be explicitely declared and manipulated, using operator '$*$'

- A reference must be initialized at declaration
  The corresponding address cannot be changed afterwards

PART 3: OBJECTS

Memory allocation

```
class Complex {
private:
  double _re, _im;  // Real and imaginary parts
public:
  double re() const { return _re; }
  double im() const { return _im; }
  double module() const
    { return sqrt((_re * _re) + (_im * _im)); }
  ...
};
```

```
#include "Complex.h"
main() {
  Complex* z = new Complex;
  Complex* tz = new Complex[10000];
  ...
  delete z;     // Delete the object pointed by 'z'
  delete[] tz;  // Delete entire array 'tz'
  ...
}
```

- Operator `new` dynamically allocates memory
  It returns the *address* of the allocated space (this address may then be assigned to a pointer or a reference)
- Operator `delete` dynamically frees memory space

Constructors

- Constructors are function members used to **initialize** objects, but not to **create** them

  They should rather be called *initializers*

- A constructor is named like its class

  It specifies no result

```
class Article {
public:
  Article(int quantity);  // Constructor
  Article();              // Default constructor
  ...
};
```

- A so-called *default-constructor* has no parameter

  When a class has no constructor, a default constructor is automatically generated

- When a class has a constructor that is not the default, it must be used to:
  - Initialize a variable of the corresponding type
  - Initialize an object dynamically created using operator `new`

```
class Complex {
private:
  double _re;
  double _im;
public:
  // No explicitely defined constructor
  ...
};
```

```
class Article {
private:
  char* _name;
  float _basePrice;
  int _quantity;
public:
  // Explicitely defined constructor
  Article(char *n, float p, int q)
    {
      _name = n;
      _basePrice = p;
      _quantity = q;
    }
  ...
};
```

```
#include "Article.h"
#include "Complex.h"
main() {
  Article turpentine("Oil of turpentine", 39.95, 20);
  Article* soap, peas;
  Complex z;  // Initialized by the default-constructor
  ...
  soap = new Article("Dove soap-ball", 4.95, 100);
  peas = new Article;  // Error:  no initialization
  ...
}
```

The initialization list

- A constructor works in two times:
  - **Implicit initialization** of the data members using the default constructors of the data members
  - **Execution** of the code of the body
- The initialization list explicitely specifies calls to existing contructors

```
class Article {
private:
  char* _name;
  float _basePrice;
  int _quantity;
public:
  Article(char *n, float p, int q)  // Constructor
    {
      _name = n;
      _basePrice = p;
      _quantity = q;
    }
  ...
};

class ArticleLot {
private:
  Article _art;  // The article of the pack
  int _number;   // Number of items in the pack
public:
  ArticleLot(char* n, float p, int q, int nb)
    : _art(n, p, q) { _number = nb; }
  ...
};
```

Fundamental objects may also be initialized using pre-defined constructors

```
class Article {
private:
  char* _name;
  float _basePrice;
  int _quantity;
public:
  Article(char *n, float p, int q)
      : _name(n), _basePrice(p), _quantity(q)
    {
      // Empty body
    }
  ...
};
```

```
#include "Article.h"
class ArticleLot {
private:
  Article _art;  // The article of the lot
  int _number;   // Number of items in the lot
public:
  ArticleLot(char* n, float p, int q, int nb)
        : _art(n, p, q), _number(nb)
    {
      /* Empty body */
    }
  ...
};
```

```
#include "ArticleLot.h"
main() {
  ArticleLot a = new("Cocoa-nut ice cream", 22.50, 100, 2);
  int i(0);  // Same as 'i = 0;'
  ...
}
```

Destructors

- Destructors are function members used to free memory space which has been dynamically allocated for data members

- A destructor is named like its class and is introduced by a '~' character

  It has neither parameter nor result

```
class Article {
public:
  ~Article() { ... }    // Destructor
  ...
};
```

- Destructors are **implicitely** invoked **before** object deallocation

  Deallocation occurs:

  – at the end of a block: { ... }

  – by an explicit deallocation, using the `delete` operator

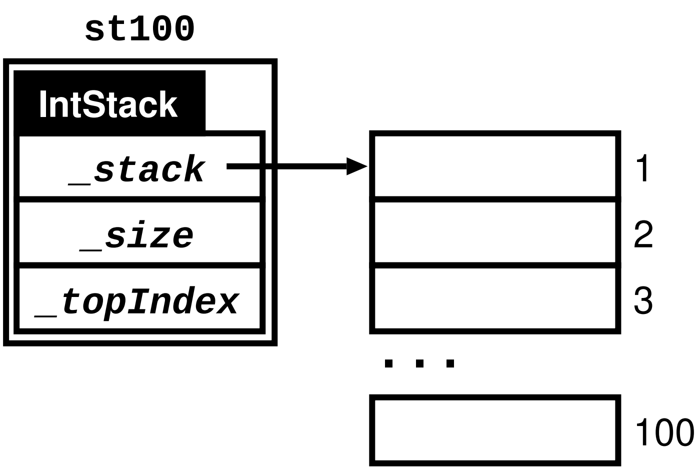
- **Destructors cannot be explicitely invoked**

```
// Standard library for input/output
#include <iostream.h>

class IntStack {  // Stack of integers
private:
  int* _stack;    // The stack is implemented by an array
  int _size;      // Size of array 'stack'
  int _topIndex;  // Index of the top of the stack
public:
  // The constructor
  IntStack(int s = 100)
    : _stack(new int[s]), _size(s), _topIndex(-1) { }
  // The destructor
  ~IntStack()
    {
      if (!empty())
        cerr << "WARNING: Stack is not empty!" << endl;
      delete[] _stack;  // Free memory space
    }                   // allocated to array 'stack'
  int top() const;
  int empty() const { return(_topIndex < 0;); }
  void push(int i);
  void pop();
  ...
};

int IntStack::top() const
  {
    if(empty()) {
      cerr << "WARNING: Empty stack!" << endl;
      abort();
    }
    return stack[_topIndex];
  }
...
```

```
#include "Stack.h"
...
  { // BEGINNING OF BLOCK
    ...
    IntStack st100;
      // Stack of 100 elements (default size)
    IntStack st500(500);
      // Stack of 500 elements
    IntStack* pst5 = new IntStack(5);
      // Pointer to a stack of 5 elements
    ...
    delete[] pst5;
      // Explicit deallocation of the stack
      // addressed by 'pst5':  The memory space
      // addressed by data member '_stack' is first
      // deallocated by the destructor.
    ...
  } // END OF BLOCK
    // Automatic deallocation of the local objects:
    // Stacks 'st100' and 'st500' (same as above)
...
```



SUMMARY

- Objects are **implicitely created** at declaration...
- ...and are then initialized:
  - initialization **must be** explicit when there is a constructor different from the default constructor
  - otherwise, initialization is implicit, using the default constructor
- The management of the memory space occupied by objects which are no longer accessible is in charge of programmers

  It is a difficult and complicated task as *objects often shares data structures because of pointers*

  $\Longrightarrow$ it is the source of a great number of bugs

Member Selection

```
#include "Article.h"
main() {
  Article a("Norwegian whisky", 454.60, 5);
  Article& ra = a;
  Article* pa = new Article("Rice", 29.95, 150);
  char* n;
  ...
  int q = a.quantity();  // Get quantity in stock
  a.remove(1);           // Withdraw 1 article
  ...
  float bp = ra.basePrice();  // Get base price
  float p = pa->price();      // Get price
  ...
  n = pa->_name;
  ...     // Error: '_name' is a private member
}
```

- Member selection uses the dot notation:

  `target.member_name;`

- When `target` is a pointer, operator '->' must be used:

  `target->member_name;`

- No (Eiffel-like) uniform reference principle
  - data members: `target.data;`
  - function members: `target.function();`

  $\Rightarrow$ *programmers have to worry about implementation*

PART 4: OVERLOADING AND CASTING

Overloading function names

- Classical case (true in any OOL):

  Different classes may define function members with same name and profile, i.e. same number and types of parameters

```
class Article {
public:
  void remove(int q);
  ...
};
```

```
class ArticleLot {
public:
  void remove(int q);
  ...
};
```

- C++ specific case (true for non-member functions too):

  Several functions of a same class may have the same name if the number or the types of their parameters are different

```
class Article {
public:
  float price() { ... }
  float price(float discount) { ... }
  ...
};
```

Overloading is particularly convenient for constructors

```
// File Time.h
class Time {
private:
  int _hths;  // Time, in hundredths of seconds
public:
  // Constructor with 3 parameters
    Time(int min, int sec, int h)
      : _hths((min * 6000) + (sec * 100) + h) { }
  // Constructor with 1 parameter
    Time(int h) : _hths(h) { }
  // Default constructor
    Time() : _hths(0) { }
  ...
};
```

```
#include "Time.h"

int main() {
  // 3 DIFFERENT WAYS TO INITIALIZE A TIME
  // Using the 3-parameter constructor
    Time t1(5, 30, 0);
  // Using the 1-parameter constructor
    Time t2(60);
  // Using the default constructor
    Time t3;
  ...
}
```

Overloading Operators

- An *operator function* corresponds to each operator:

  `operator+()` for '+'

  `operator<()` for '<'

  etc.

- Such a function (except for `'::'`, `'.*'`, `'.'` and `'?:'`) may be overloaded...

  ...but:

  - Arity (number of args) and priority must remain unchanged
  - It is impossible to define new operators

- An operator function (except for `'='`, `'[]'`, `'()'` and `'->'`) may also be defined as a non-member function

  Non-member operator functions applying to basic types (`char`, `int`, etc.) cannot be overloaded

```
// File Time.h
class Time {
private:
  int _hths;  // Time, in hundredths of seconds
public:
  Time(int h = 0) : _hths(h) { }
  int hundredths() const { return _hths; }
  ...
  // Returns the address of an object of class Time
  Time& operator+(const Time& t) const
    { return Time(_hths + t._hths); }
    // An object of a given class, Time, can
    // directly access the private members
    // of any object of the same class, Time

  int operator<(const Time& t) const
    { return(_hths < t._hths); }
  ...
};
```

```
#include "Time.h"
main() {
  Time t1, t2, t3, tmin;
  t3 = t1 + t2;             // Addition of two times
  t3 = t1.operator+(t2);  // Same as previous line
  ...
  if (t1 < t2) tmin = t1;
  ...
}
```

Casting

Three kinds of casting operations:

- implicit casting operations, e.g. *integer* $\rightarrow$ *float*

- A single-parametered constructor defines a casting operation:
  parameter type $\rightarrow$ *type defined by the class of the constructor*

- A class may be provided with casting operators to transform the current object into objects of other types

  Such an operator:
  - is named like the target type
  - has no parameter
  - specifies no result

```
class Article {
private:
  float _basePrice;
  ...
public:
  operator float() { return _basePrice; }
    // Cast an Article into a float!
  ...
};
```

Casting may be explicit, and also implicit (im particular, when the type of a variable or an effective parameter does not correspond to the expected type)

```
class Time {
private:
  int _hths;  // Time, in hundredths of seconds
public:
  // Constructor with 3 parameters
    Time(int min, int sec, int h)
      : _hths((min * 6000) + (sec * 100) + h) { }
  // Constructor with 1 parameter: It also
  // defines how to cast an integer into a Time
    Time(int h) : _hths(h) { }
  // Default constructor
    Time() : _hths(0) { }
  // Cast a Time into an integer
    operator int() const { return _hths; }
  ...
};

int main() {
  Time t;
  int i1, i2, i3;
  t = 100;           // Same as `t = Time(100);'
  ...
  if (t < 600) ...   // Same as `t < Time(600);'
  ...
  i1 = int(t); // Explicit casting
  i2 = (int)t; // Same as previous line (C notation)
  i3 = t;      // Same as `i3 = int(t);'
  ...
}
```

PART 5: INHERITANCE

Simple Inheritance

```
// File Clothing.h
#include "Article.h"
class Clothing : public Article {
private:
  int _size;
  char* _colour;
public:
  ...
  int size() const { return _size; }
  char* colour() const { return _colour; }
  ...
  void discount(float discount)
    { _basePrice -= discount); }
    // Error:
    // '_basePrice' is a private member of Article
  ...
};
```

- `Clothing` derives from `Article`
- `Article` is the base class of `Clothing`
- private members are not in the scope in derived classes

Derivation

A derivation may be qualified as:

- **public**:
  public inherited members remain public

- **private**:
  public inherited members become private

- **protected**:
  public and protected inherited members become protected

$\Rightarrow$ a protected member of a class can be accessed by:

- the member functions and the friends of the class
- the member functions and the friends of the derived classes

```
class Article {
protected:
  char* _name;
  float _basePrice;
  int _quantity;
public:
  Article(char* n, float p, int q)
    : _name(n), _basePrice(p), _quantity(q) { }
  char* name() const { return _name; }
  float basePrice() const { return _basePrice; }
  int quantity() const { return _quantity; }
  float price() const { return(_basePrice * 1.2); }
  void remove(int n) { _quantity -= n; }
  ...
};
```

```
#include "Article.h"
class Clothing : public Article {
protected:
  int _size;
  char* _colour;
public:
  int size() const { return _size; }
  char* colour() const { return _colour; }
  void discount(float discount)
    { _basePrice -= discount); }
    // OK: '_basePrice' is
    // a protected member of Article
  ...
};
```

Initialization of an object of the derived class

- The constructor of the base class (`Article`) is first invoked
- The constructors of the data members are then invoked, in declaration order: `_size`, and then `_colour`
- The body of the constructor of the derived class (`Clothing`) is finally executed
- *Default constructors are used, unless otherwise specified by the initialization list*

```
#include "Article.h"
class Clothing : public Article {
protected:
  int _size;
  char* _colour;
public:
  // CONSTRUCTOR
  Clothing(char* n, float p, int q, int s, char* c)
    // Explicit initializations
    // using the initialization list
    :  Article(n,p,q),      // Base class
       _size(s), _colour(c)  // Data members
    { }  // Empty body
  ...
};
```

Function redefinition (shadowing)

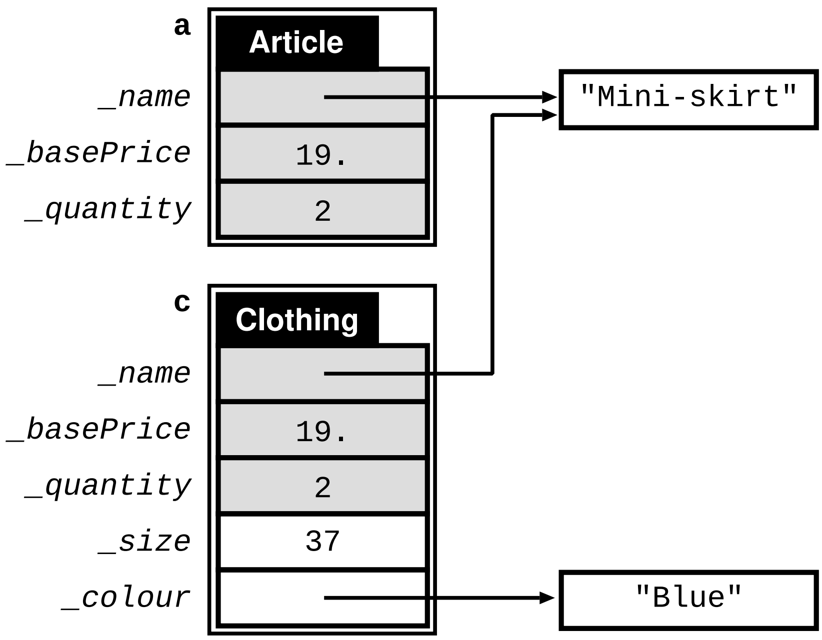
```
class Base {
public:
  int f(int);          // Function #1
  int f(float);        // Function #2
  int f(int, char);    // Function #3
};
class Derived : public Base {
public:
  // REDEFINITION SHADOWS ALL INHERITED DEFINITIONS
  int f(int);          // Function #4
  int f(char);         // Function #5
};

int main() {
  Base b;
  Derived d;
  int i;
  i = b.f(10);      // Call to function #1
  i = b.f(37.2);    // Call to function #2
  i = b.f('$');     // Error: No matching profile
  ...
  i = d.f(100);       // Call to function #4
  i = d.f(6.35);      // Call to function #4 after
                      // implicit conversion of 6.35
  i = d.f(4, '*');    // Error: No matching profile
  i = d.Base::f(4, '*');  // Explicit call
                          // to function #3
}
```

Polymorphism

Polymorphism is only effective for **pointers** and **references**

```
#include "Article.h"
#include "Clothing.h"
void main() {
  Article a("Bubble-gum", 0.35, 500);
  Clothing c("Mini-skirt", 19., 2, 37, "Blue");
  Article* pa;
  Clothing* pc =
    new Clothing("Shirt", 197., 10, 44, "Black");
  ...
  a = c;    // Implicit casting
  pa = pc;  // Address assignment (polymorphism)
  ...
}
```



Be careful with initializations!

```
#include "Article.h"
#include "Clothing.h"
void main() {
  Article a("Bubble-gum", 0.35, 500);
  Clothing c("Mini-skirt", 19., 2, 37, "Blue");
  // Class Article is not provided with a default
  // constructor:  Explicit initialization, using
  // an existing constructor, is required
  Article* pa = ("Fake", 0., 0);
  Clothing* pc =
    new Clothing("Shirt", 197., 10, 44, "Black");
  ...
  a = c;    // Implicit casting
  pa = pc;  // Address assignment (polymorphism)
  ...
}
```

Dynamic binding

Binding is **implicitely static**

```
#include "Article.h"
class Food :public Article {
protected:
  char* _date;  // Consumption date
public:
  float price() const { return(_basePrice * 1.05); }
  ...
};
```

```
#include "Article.h"

main {
  Article* order[1000]; // Array of ordered articles
  int na;               // Nb of ordered articles
  int i;
  float total;
  ...
  // Compute total cost of order
  for(i = 0, total = 0. ; i < na ; i++)
    total += order[i]->price();
    // Static binding: Function ‘Article::price()’
    // is always called because the static type
    // of ‘order’ is ‘Article’
  ...
}
```

Dynamic binding requires:

- **polymorphism**: the target must be a pointer or a reference
- **virtual functions**

```
class Article {
  ...
public:
  virtual float price() const { return(_basePrice * 1.2); }
  ...
};
```

```
#include "Article.h"
class Food : public Article {
protected:
  char* _date;  // Consumption date
public:
  float price() const { return(_basePrice * 1.05); }
    // The redefinition of a virtual function is also virtual
  ...
};
```

```
#include "Article.h"
main {
  Article* order[1000];  // Array for ordered articles
  int na;                // Number of ordered articles
  int i;
  float total;
  ...
  for(i = 0, total = 0. ; i < na ; i++)
    total += order[i]->price();
    // Dynamic binding:
    // Function 'price()' is called according to
    // the dynamic type of 'order[i]'
  ...
}
```

Overloading and binding

When a function `f` is called by a reference or a pointer of type `T`:

- the **best-suited** definition is **statically** chosen among the definitions of function `f` in class `T`
- if the selected function is virtual, dynamic binding is applied

```
class Article {
public:
  ...
  virtual float price() const
    { return(_basePrice * 1.05); }
  float price(float discount) const
    { return((_basePrice * 1.05) - discount); }
  float price(int discountRate) const
    { return(_basePrice * 1.05 * discountRate); }
};
```

```
#include "Article.h"
#include "Clothing.h"
main {
  Article* a;
  Clothing* c;
  ...
  a = c;
  a->price();       // Dynamic binding
  a->price(4.95);   // Static binding
  a->price(12);     // Static binding
}
```

Abstract Classes

- An abstract class declares **pure virtual functions** whose implementation is given in the derived classes
- The declarations of pure virtual functions are required for static type checking
- An abstract class cannot be instantiated

```
class Article {
public:
  virtual float price() const = 0;
    // Pure virtual function
    // to be defined in concrete classes
  ...
};
```

```
#include "Article.h"
class Clothing : public Article {
public:
  float price() const { return(_basePrice * 1.2); }
    // Class Article is not abstract
  ...
};
```

```
#include "Article.h"
class Food : public Article {
public:
  float price() const { return(_basePrice * 1.05); }
    // Class Food is not abstract
  ...
};
```

SUMMARY

- Polymorphism is only effective for pointers and references

  An object of a given type may receive an object of a subtype, **after an implicit conversion**

  A pointer or a reference of a given type may receive a pointer or a reference of a subtype, without any conversion

- **Binding is implicitely static**

- Dynamic binding requires:
  - polymorphism, i.e. a pointer or reference
  - a virtual function

- Because overloading and casting combine with binding, the rules to determine which function is effectively called are complicated

- Abstract classes declare pure virtual functions

Multiple inheritance

- A class may derive from several base classes

- The declaration order of the base classes determines the order of the invocations of the constructors of the base classes

  Destructors are invoked in the reverse order

- In case of repeated inheritance, an object of the derived class contains as many instances of the common ancestor class as inheritance paths

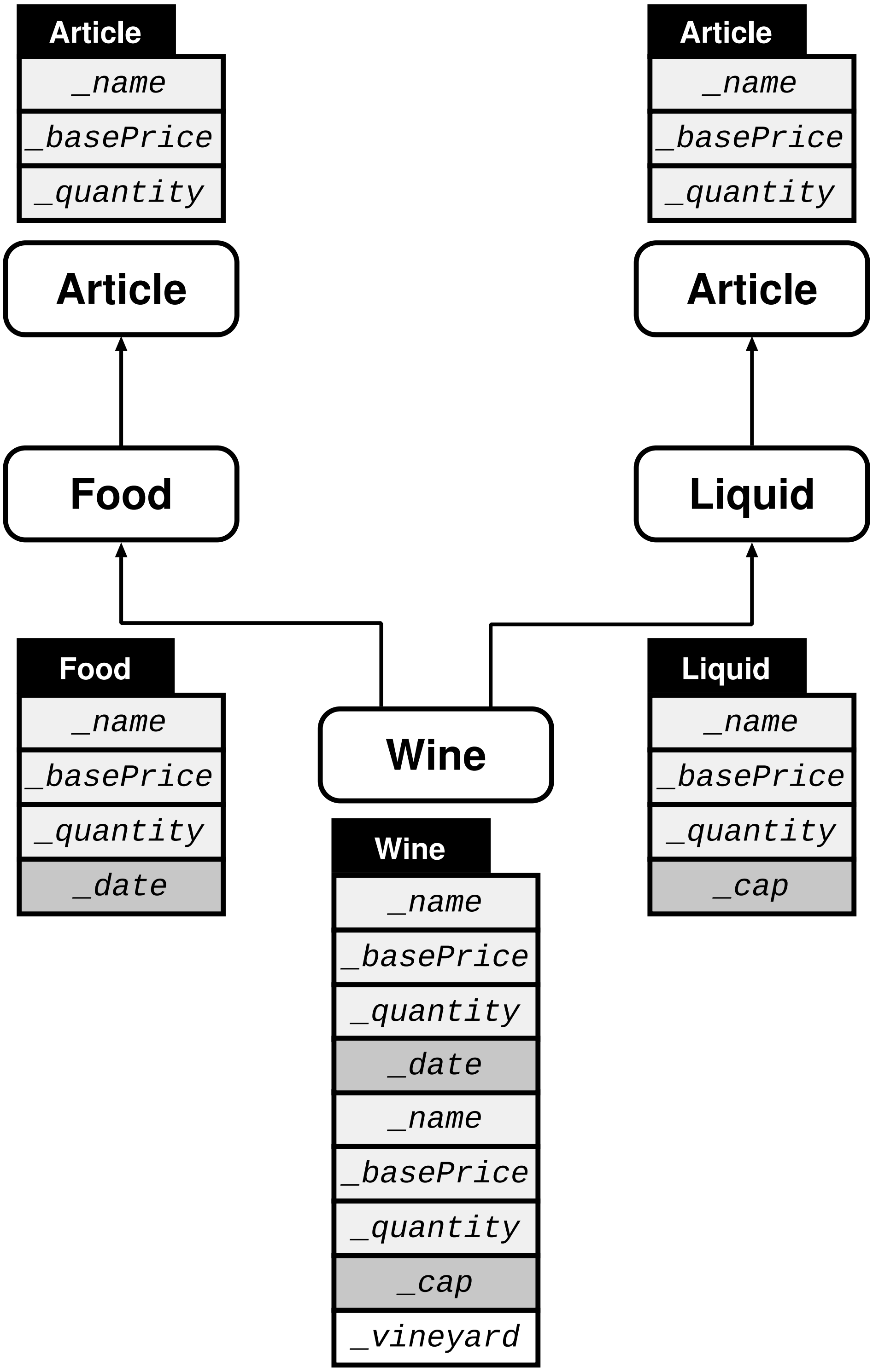
- *Using multiple inheritance is difficult and tedious (especially because of constructors and destructors)*

```
#include "Article.h"
class Food : public Article {
protected:
  char* _date;
public:
  Food(char* n, float p, int q, char* d)
    : Article(n, p, q), _date(d) { }
  ...
};

class Liquid : public Article {
protected:
  float _cap;  // Capacity of a bottle, in litres
public:
  Liquid(char* n, float p, int q, float c)
    : Article(n, p, q), _cap(c) { }
  ...
};

class Wine : public Food, public Liquid {
protected:
 char* _vineyard;
public:
 Wine(char* n,float p,int q,char* d,float c,char* v)
   : Food(n,p,q,d), Liquid(n,p,q,c), _vineyard(v)
   { }
  ...
};
```

Repeated Inheritance is the default



Virtual derivation disables repeated inheritance

```
#include "Article.h"
class Food : virtual public Article {
protected:
  char* _date;
public:
  Food(char* n, float p, int q, char* d)
    : Article(n, p, q) _date(d) { }
  ...
};
class Liquid : public virtual Article {
protected:
  float _cap;  // Capacity of a bottle, in litres
public:
  Liquid(char* n, float p, int q, float c)
    : Article(n, p, q), _cap(c) { }
  ...
};
class Wine : public Food, public Liquid {
protected:
 char* _vineyard;
public:
 Wine(char* n,float p,int q,char* d,float c,char* v)
 // Virtual base classes must be initialized first
    : Article(n, p, q),
      Food(n,p,q,d), Liquid(n,p,q,c), _vineyard(v)
    { }
  ...
};
```

Inheritance conflicts are solved using the scope resolution operator ‘::’

```
#include "Article.h"
class Food : virtual public Article {
  ...
public:
  float price() const { return(_basePrice * 1.05); }
  ...
};

class Liquid : virtual public Article {
  ...
public:
  float price() const
    { return(cap * _basePrice * 1.05); }
  ...
};

class Wine : public Food, public Liquid {
  ...
public:
  float jeroboamPrice() const
    { return(Liquid::price() * 4.); }
    // Explicit invocation
    // of function ‘price()’ from class Liquid
  ...
};
```

SUMMARY

- Data members are implicitely duplicated in case of repeated inheritance

  Repeated inheritance may be disabled by declaring common base classes as **virtual**

- A class may inherit different members of same name

  For **each use** of such a name, ambiguity must be *explicitely* removed with the help of the scope resolution operator

PART 6: TEMPLATES

Templates implement genericity

Template functions

- The definition of a template function is parameterized by formal types so as to be applied to different types
- A formal type may represent the type of an argument of the function or the type of the result of the function

```
template <class Type>
  int in(Type el, Type a[], int sz)
// Does array ‘a’ (of size ‘sz’) include ‘el’?
    {
      int i;
      for(i = 0 ; (i < sz) && (a[i] != el) ; i++);
      return (i != sz);
    }
```

When a template function is called, the template is **instantiated**, if necessary: An effective type is substituted for each occurrence of a formal type

```
int main() {
  char cTab[256];
  Article aTab[100];
  Article art("Mosquito-net", 235.60, 1);
  int i, j, k;
  i = in('C', cTab, 256);  // Call #1
  j = in(art, aTab, 100);  // Call #2
  k = in('@', aTab, 100);
    // Error: 1st and 2nd arguments
    // have not the same type
}
```

```
// Function generated by call #1
int in(char el, char a[], int sz) {
  int i;
  for(i = 0 ; (i < sz) && (a[i] != el) ; i++);
  return (i != sz);
}
```

```
// Function generated by call #2
int in(Article el, Article a[], int sz) {
  int i;
  for(i = 0 ; (i < sz) && (a[i] != el) ; i++);
  return (i != sz);
}
```

Template classes

- The definition of a template class is also parameterized by formal types
  $\Rightarrow$ avoid the definitions of many similar classes to describe a same concept applying to different data types
- A formal type may be used in:
  - the declaration of a data member or a local variable
  - the declaration of a parameter (of a function)
  - the declaration of the result of a function
  - an explicit conversion, etc.

```
// File Stack.h: Rudimentary definition of a stack
template <class Type>
  class Stack {
  private:
    Type* _stack;  // Implementation of the stack
    int _size;     // Size of the stack
    int _topIndex; // Index of the top of the stack
  public:
    Stack(int sz);
    Type top() const;
    void push(Type el);
    ...
};
```

```
#include <iostream.h>
#include "Stack.h"
// DEFINITION OF THE (TEMPLATE) FUNCTION MEMBERS
template <class Type>
  Stack<Type>::Stack(int sz) {
    _topIndex = -1;
    _stack = new Type[_size = sz];
}
template <class Type>
  Stack<Type>::~Stack() {
    if (!empty())
      cerr << "WARNING: Stack is not empty!" << endl;
    delete [] stack;
}
template <class Type>
  void Stack<Type>::push(Type el) {
    if (_topIndex == (_size - 1)) {
      cerr << "WARNING: Stack is full!" << endl;
      abort();
      }
    stack[++_topIndex] = el;
}
template <class Type>
  void Stack<Type>::pop() {
    if (empty()) {
      cerr << "WARNING: Stack is empty!" << endl;
      abort();
      }
    _topIndex--;
}
```

- A specific type of a template class is obtained by giving effective types for the formal types

  An effective class is then instantiated by substituting an effective type for each occurrence of the corresponding formal type

```
#include "Stack.h"
int main() {
  Stack<int> istack(500);
  Stack<Article*> astack(100);
  a = new
    Article("Luneville faience tureen", 649.54, 2);
  c = new
    Clothing("Overalls", 349.50, 200, 45, "khaki");
  ...
  istack.push(666);
  ...
  astack.push(a);
  astack.push(c);  // Polymorphism
  ...
}
```

Formal parameter types cannot be constrained

```
#include <iostream.h>
template <class Type> class SortedList {
protected:
  Type* _list; // Implementation of the list
  int _size;   // Size of the list
  int _last;   // Index of the last element
public:
  void add(Type el);
  ...
};
template <class Type>
  void SortedList<Type>::add(Type el) {
    if (_last == (_size - 1)) {
      cerr << "WARNING: List is full!" << endl;
      abort();
    }
    else {  // Naive algorithm
      for (int i = 0 ;
           (i <= _last) && (_list[i] <= el) ;
    // '<=' is supposed to be defined for any type
           i++);
      for (int j = _last ; j >= i ; j--)
        _list[j+1] = _list[j];
      _list[i] = el;
      _last++;
    }
}
...
```

- In case of separate compilation, an error may occur, depending on effective types, as operator <= is not defined for any type

```
# include "SortedList.h"
int main() {
  SortedList<int> iList(500);
  SortedList<Article> aList(100);
  Article a("Walnut toffees", 1.50, 500);
  ...
  iList.ajouter(55);
    // OK: '<=' is defined for integers
  ...
  aList.ajouter(a);
    // Error: '<=' is not defined for type Article
  ...
}
```

SUMMARY

- Templates may be seen as a pragmatic implementation of genericity:

  `Stack<Article>` and `Stack<Clothing>` are two independant instances of `Stack<T>`
  although `Clothing` derives from `Article`

- No Eiffel-like constrained genericity

  Operations allowed on objects of the formal parameter type `T` are specified using inheritance

  `SORTED_LIST[T->COMPARABLE]`

  An effective parameter type must be a subtype of `COMPARABLE`, which is supposed to implement operation <=

  $\Longrightarrow$ the correction of a generic class can be statically checked in case of separate compilation

PART 7: EXCEPTION HANDLING

exception = run-time anomaly
e.g. an access to an array outside of its bounds

The (portion of) program detecting an anomaly can raise, or **throw**, an exception

- An exception is an object (of class type)
  $\Longrightarrow$ this object is created by calling the class constructor

- A predefined exception class hierarchy is supplied by the C++ library standard

- Built-in features to deal with exceptions are provided:
  - **throw**: to raise an exception
  - **catch**: to define the handler of an exception

```
class IntStack {
private:
  int* _stack;   // Implementation of the stack
  int _size;     // Size of the stack
  int _topIndex; // Index of the top of the stack
public:
  IntStack(int s)
    {
      _topIndex = -1;
      _stack = new int[_size = s];
    }
  ...
  void pop()
    {
      if (empty())
        throw PopOnEmpty();  // No parameter
      _topIndex--;
    }

  void push(int v)
    {
      if (_topIndex == (_size - 1))
        throw PushOnFull(v);  // One parameter
      stack[++_topIndex] = v;
    }
  ...
};
```

The try block and catch clauses

```
int main() {
// A TRY BLOCK ENCLOSES STATEMENTS THAT CAN THROW
// EXCEPTIONS (IT DEFINES A LOCAL SCOPE)
  try {
    IntStack s(100);
    // Code including calls to pop() and push()
    return 1;
  }
// CATCH CLAUSES
  catch(PopOnEmpty) {
    // Handler of the PopOnEmpty exception
    // Cannot refer to stack 's' here
  }
  catch(PushOnFull &exceptionObj) {
    // Handler of the PushOnFull exception
    // Cannot refer to stack 's' here
  }
  catch(...)  {
    // Catch-All Handler (for all other exceptions)
    // Cannot refer to stack 's' here
  }
// EXECUTION CONTINUES HERE (cannot refer to 's')
// C++ exception handling is nonresumptive: once
// the exception has been handled, execution doesn't
// resume where the exception was originally thrown.
  return 0;
}
```

The exception classes

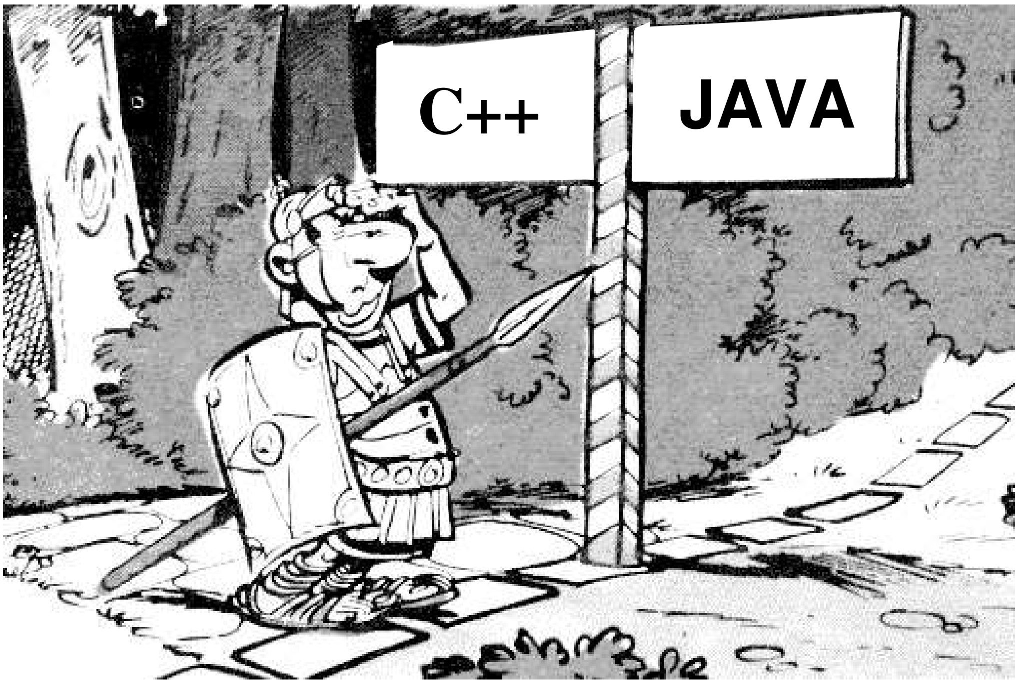
```
// File PopOnEmpty.h
class PopOnEmpty {
public:
  // Constructor with no parameter
  PopOnEmpty() { }
};
```

```
// File PushOnFull.h
class PushOnFull {
private:
  // Value that cannot be pushed on the stack
  int _valueToBePushed;
public:
  // Constructor with one parameter representing
  // the value that cannot be pushed on the stack
  PushOnFull(int v) : _valueToBePushed(v) { }
  // Access function to the value
  int valueToBePushed() { return _valueToBePushed; }
};
```

Definitions of the handlers

```
#include <iostream.h>
#include "IntStack.h"
#include "PopOnEmpty.h"
#include "PushOnFull.h"
int main() {
  try {
    ...
    return 1;
  }
  catch(PopOnEmpty) {
    cerr << "ERROR: Stack is empty!" << endl;
    return errorCode38;  // Terminates execution
  }
  catch(PushOnFull &exceptionObj) {
    cerr << "ERROR: Stack is full: Cannot push "
         << exceptionObj.valueToBePushed() << endl;
    return errorCode39;  // Terminates execution
  }
  catch(...)  {
    cerr << "ERROR: Oops, I forgot some exception!"
         << endl;
    return errorCode00;  // Terminates execution
  }
}
```

CONCLUSION



So, C++ or Java?

C++

Designed by adding facilities to an existing language (C) without actual care about uniformity

- C is a low-level language $\Rightarrow$ C++ too
- C syntax is cryptic $\Rightarrow$ C++ syntax too
- Two layers: C and classes

  A programmer may use just one layer and not the other
- Facilities (casting, overloading, destructors, templates, etc.) are neither easy to understand nor easy to use

  It may be very tricky to understand an elementary instruction like `x=y;` *because operator* `=` *may be overloaded and implicit casting operations may be defined*
- The multiple facilities allow (and encourage) programmers to ignore the fundamental principles of (object-oriented) programming

$\Rightarrow$ C++ is ill-suited to large software design by programming teams

$\Rightarrow$ but C++ probably is a good language to generate code for a higher-level language

As **efficiency is privileged**, C++ designers' choices go against programmers' comfort:

- An object is not implicitely represented by its address
  $\Rightarrow$ programmers have to explicitely manipulate pointers

- **No garbage collector**
  $\Rightarrow$ programmers must define destructors

- Static binding is the default

- Repeated inheritance is the default

Despite all that, C++ is still intensively used

mainly because of its compatibility with C

May the future belong to... Java?