Institut Mines-Télécom
Didier GUERIOT (Dpt. ITI)
Johanne VINCENT (Dpt. INFO)
Reda BELLAFQIRA (Dpt. ITI)
Didier.Gueriot@imt-atlantique.fr
Johanne.Vincent@imt-atlantique.fr
Reda.Bellafqira@imt-atlantique.fr

Advanced C++ programming

- **Introduction to C++**
  - → C++: from C and beyond
  - → Classes, objects and lifetime (vs. JAVA)
  - → Oriented-Object Programming (inheritance, polymorphism)
- **Memory management & object manipulation**
  - → References, operators, « copy » object construction
  - Slot 5 → « move » object construction, lambda functions
- **Template vs OO programming**
  - → Template functions and classes
- **The Standard Template Library**
  - → Containers, iterators and algorithms
  - → Using sequence & associative containers …
- **Smart pointers (STL & Boost)**

C++03
C++11
C++14
C++17
C++20

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

A “move” constructor? (1/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

TP 7.3

- What’s a lvalue ?
  - Expression whose address can be retrieved

```
Image ReturnBigImage () {
  Image F (100 , 100) ;
  return F ;
}
```

```
int main() {
  Image I2 = ReturnBigImage () ;
}
```

➡ An assignment is possible on a lvalue

`ReturnBigImage()` is not a reference to F : it’s a temporary object on the stack

- What’s a rvalue ?
  - Everything that is not a lvalue !!!
    - Temporary object,…

➡ Not possible to manipulate **explicitly** before C++11

- Compiler help through « const lvalues » cast: `const Image&`

➡ C++11 brings a new type: « rvalue references »: `Image&&`

A “move” constructor? (2/3)

C++11

TP 4.2

- Move constructor: T (T&&)
  - ▶ Useful to build a new object B from an existing one A, “stealing” all resources from A instead of making useless copies

```
struct Image {
  int   width , height ;
  byte* image ;

  Image (int w , int h) : width(w) , height(h) {
    image = new byte [w*h] ;
  }

  // Copy constructor
  Image (const Image& I) : width(I.width) , height(I.height) {
    image = new byte [I.width * I.height] ;
    memcpy (image , I.image , I.width*I.height) ;
  }

  // Move constructor
  Image (Image&& I) :
        width(I.width) , height(I.height) , image(I.image) {
    I.width = I.height = 0 ;
    I.image = NULL ;
  }
} ;
```

```
Image ReturnBigImage () {
  Image F (100 , 100) ;
  return F ;
}
```

```
int main() {

  // The Image copy constructor is not called
  // as the result of 'ReturnBigImage()'
  // is a temporary object (type: Image&&)
  // => the Image move constructor is then
  // called to build I2
  Image I2 = ReturnBigImage () ;

  // The memory address in I2.image is the
  // same as the memory address in F.image:
  // No useless copy of the image array
  // 'F.image' has been performed!
}
```

▶ Before C++11, only the copy constructor could have been called

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

A "move" constructor? (3/3)

C++11

- **Goal:**
  - avoid unnecessary copies using « rvalue references »
- 1 **When functions return values (by priority)**
  - « RVO » then « move ctor » then « copy ctor »
- 2 **Wider move semantics introduced in C++**
  - '`std::move()`' makes the compiler **consider** a lvalue as a rvalue, involving move constructor and move assignment use

move ctor

2 x move =

```
// Echange sans copies inutiles
void swap (Image& a, Image& b) {
  Image tmp = std::move(a) ;
  a = std::move(b) ;
  b = std::move(tmp) ;
}
```

VS

```
// Echange avec copies inutiles
void swap (Image& a, Image& b) {
  Image tmp (a) ;
  a = b ;
  b = tmp ;
}
```

copy ctor

2 x copy =

More on move semantics

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

TP 4.4

- Which function is called?

```
void f (int& i) {
  std::cout << "lvalue ref: " << i << "\n" ;
}

void f (int&& i) {
  std::cout << "rvalue ref: " << i << "\n" ;
}

void g (int& i) {
  std::cout << "lvalue ref: " << i << "\n" ;
}

int h () {
  int k = 10 / 10 ;
  return k ;
}
```

```
int main() {

  int i = 77 ;

  f(i) ;                // lvalue ref called
  f(99) ;               // rvalue ref called
  f(std::move(i)) ;     // rvalue ref called

  g(i) ;                // lvalue ref called
  g(99) ;               // compilation failed

  f(h()) ;              // rvalue ref called
  g(h()) ;              // compilation failed

  return 0 ;
}
```

99 or the integer returned by h() are not int&

If function f were overloaded with 'void f (int)', the compiler would give an error when calling:
- -f(99) because two valid choices `void f(int i)` and `void f(int&& i)`,
- -f(i) because two valid choices `void f(int i)` and `void f(int& i)`.

```
void f (int i) {
  std::cout << "lvalue : " << i << "\n" ;
}
```

Operators

- **C++ concept**
  - An operator performs specific computations on values.
  - Built-in operators are provided but they can be overloaded.
- **Unary operators (arithmetic, logical, …)**
  - `+ - * & ~ ! ++ -- -> ->*`
- **Binary operators (arithmetic, logical, assignment, …)**
  - `+ - * / % ^ & | << >> > < == != += -= *= /= %= ^= |= && ||`
  - `() , = , [] , << , new new[] delete delete[]`
- **You may define operators for your classes**
  - Coded semantics vs. commonly expected semantics

Overloading operator ( )

TP 6.4

- **Creating 'functors' (or function object)**
  - A functor is a class whose objects act like a function (they are 'called' using the standard function call syntax)
    - the operator () must be overloaded for the class
  - Unlike function, a functor keeps its **context** (like object) between calls

```
class Polynomial {
  unsigned int _degree ;
  double* _coeff ;
public :
  Polynomial (int degree , double* coeff) : _degree (degree), _coeff (coeff) {}
  double operator() (double x) const ;
}

double Polynomial::operator() (double x) const {
  double res = 0.0 ;
  for (unsigned int k = _degree-1 ; k >= 0 ; k--) {
    res = res*x + _coeff[k] ;
  }
  return res ;
}
```

```
double coeff[2] = { 2.0 , 5.0 } ;
Polynomial P (2 , coeff) ;

// P is an object!
double p1 = P(10.0) ; // = 52.0
double p2 = P(20.0) ; // = 102.0
```

New syntax to declare a function

C++11

- No return type inference

```
int multiply (int x , int y) ;
```

```
auto multiply (int x , int y) -> int ;
```

  - Why it may be interesting? (example)

```
struct LinkedList
{
  struct Link { /* ... */ } ;

  Link* erase (Link* p) ;
} ;

LinkedList::Link* LinkedList::erase (Link* p)
{ /* ... */ }
```

```
struct LinkedList
{
  struct Link { /* ... */ } ;

  Link* erase (Link* p) ;
} ;

auto LinkedList::erase(Link* p) -> Link*
{ /* ... */ }
```

- Automatic return type deduction

C++14

```
auto square (int n)
{
  return n*n ;
}
```

Lambda functions (1/3)

C++11

- Anonymous functions or closures

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

int main() {

  // A lambda function object 'func' is created
  auto func = [] () { cout << "Hello, World !" << endl ; } ;
  // Calling 'func'
  func () ;

  //  Local creation + immediate call
  [] { cout << "Hello, World again !" << endl ; } () ;
}
```

These "functions" return nothing (**void**)

- Full syntax

`[ capture-list ] ( params ) mutable -> ret { body }`

```
auto z1 = [] (int k) { return k+1 ; } ;    // the compiler infers the function
                                           //     returns an integer value
auto z2 = [] () -> int { return 1 ; } ;    // you can also give the return type (here int)
```

```
int k = z1 (10) ;       // k = 11
int m = 1 + z2 () ;     // m = 2
```

TP 6.4

Lambda functions (2/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- **Capturing external variables (from the enclosing scope)**
  - where the lambda is **declared**, not where it is called

```
[ capture-list ] ( params ) mutable -> ret { body }
```

		Capturing ▶ what information "body" may have access* to?
~ function	`[]`	nothing
~ functor	`[&]`	all variables from the enclosing scope, through reference
	`[=]`	a copy of all variables from the enclosing scope (by value)
	`[=,&foo]`	all variables (by copy), except `foo` by reference
	`[bar]`	only a copy of `bar` (nothing else is captured)
	`[this]`	only the `this` pointer of the enclosing class

* from the "body" of the lambda

- Keyword: **`mutable`**?

Allow « body » to update locally the variables captured by value (only the copy inside the lambda, obvously) directly or through their « non const » methods ▶ by default, a lambda is immutable (≈ a operator () const) then no modification is allowed for the captured variables (i.e. "the attributes of the lambda object")

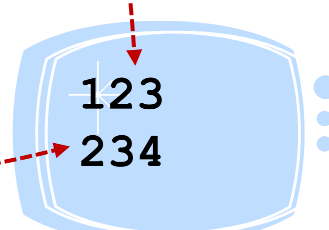
Lambda functions (3/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- Example: capture & runtime

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```



c | m2	
b | m2	~~2~~ 4
a | m2	~~1~~ 4
m2 | m1	'm2(1,2,&c|m1)'
c | m1	
b | m1	
a | m1	~~1~~ 3
m1	'm1(1,&b,&c)'
c	~~1~~ ~~2~~ ~~3~~ 4
b	~~1~~ ~~2~~ 3
a	~~1~~ 2

Lambda functions (3/3)

- Runtime (pause 1)

```
int a = 1, b = 1, c = 1 ;
auto m1 = [a, &b, &c]() mutable {
  auto m2 = [a, b, &c]() mutable {
    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;
  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;
} ;

a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

1

c	1
b	1
a	1

Lambda functions (3/3)

- Runtime (pause 2)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;

a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

2

m1	'm1(1,&b,&c)'
c	1
b	1
a	1

Lambda functions (3/3)

- Runtime (pause 3)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

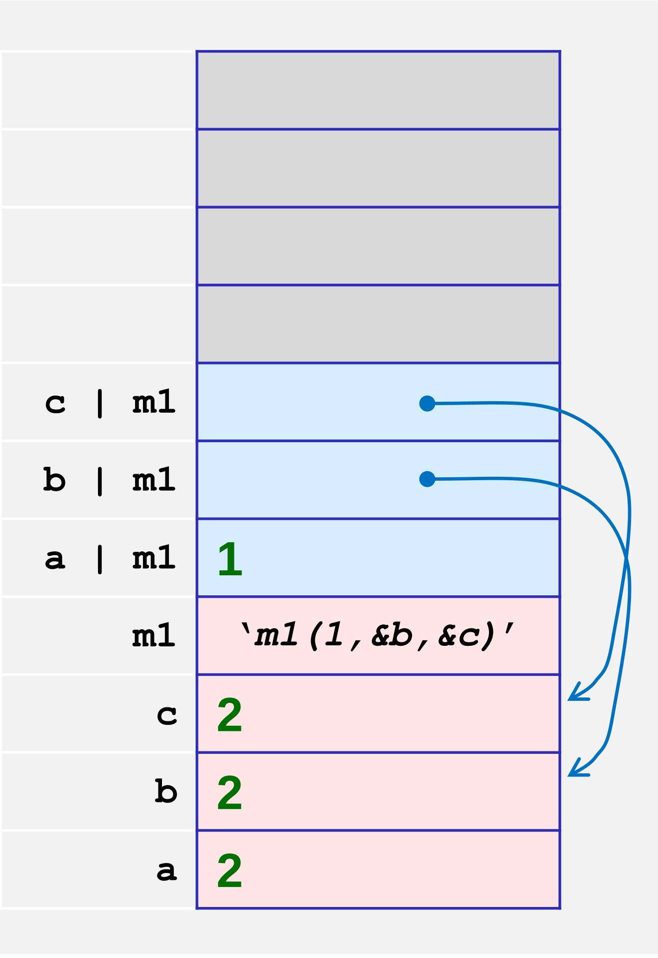
3

m1	'm1(1,&b,&c)'
c	2
b	2
a	2

Lambda functions (3/3)

- Runtime (pause 4)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {
  auto m2 = [a, b, &c]() mutable {
    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;
  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;
} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```



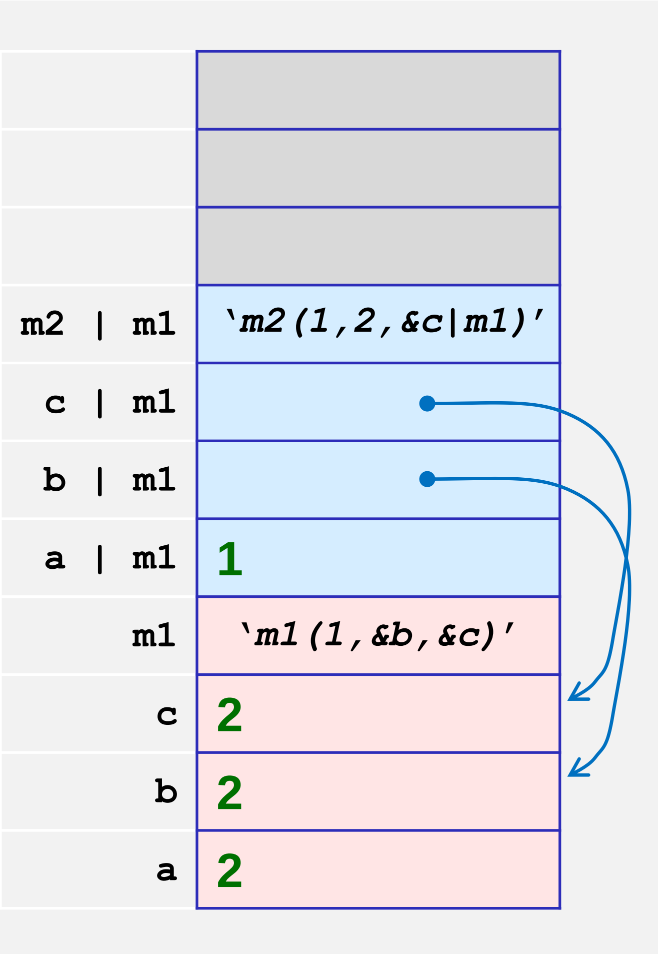
Lambda functions (3/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

- Runtime (pause 5)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

5



Lambda functions (3/3)

- Runtime (pause 6)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;

a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

6

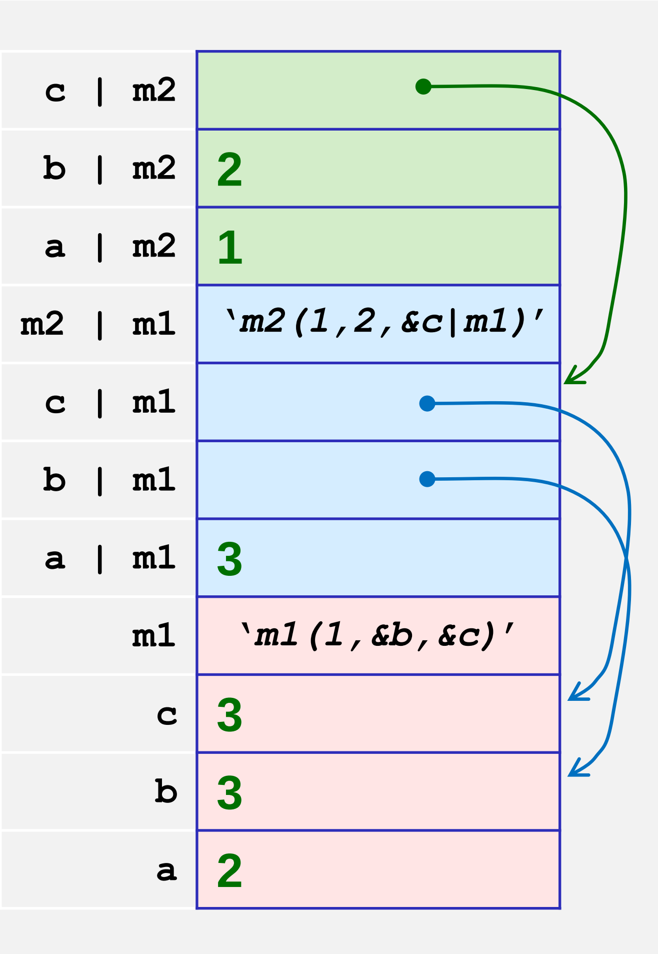
m2 | m1	'm2(1,2,&c|m1)'
c | m1	
b | m1	
a | m1	3
m1	'm1(1,&b,&c)'
c	3
b	3
a	2

Lambda functions (3/3)

- Runtime (pause 7)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

7

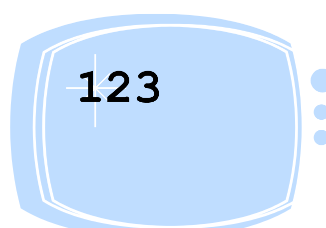


Lambda functions (3/3)

- Runtime (pause 8)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

c | m2	
b | m2	2
a | m2	1
m2 | m1	'm2(1,2,&c|m1)'
c | m1	
b | m1	
a | m1	3
m1	'm1(1,&b,&c)'
c	3
b	3
a	2



Lambda functions (3/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

- Runtime (pause 9)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

9

c | m2	
b | m2	4
a | m2	4
m2 | m1	'm2(1,2,&c|m1)'
c | m1	
b | m1	
a | m1	3
m1	'm1(1,&b,&c)'
c	4
b	3
a	2

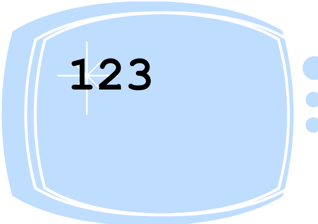
```
123
```

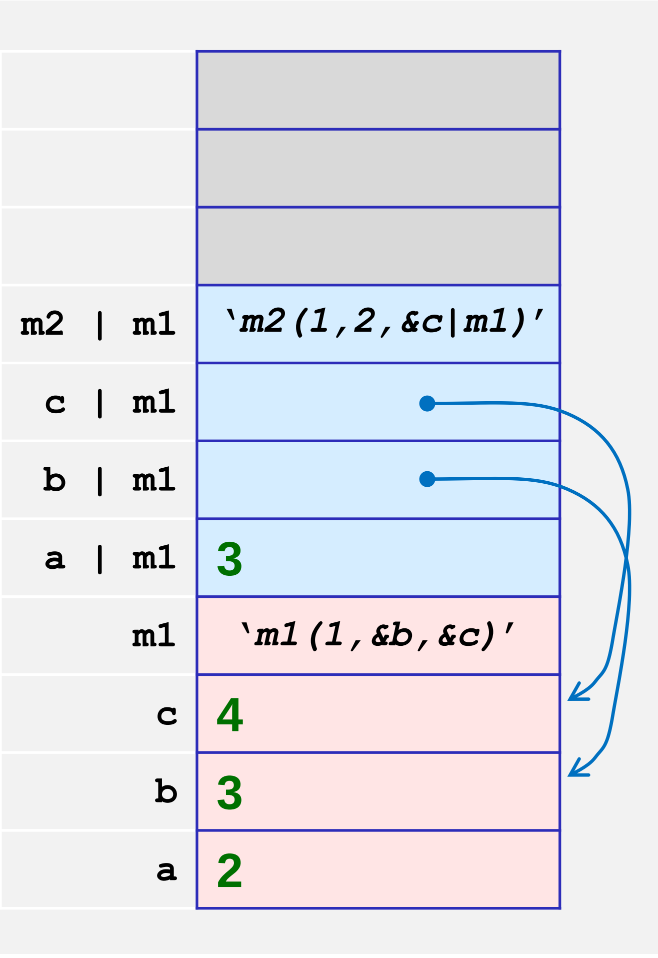
Lambda functions (3/3)

- Exécution (pause 10)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;
} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

10

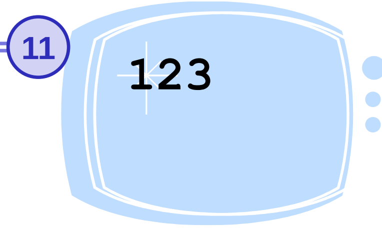


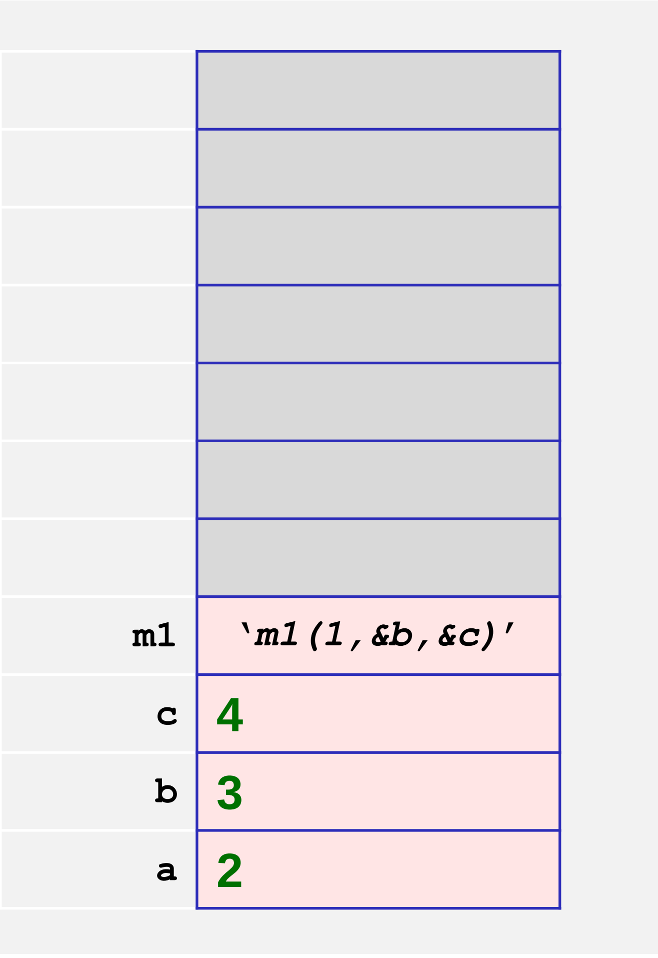


Lambda functions (3/3)

- Runtime (pause 11)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```



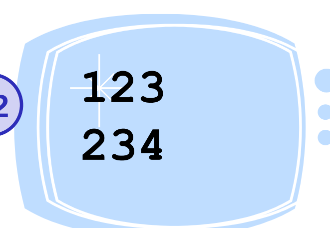


Lambda functions (3/3)

- Runtime (pause 12)

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

12



m1	'm1(1,&b,&c)'
c	4
b	3
a	2

Lambda functions (3/3)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- Example: capture & runtime

```
int a = 1, b = 1, c = 1 ;

auto m1 = [a, &b, &c]() mutable {

  auto m2 = [a, b, &c]() mutable {

    cout << a << b << c << endl ;
    a = 4 ; b = 4 ; c = 4 ;

  } ;

  a = 3 ; b = 3 ; c = 3 ;
  m2() ;

} ;


a = 2 ; b = 2 ; c = 2 ;
m1() ;
cout << a << b << c << endl ;
```

c | m2	
b | m2	~~2~~ 4
a | m2	~~1~~ 4
m2 | m1	'm2(1,2,&c|m1)'
c | m1	
b | m1	
a | m1	~~1~~ 3
m1	'm1(1,&b,&c)'
c	~~1~~ ~~2~~ ~~3~~ 4
b	~~1~~ ~~2~~ 3
a	~~1~~ 2

```
123
234
```

Type inference (1/4)

C++11

- **auto**: introduction of automatic type inference
  - nice when declaring variables (the compiler is in charge)

```
int x = 4 ;
T* p = new T(12) ;
```

```
auto x = 4 ;
auto p = new T(12) ;
```

  - even nicer and useful when using templates / iterators

```
map<string,list<int>::iterator>::const_iterator it = m.cbegin() ;
```

```
auto it = m.cbegin() ;
```

```
template <typename BuiltType, typename Builder>
void makeAndProcessObject (const Builder& builder) {
  BuiltType val = builder.makeObject() ;
  ...
}

MyObjBuilder builder ;
makeAndProcessObject<MyObj>(builder) ;
```

```
template <typename Builder>
void makeAndProcessObject (const Builder& b) {
  auto val = builder.makeObject() ;
  ...
}

MyObjBuilder builder ;
makeAndProcessObject(builder) ;
```

? And how to name the type if **`makeAndProcessObject()`** returns **`val`**

Type inference (2/4)

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

C++11

- **auto: references, pointers & const**
  - default: by value (add **&** if a reference is needed)

```
int& foo() ;

auto bar = foo() ;
```

?

bar : int ou int&

```
auto  bar = foo() ;     // bar : int
auto& bar = foo() ;     // bar : int&
```

  - no issue with pointers

```
int* foo() ;

auto bar = foo() ;     // bar : int*
```

OU

```
int* foo() ;

auto* bar = foo() ;     // bar : int*
```

  - **const** to be added if needed

```
int& foo() ;
```

```
const auto& bar = foo() ;  // bar : const int&
```

```
int* foo() ;
const int* cfoo() ;
```

```
const auto* bar1 = foo() ;   // bar1 : const int*
auto bar2 = cfoo() ;         // bar2 : const int*
```

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Type inference (3/4)

C++11

- **`decltype`: get an expression type**
  - when compiling (same behavior as `sizeof`)

```
int x = 3 ;
const int && foo() ;
struct A { double x ; } ;
const A* a = new A() ;

decltype(x)      y1 = x ;      // y1: type int
auto             y1 = x ;      // y1: type int
decltype((x))    y2 ;          // y2: type int&
auto             y2 = (x) ;    // y2: type int
decltype(foo())  y3 ;          // y3: type const int&&
auto             y3 = foo() ;  // y3: type int
decltype(a->x)   y3 ;          // y3: type double
decltype((a->x)) y4 ;          // y4: const double&
```

Adding ()

C++14

- **`decltype(auto)`** and “perfect forwarding”

```
int& foo() ;

auto           bar1 = foo() ;   // bar1 : int
decltype(auto) bar2 = foo() ;   // bar2 : int&
```

Type inference (4/4)

- **auto/decltype/decltype(auto)**

```
template <typename Builder>
void makeAndProcessObject (const Builder& builder) {
  auto val = builder.makeObject() ;
  ...
}

MyObjBuilder builder ;
makeAndProcessObject(builder) ;
```

And how to name the type if **makeAndProcessObject()** returns **val**

```
template <typename Builder>
auto makeAndProcessObject (const Builder& builder) -> decltype(builder.makeObject()) {
  auto val = builder.makeObject() ;
  ...
  return val ;
}

MyObjBuilder builder ;
auto res = makeAndProcessObject(builder) ;  // attention si référence !
```

C++11

```
template <typename Builder>
decltype(auto) makeAndProcessObject (const Builder& builder) {
  auto val = builder.makeObject() ;
  ...
  return val ;
}

MyObjBuilder builder ;
decltype(auto) res = makeAndProcessObject(builder) ;  // exact type (reference: ok)
```

C++14

Some “dev” tips!

• C++ « attributes » or annotation

C++11

- Standard ways of extending the language
  - Directives: `#pragma`
  - Compiler specific annotations:

    `attribute__((…))` ou `__declspec()`

➡ New standard syntax: `[[`*`attr`*`]]`

- Available attributes
  - in the standard: `[[noreturn]]`, `[[carries_dependency]]`
  - otherwise compiler specific (clang, g++, Microsoft, …)

C++14

- New interesting attribute:
  - `[[deprecated]]` or `[[deprecated (« message »)]]`

  ▶ make the compiler issue a warning when compiling to indicate such a flagged entity will not be available in next versions of your code.

Before going further…

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

- You must be able to write C++ classes using encapsulation principles
  - including relevant use of « `const` »
- You must be able to create / copy / manipulate objects (stack or heap allocated) taking into account their lifetime
  - avoid memory leaks, avoid non valid reference to objects, …
  - know **when** and **why** copy / move constructors or assignments are called (even silently) and what are the consequences
- You must be able to fully use polymorphism
  - virtual methods, abstract classes and interface

Practice

- **Using functors or lambda functions**
  - Implementation a "Lumberjack" functor / lambda function
  - Differences with functions

- **Going further with move semantics...**
  - Avoid useless copy of pixels buffers when passing trees