

Advanced C++ programming

♦ Introduction to C++

- C++: from C and beyond
- Classes, objects and lifetime (vs. JAVA)
- Oriented-Object Programming (inheritance, polymorphism)

Slot 2

♦ Memory management & object manipulation

- References, « copy » / « move » object construction
- Overloading operators

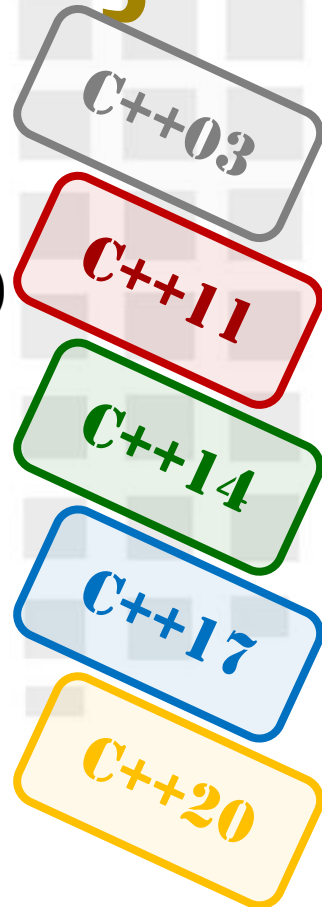
♦ Template vs OO programming

- Template functions and classes

♦ The Standard Template Library

- Containers, iterators and algorithms
- Using sequence & associative containers ...

♦ Smart pointers (STL & Boost)

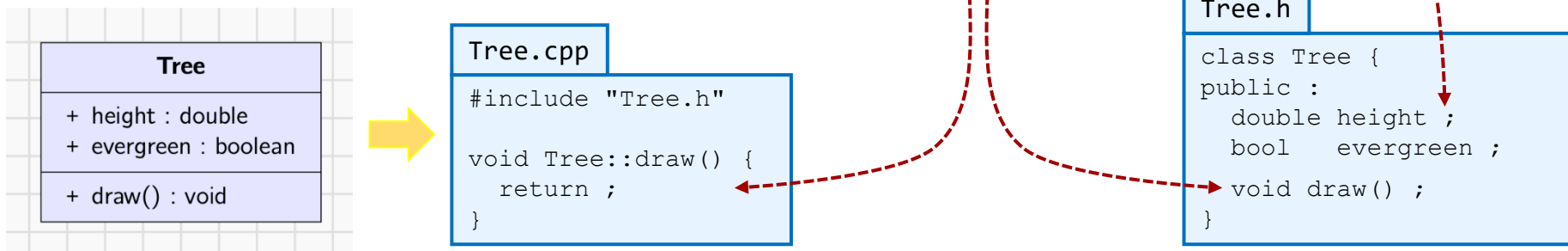




Object vs. class (1/3)

• What's a class ?

- “A **user-defined type** or data structure having data (attributes) and functions (methods)”
- Basically, it's just a “map” or instructions giving:
 - how to build and destroy entities (**objects**) complying with this map,
 - what's inside such objects (types of data): **attributes**
 - how to interact with such objects: **methods**





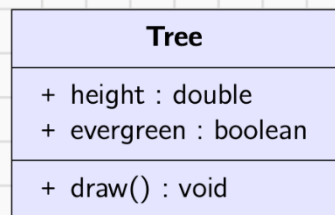
Object vs. class (2/3)

• What's a class ?

• C++ standard code organization (≠ JAVA)

• Two separate files

- declaration (.h) : attributes (type, size) & methods signatures specification
- definition (.cpp) : methods implementation (their C++ code)



Tree.h

```
class Tree {
public :
    double height ;
    bool    evergreen ;

    void draw() ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw() {
    return ;
}
```

• Desired objectives

- when building the binary executable, prevent from useless class recompiling
- split up the class interface (public) from its implementation (often more private)



Object vs. class (3/3)

• What's an object ?

- A class is like a **mold** allowing to create different **objects** (called **instances** of the class) ► each instance represents a specific entity thus filling a specific area in memory.

• Example :

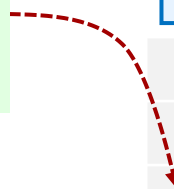
```
// A 50 meters high evergreen tree
Tree a ;

// Another 120 meters high deciduous tree
Tree b ;
```

Tree.h

```
class Tree {
public :
    double height ;
    bool    evergreen ;
    void draw() ;
}
```

- The **Tree** class represents, in the computer :
 - what's a tree (tree height and if evergreen)
 - how the program may use it (here, it can only draw itself)
- **a** and **b** are two instances of the **Tree** class
- **a** and **b** are two variables (objects) of type **Tree**
- **a** and **b** are two different trees



		0x011C
	false	0x0118
	120.0	0x0114
b		0x0110
	true	0x0108
	50.0	0x0104
a		0x0100



Using objects (1/2)

- Access to attributes and methods calls
 - « . » from one object
 - « -> » from a pointer to one object

Tree.h

```
class Tree {
public :
    double height ;
    bool evergreen ;
    void draw() ;
}
```

```
// A tree
Tree a ;
```

```
// Another tree
Tree b ;
```

```
// c is a pointer
// to Tree a
// (*c) is Tree a
Tree* c = &a ;
```

```
// Modifying attributes values (write access)
// => set the tree height and if it's evergreen
a.height      = 50.0 ; // Height of tree a becomes 50.0 meters
b.evergreen   = true ; // Tree b becomes evergreen
(*c).height  = 100.0 ; // Height of tree a becomes 100.0 meters
c->evergreen  = false ; // Tree a becomes deciduous
```

```
// Getting attributes values (read access)
// => get the tree height and if it's evergreen
cout << "Height of a = " << a.height << endl ;
cout << "b evergreen ? " << b.evergreen << endl ;
cout << "Height of a = " << (*c).height << endl ;
cout << "a evergreen ? " << c->evergreen << endl ;
```

```
// Calling the draw() method of one tree
// => from the object or the object address
a.draw() ; // Tree a draws itself
b.draw() ; // Tree b draws itself
(*c).draw() ; // Tree a draws itself
c->draw() ; // Tree a draws itself
```

		0x011C
	true	0x0118
	?	0x0114
b		0x0110
	false	0x0108
	100.0	0x0104
a		0x0100



Using objects (2/2)

• Method call ~ function call

- An extra variable is added on the stack

- **this** : Inside the method, **this** is the address of the object from which the method is called (thus, this specific object is known)

Tree.h

```
class Tree {
public :
    double height ;
    bool    evergreen ;

    void draw() ;
}
```

```
int main (int argc , char* argv[]) {
    Tree a ; a.evergreen = true ; a.height = 0.0 ;
    a.draw () ;
    return 0 ;
}
```

Tree.cpp

```
#include "Tree.h"

void Tree::draw () {
    int nb = 4 ;
    if (evergreen)
        cout << "Green tree" ;
    else
        cout << "Yellow tree" ;
}
```

```
void Tree::draw () {
    ...
    if (this->evergreen)
    ...
}
```

Green tree

(1)	(2)	(3)	(4)
		nb	
		4	
	this	this	
	&a	&a	
	true	true	true
a	a	a	a
0.0	0.0	0.0	0.0



Objects birth and death (1/2)

• Two specific methods are always called at both ends of an object lifetime (provided by you or the compiler)

• a **constructor** : called after the system reserves enough memory to store object attributes values

Tree()

► initialization of object attributes values, resources allocation

• the **destructor** : called just before the system frees the memory reserved for the object

~Tree()

► release of resources the object still owns, ...

Tree.h

```
class Tree {
public :
    double height ;
    bool    evergreen ;

    Tree () ;
    ~Tree() ;
}
```

Tree.cpp

```
#include "Tree.h"

Tree::Tree():height(10.0), evergreen(false)
{}

Tree::~~Tree() {}
```

```
int main (...)
{
    Tree a ;
}
```

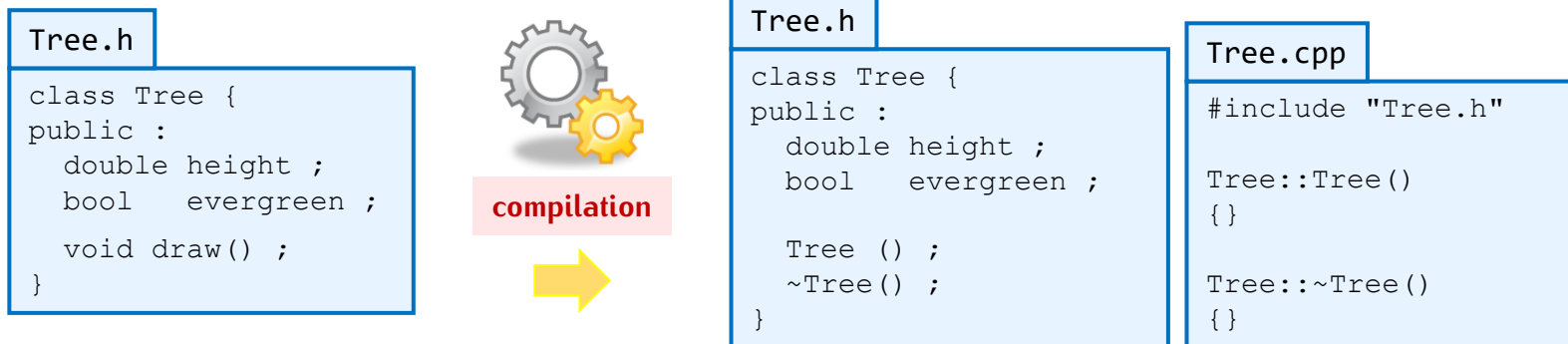
		0x010C
	false	0x0108
	10.0	0x0104
a		0x0100



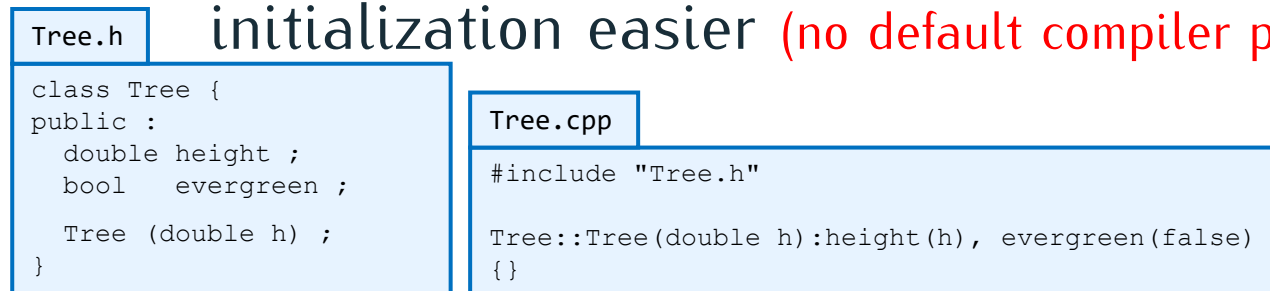
Objects birth and death (2/2)

• Remarks

- A default constructor (no parameter) and the destructor are provided by the compiler only when you don't.



- You may also provide several constructors to make object initialization easier (no default compiler provided one in that case)



Objects lifetime (1/2)

- How are the objects you created, destroyed ?
- Depends on the kind of memory allocation you asked
 - If on the **stack** (the compiler is in charge: destructor is called and memory released when exiting object scope)
 - If in the **heap** (you are responsible for requesting the destruction, memory leaks if you don't)

```
#include "Tree.h"

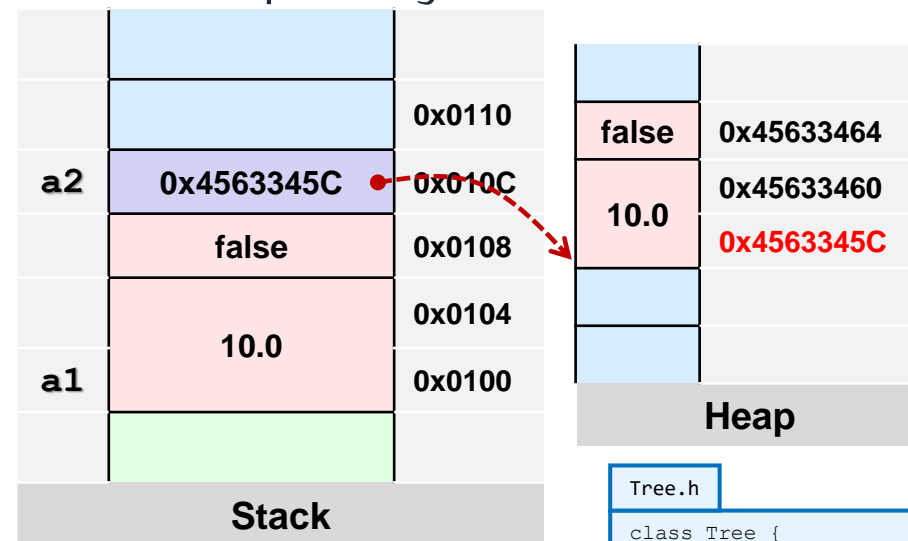
int main (int argc , char* argv[]) {

    // We want to create 2 trees
    Tree a1 ;                // on the stack
    Tree* a2 = new Tree ;    // in the heap

    // Ask them to draw themselves
    a1.draw() ;
    a2->draw() ;

    // We need to explicitly request the destruction
    // of the heap-allocated tree though the pointer a2
    delete a2 ;

}
```



Tree.h

```
class Tree {
public :
    double height ;
    bool evergreen ;
}
```

The compiler is only responsible for the lifetime of the variables you declare i.e. Tree a1 and pointer to Tree a2 (not the tree itself), according to their **scopes**.



Objects lifetime (2/2)

• What is really happening?

```
#include "Tree.h"

int main (int argc , char* argv[]) {

    // We want to create 2 trees
    Tree a1 ; // on the stack
    Tree* a2 = new Tree ; // in the heap

    // Ask them to draw themselves
    a1.draw() ;
    a2->draw() ;

    // We need to explicitly request
    // the destruction of the heap-allocated
    // tree through the pointer a2
    delete a2 ;
}
```

a1 lifetime

- 1) System allocates memory on the **stack** to store Tree **a1**
- 2) Constructor of class Tree is called to initialize **a1** contents

Program execution is leaving the variable **a1** scope:

- 1) **a1** is a Tree object: class Tree destructor is called on **a1**
- 2) System releases memory allocated for object **a1** in the stack

a2 lifetime

- 1) System allocates memory on the **stack** to store **a2**, a pointer to a Tree
- 2) System allocates memory in the **heap** to store a Tree
- 3) Constructor of class Tree is called to initialize this Tree contents
- 4) The address of this Tree object is returned and stored in variable **a2**

You request to destroy the tree **a2** points to:

- 1) Class Tree destructor is called on this object
- 2) System releases memory allocated for this Tree object in the **heap**

Program execution is leaving the variable **a2** scope:

- 1) **a2** is a pointer (a number): no destructor.
- 2) System releases memory allocated for the pointer **a2** in the **stack**



Building a binary executable (1/3)



- C++ files organization
 - Header files (« .h ») :
 - **Declaration** : types, constants, functions, classes (attributes and signature methods)
 - Files to be included when the declared entities are needed.
 - Implementation files (« .cpp ») :
 - **Definition** : global variables, functions contents, class methods code
 - Files to be compiled to produce intermediate binaries and finally the binary executable.
- Only one entry point for one program
 - The « main » function

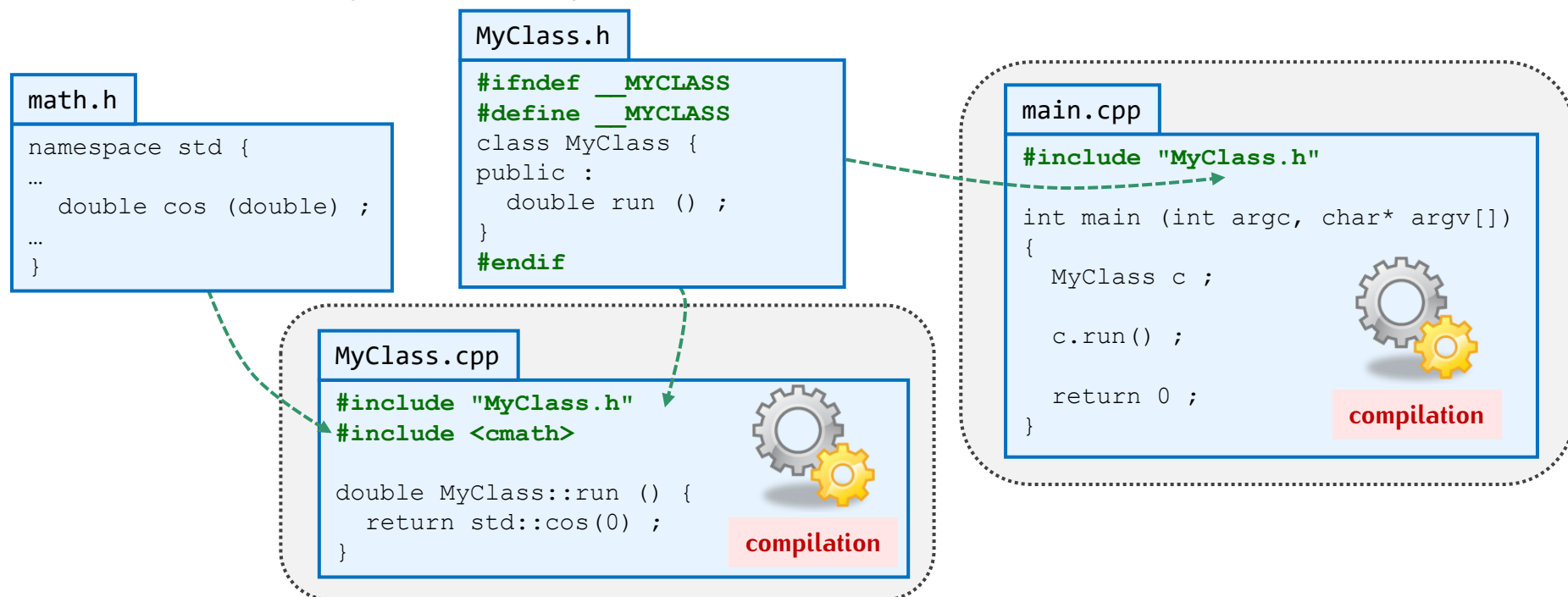
```
int main (int argc , char* argv[]) {  
    ... // Program ...  
    return 0 ;  
}
```



Building a binary executable (2/3)

- Using C++ preprocessor

- Including file class declaration (.h): `#include`
- Preventing one class declaration from multiple inclusion: `#define`, `#ifndef`, `#endif`, ...



Building a binary executable (3/3)



MyClass.h

```
class MyClass {
public :
    double run () ;
}
```

MyClass.cpp

```
#include "MyClass.h"
#include <cmath>

double MyClass::run () {
    return std::cos(0) ;
}
```

main.cpp

```
#include "MyClass.h"

int main (int argc, char* argv[])
{
    MyClass c ;

    c.run() ;

    return 0 ;
}
```

math.h

```
namespace std {
...
    double cos (double) ;
...
}
```

libm.so

```
01010110111
10100010010
01010000101
```

1

Compiling :

g++ -c MyClass.cpp

MyClass.o

```
01010110111
10100010010
01010000101
```

2

Compiling :

g++ -c main.cpp

main.o

```
0101011
1010001
0101000
```

3

Linking :

g++ -o my_prog main.o MyClass.o -lm



my_prog

```
01010110111
10100010010
01010000101
```

-std=c++11 -Wall



Practice



- **Building your first C++ program with class ☺**



- Compiling, linking, running,
- “Make” the process automatic ...

- **Managing objects lifetime**



- Understanding constructor / destructor calling time
- Static vs. dynamic allocation

- **Bonus : “Inside memory”!**



- Displaying addresses
- Analyzing object memory footprints

