A <Basic> C++ Course

4 – Rappels++

Julien Deantoni

Structure of programs

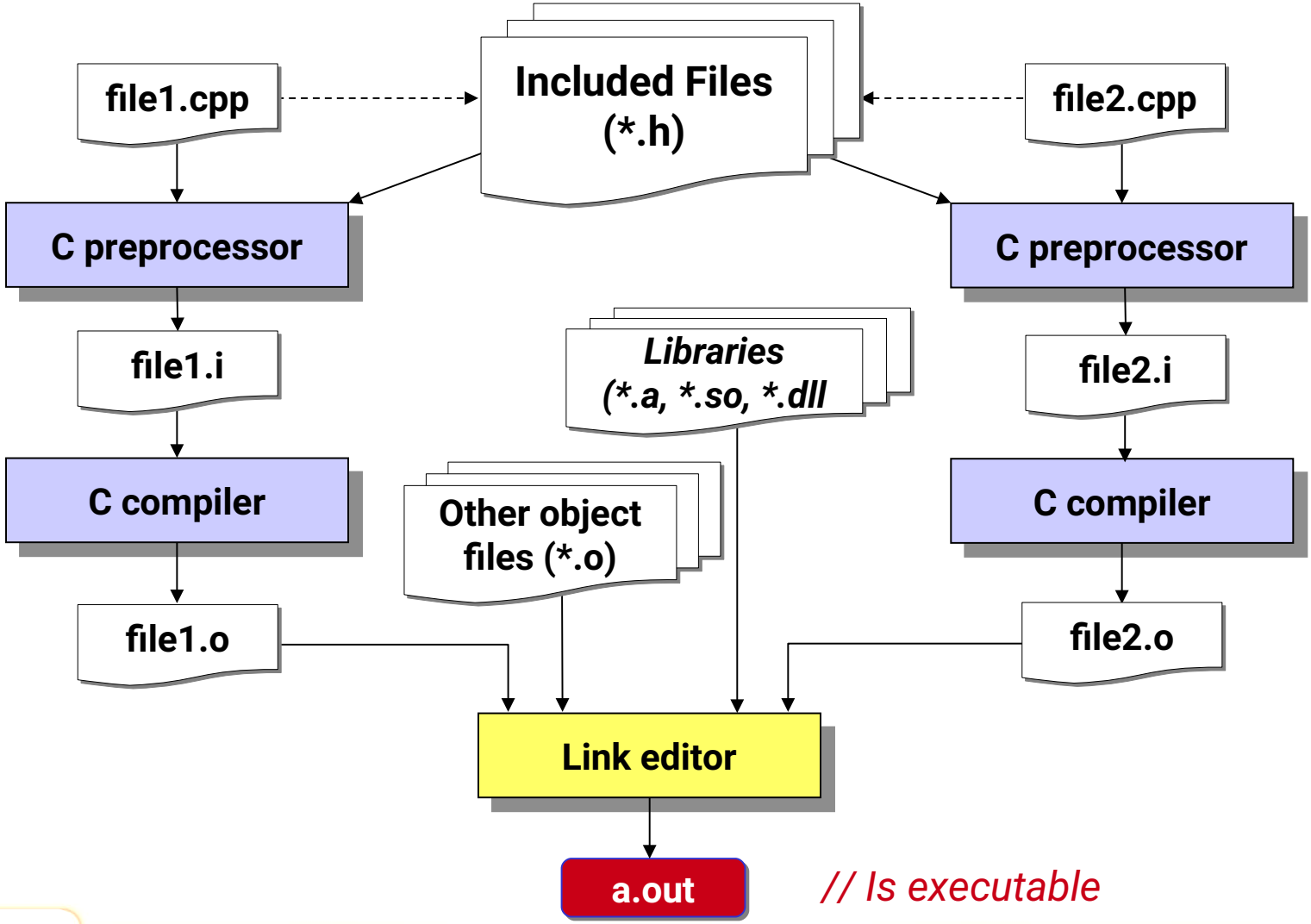
Header file (.h or .hpp)

- Specification of a module
- Not a compilation unit: included in other source files
  - global variables declaration
  - constant and static (file scope) variable and function definitions
  - inline function definition
  - class definitions
  - free functions declarations
  - template declarations **AND definitions**

Non header (.cpp)

- Implementation of a module
- Compiled separately
  - global variable definitions
  - function definitions

compiler le C++



compiler *une classe*

```
# sketchy makefile example
EXE_NAME=executable
LINK_CXX=g++
COMPIL_CXX=g++ -c

example: main.o rectangle.o
	$(LINK_CXX) main.o rectangle.o -o $(EXE_NAME)
main.o: main.cpp
	$(COMPIL_CXX)  main.cpp
rectangle.o: rectangle.cpp rectangle.h
	$(COMPIL_CXX) rectangle.cpp
```

Makefile

compiler *une classe*

```
# sketchy makefile example
FLAGS =-g -ansi -Wall -Wextra -Wold-style-cast -Woverloaded-virtual ...
EXE_NAME=executable
LINK_CXX=g++
COMPIL_CXX=g++ -c

example: main.o rectangle.o
	$(LINK_CXX) $(FLAGS) main.o rectangle.o -o $(EXE_NAME)
main.o: main.cpp
	$(COMPIL_CXX)  $(FLAGS) main.cpp
rectangle.o: rectangle.cpp rectangle.h
	$(COMPIL_CXX) $(FLAGS) rectangle.cpp
```

Makefile

La cible “clean” qui détruit les .o et l'executable s'avère pratique

RAJOUTER TOUS LES WARNINGS !!!

Miscellaneous

- Objects may be declared anywhere in a block (before it is used)

```
void f(int x)
{
   ++x;
   int y = x; // OK in C++
  ++y;        // not in C
}
```

- Constant integers can be used as array dimensions

```
const int N = 10;
double t[N];// OK in C++,
    // not in ANSI C
```

- Boolean type
  - **`bool`** : values **`true`** / **`false`**
- Generic pointers (`void *`)
  - A **`void *`** may be assigned a pointer value of any pointer type

```
Person *pp = &me;
void *pv = pp;
```

  - The converse is true in ANSI C, false in C++!

```
pp = pv;// OK in ANSI C
pp = (Person *)pv;
  // in C++
```

- Null pointer is nullptr !
  - No more 0 (or NULL)

User defined types

- Automatic typedef
  - The structure, union, or enum tag may be used directly as the type name, as if it was typedef'ed

```
enum Color {BLUE, RED, GREEN};
struct Person {
     string name;
     short age;
};
union RealInt {int i; double x;};
Color c = BLUE;
Person p;
RealInt ri;
```

  - The old ANSI C form can still be used to solve ambiguities

Dynamic memory allocation

- Typed allocation of a memory area

```
int *pi = new int;
Person *p1 = new Person[1000];
int *pj = new int(10);
Person *p2 = new Person[*pj];
```

- Returning the area to the free store

```
delete pi;
delete[] p1;
delete[] p2;
```

  - The operand value must be the result of a (previous) new
  - **`delete nullptr`** is harmless

Instructions

- No new form of instructions, compared to ANSI C
- Identical control structures
- **A difference**: local scope of loop variable

```
for (int i = 0; i < N; i++) {
   if (t[i] == 0) break;
}
// here i is unknown
```

```
int i;
  for (i = 0; i < N; i++) {
       if (t[i] == 0) break;
  }
  // here i is known
```

Free functions and methods (1)

- Method
  - A function member of a class, which can be called only through a class instance
  - Everything is as if each instance own a copy of the function
- Free function
  - A function which does not need an object to be called
  - Can be a global function or a static class member

Free functions and methods (2)

```
void f(int, int);

class A {
public:
  void m();
  static void st();
  ...
};
```

```
#include <cmath>

int main() {

  f(3, 5);
  A::st();
  double x;
  x = std::cos(3.14);

  A a;
  a.m();
}
```

Prototype

- Function without parameters

```
int f();          // in C++, not ANSI C
```

- Function with unchecked parameters

```
int f(...);     // in C++ (and ANSI C)
int printf(const char *, ...);
```

- Arguments with default value (C++ only)

```
void Error(char * = 0);
int f(int i, double x = 0.0, int j = 1);
```

  - Only a terminal sub-list of the arguments
  - Beware: do not repeat the default values in the function definition

Inline functions

- The compiler textually expands the body
  - Save a function call, but increase generated code
  - Replace macros with parameters (**`#define`**)
  - A simple hint for the compiler
  - Inline functions are static
    - The body of an inline function must be in the same compilation unit as the function calls and before them

```
inline int nop(int i) {return i;}
inline double middle(double x, double y)
{
   return (x + y)/2;
}
```

Parameter passing: C style

- Default is passing by value
  - Copy of effective parameter value into a local variable
- Explicitly passing a pointer
  - Allow to modify the effective argument within the function body

```
void incr2(int *px) {*px += 2;}
// ...
int i = 2;
incr2(&i);
```

  - Or simple convenience to avoid large memory copy

```
void print_Person(const Person *p);
Person me;
print_Person(&me);
```

Parameter passing: by reference

- Only the signature of the function specifies that an argument is to be passed by address
  - Allow to modify the effective argument within the function body

```
void incr2(int& x) {x += 2;}
// ...
int i = 2;
incr2(i);
```

  - Or simple convenience to avoid large memory copy

```
void print_Person(const Person& p);
Person me;
print_Person(me);
```

References as aliases

- A reference is in fact another name for an object

```
int i = 5,j;
int& ri = i;
const int& rci = i;
// ...
ri = 10;              // i == 10 too
j = rci;              // j == 10
j = rci + ri;         // j == 20
rci++;                // NO: not an lvalue

Symbol tab[100];
Symbol& first_tab = tab[0];
Symbol& last_tab = tab[99];
```

References vs. pointers

- References and pointers
  - A reference must be initialized
  - There is nothing such as a null reference
  - There is nothing such as a reference to a function
  - References cannot be assigned to (the referenced objects are)

- Address of a reference

```
int i;
int& ri = i;
int *pi = &ri; // pi == &i
```

Functions returning an address

- Return type may be a pointer or a reference

```
int* f1(int);
int& f2(int);
```

  - **`*f1(i)`** and **`f2(i)`** are *lvalue*s

```
*f1(10) = 14;
f2(i) = j;
```

  - The function must return the address of/a reference to an object which will still exist after leaving the function body

```
int& f2(int i)
{
    int tmp = i + 3; // temporary
    return tmp;      // ERROR
}
```

The C(++) preprocessor

- ANSI C preprocessor only
- Less crucial in C++ than in C
- Features useful in C++
  - Inclusion of source file (header files)

```
#include <fic.h>
#include "fic1.h"
```

  - Conditional compilation (**`#if`**, **`#ifdef`**, ...)
  - "Stringification" (operator #)
  - Macro-definition (**`#define`**)
    - most often replaced by **`inline`** and **`const`**

Function overloading

- Several functions with the same name but different bodies, distinguished by their signature
  - In fact only by the arguments part of the signature

    In general the return type does not play any role

```
void Error(const char *, double);
void Error(const char *, int);
void Error(const char *, const Person&);
```

Function overloading

- Several functions with the same name but different bodies, distinguished by their signature
  - In fact only by the arguments part of the signature

    In general the return type does not play any role

```
void Error(const char *, double);
void Error(const char *, int);
void Error(const char *, const Person&);
```

- Beware: ambiguities are possible

```
void Error(const char *);
void Error(const char *, char = 'a');
int Error(const char *);

Error("bonjour");
```

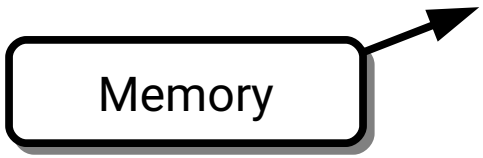
3 possible choices...

Memory concerns...

- Les variables / objets sont stockés en mémoire

```
main(){

}
```



Memory concerns...

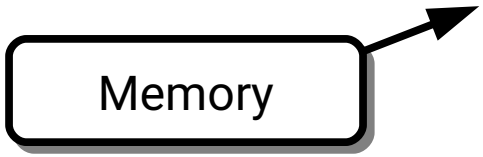
- Les variables / objets sont stockés en mémoire

```
main(){
 int i;

}
```

i:int

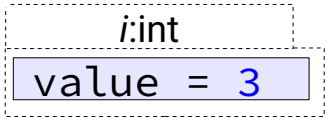
value = ??



Memory concerns...

- Les variables / objets sont stockés en mémoire

```
main(){
 int i;
 i=3;
}
```



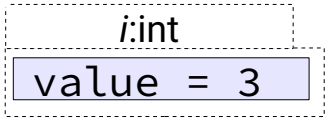
Memory concerns...

passage par copie / valeur

- Spécifié dans la déclaration & la définition d'une fonction

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
}
```



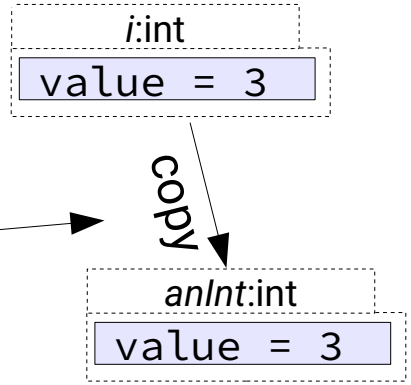
Memory concerns...

passage par copie / valeur

- Spécifié dans la déclaration & la définition d'une fonction

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



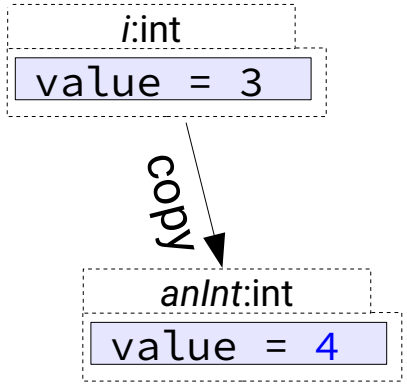
Memory concerns...

passage par copie / valeur

- Spécifié dans la déclaration & la définition d'une fonction

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



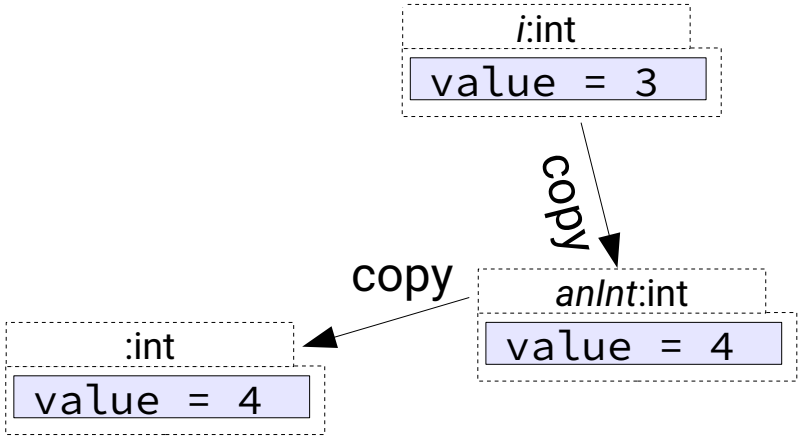
Memory concerns...

passage par copie / valeur

- Spécifié dans la déclaration & la définition d'une fonction

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



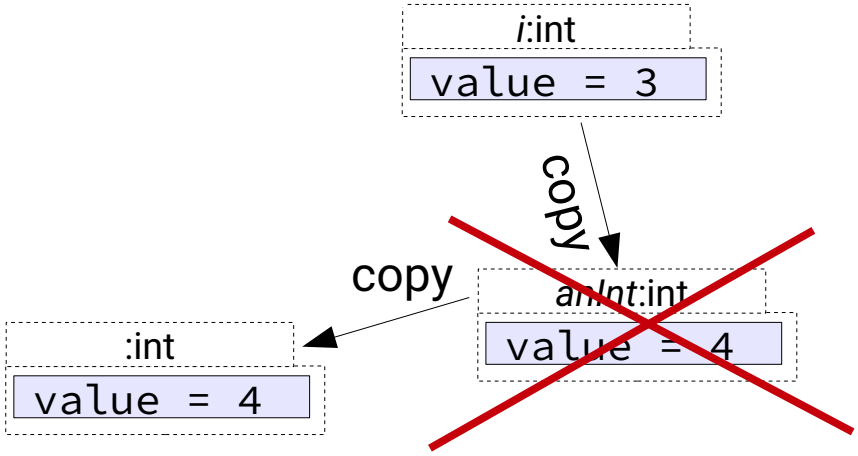
Memory concerns...

passage par copie / valeur

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



Les variables/objets statiques
sont détruits à la fin
du bloc de déclaration

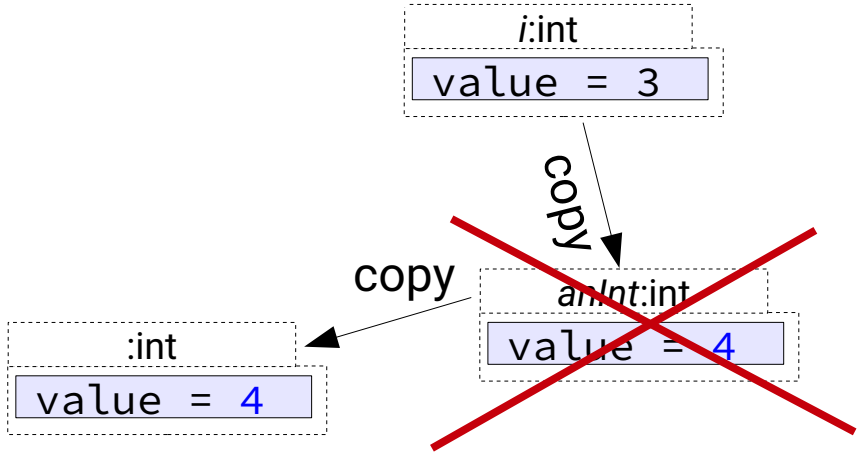
Memory concerns...

passage par copie / valeur

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



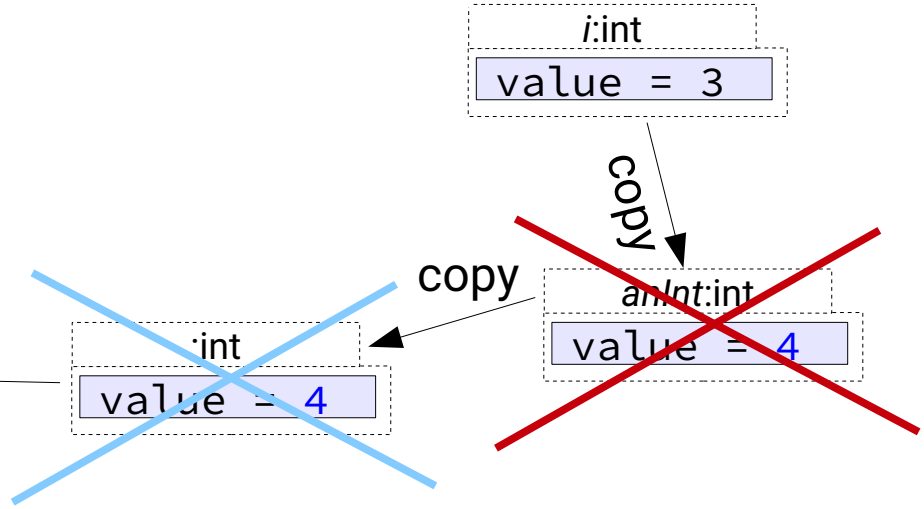
Memory concerns...

passage par copie / valeur

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```



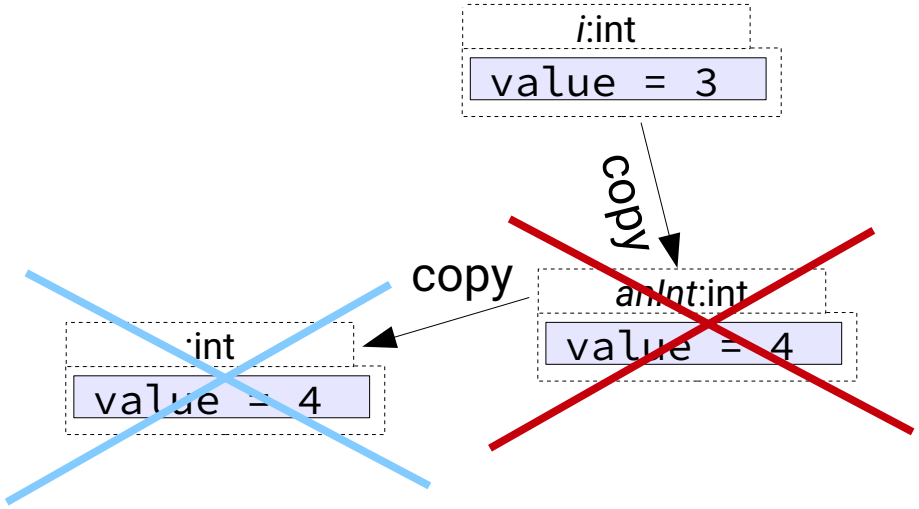
Memory concerns...

passage par copie / valeur

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
 std::cout<< i <<std::endl;
}
```




```
jdeanton@FARCI:$./executable
3
jdeanton@FARCI:$
```

Memory concerns...

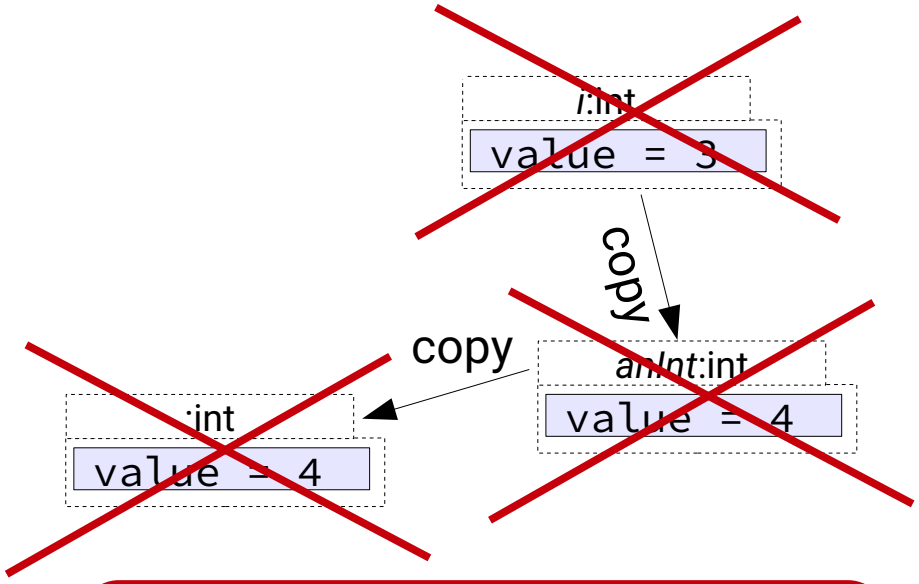
passage par copie / valeur

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(int anInt){
  anInt = anInt + 1;
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
 std::cout<< i <<std::endl;
}
```

```
jdeanton@FARCI:$./executable
3
jdeanton@FARCI:$
```



Les variables/objets statiques
sont détruits à la fin
du bloc de déclaration

Memory concerns...

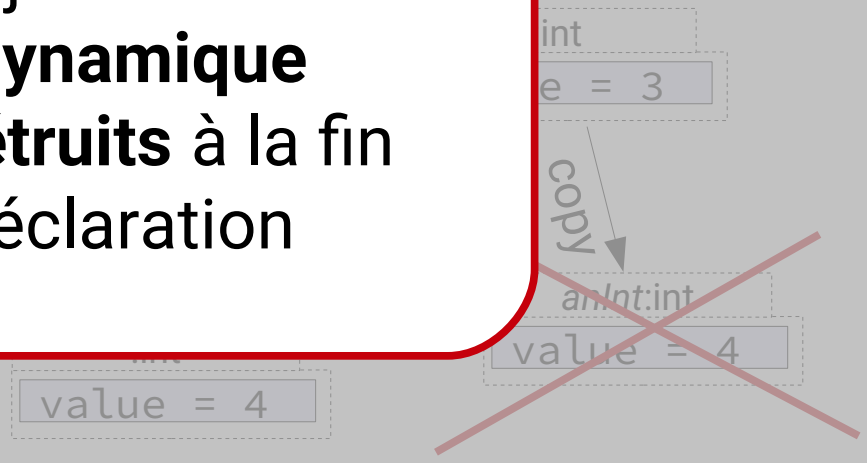
en dynamique...

- `anInt` est local à la fonction et détruit à la fin

```
int incremente(
  anInt = anInt
  return anInt;
}


main(){
 int i;
 i=3;
 incremente(i);
}
```

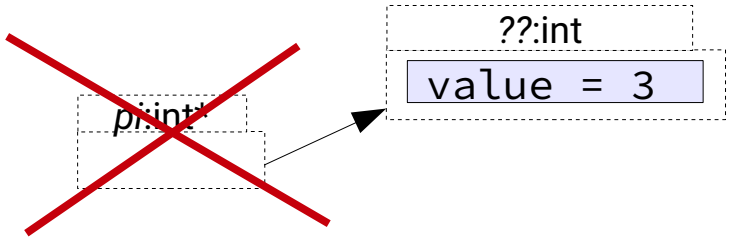
Les variables/objets reservés de manière **dynamique** **ne sont PAS détruits** à la fin du bloc de déclaration



Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```

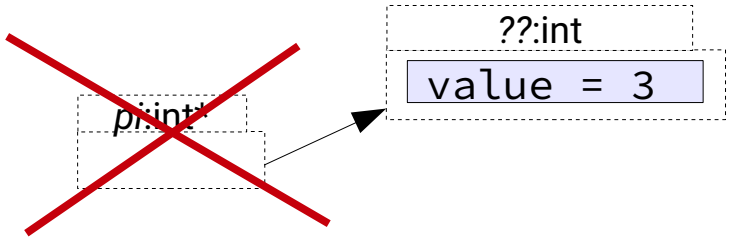


Les variables/objets statiques
sont détruits à la fin
du bloc de déclaration

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```



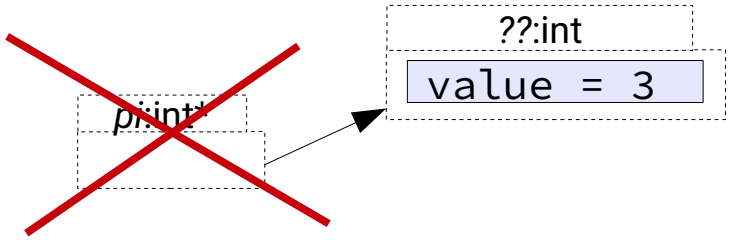
Mais PAS les objets reservés de manière dynamique
→ Fuite mémoire !!

Les variables/objets statiques sont détruits à la fin du bloc de déclaration

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**

```
main(){
 int* pi = new int(3);
}
```



Mais PAS les objets reservés de manière dynamique
→ Fuite mémoire !!

Les variables/objets statiques sont détruits à la fin du bloc de déclaration

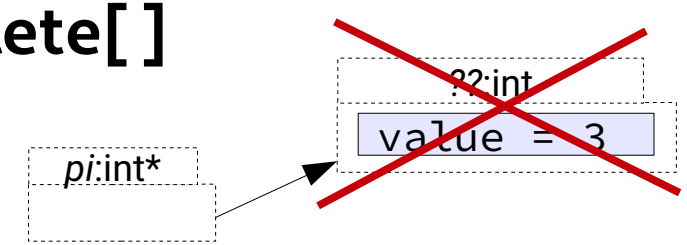
Les objets restent/s'accumulent en mémoire
jusqu'au prochain redémarrage

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
  int* pi = new int(3);
  delete pi;
}
```



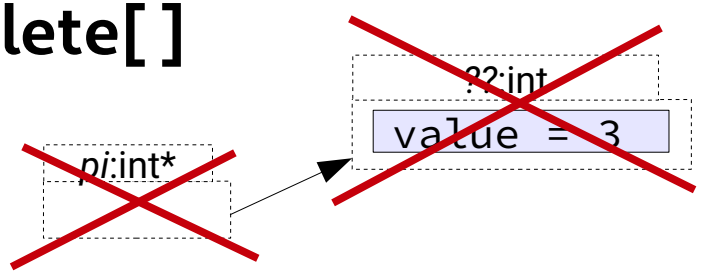
- Il faut que l'opérande de `delete` soit le résultat d'un (précédent) `new`
- `delete 0` n'est pas dangereux

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
}
```



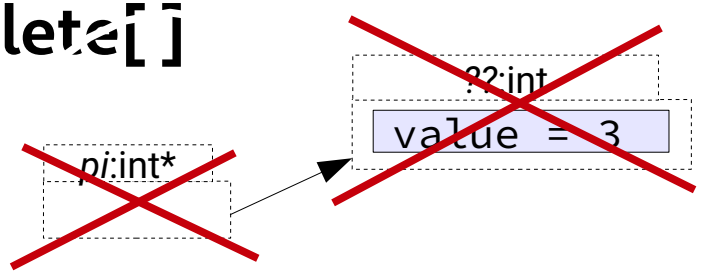
- Il faut que l'opérande de `delete` soit le résultat d'un (précédent) `new`
- `delete 0` n'est pas dangereux

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
  int* pi = new int(3);
  delete pi;
}
```



Attention `pi` peut exister sans l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
  int* pi = new int(3);
  delete pi;
  cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
A votre avis ??????
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
  int* pi = new int(3);
  delete pi;
  cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
0
jdeanton@FARCI:$
```

Attention `pi` peut exister sans
l'objet pointé...

Memory concerns...

- Reservation de mémoire :
  - Operateur **new**
- Destruction de mémoire :
  - Opérateur **delete** et **delete[ ]**

Il faut détruire explicitement
la mémoire allouée explicitement

```
main(){
   int* pi = new int(3);
   delete pi;
   pi=nullptr;
   cout << (*pi) << endl;
}
```

```
jdeanton@FARCI:$./executable
Segmentation fault
jdeanton@FARCI:$
```

Attention `pi` peut exister sans l'objet pointé...

→ bonne pratique: pointeur a NULL

Memory concerns...

résumé des notations

- Seule la définition et la déclaration d'une fontion porte la marque de l'utilisation du passage par référence