Langages\_II\_synthese

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Langage | Style | Data & function | Function | | Access control |
| Datatype supported | Code executed |
| C | tradition | separated | 1  Execute code | same | Anything used anywhere |
| C++ | OOP | Attributes + methods  encapsulation in class as members | 1 or more  Execute after | Different according to executor object | v |

|  |  |  |  |
| --- | --- | --- | --- |
| Variable |  | Example | lifetime |
| Global | Static :  May be used with méthodes/fonctions/classes  dépend d’aucune instance et peuvent être perçue comme des variables de namespace | struct Person{  string nom, prenom;  **static** int count;  };  **int Person::count=0 // count is global: only one initialization out of the class** | Declaration to program end.  If anormal program end:  variable not destoyed correctly |
|  |  |
| Local |  | Function parameter | Calling bloc appelant delimited by ‘{ }’ |
| Instance |  |  | Associated to a class et sa durée de vie est déterminée par  la durée de vie d’un objet |
| Temporary |  |  | Crée par appel à un constructeur ou un appel de fonction (via le return-by-value).  Détruit à la fin de l’expression |
| Dynamic |  | T\* p= new T; //crée une nouvelle instance de T, appelle le constructeur  delete p; // appelle le destructeur de l'objet pointé par p | From « new » to « delete » |

|  |  |  |
| --- | --- | --- |
| Code | Constructor used | Else |
| C x | default |  |
| C x{} |  | Initializer list’s |
| C x{1,2} | Initializer list’s | normal |
| C x (1,2) | normal |  |
| C x () | Fonction x déclarée |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | From | get value | Security | Liberty |
| Pointer | Adresses mémoire de taille fixe, de type variable.  Un pointeur vide contient nullptr  int x=57; //variable int  int \*p=&x; // pointeur vers un entier  \*p = 42; // déréférence le pointeur, change la valeur  cout << x;  42 | une variable ( &var )  • d’un autre pointeur  • de new T qui créé une nouvelle instance d’un objet T et qui retourne un pointeur vers ce dernier (allocation dynamique) | \*p | Dangerous :  • conversion possible de pointeur via typecast sans protection  • les données sont peut-être plus présente  • arithmétique des pointeurs (changer l’adresse, etc) |  |
|  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Pointer | | Definition | | Declaration - initialisation | Access |  | Security |
| Adress | Value |  |
| Normal | | Adresses mémoire de taille fixe, de type variable.  Un pointeur vide contient nullptr | | *valuetype*\* name | int \*\*pp = &p; | \* name | Dangerous :  • conversion possible de pointeur via typecast sans protection  • maybe no value  • arithmétique des pointeurs (changer l’adresse, etc) |
| **Smart**  *If object is destroyed : memory is freed* | Unique | stores one pointer only | | unique\_ptr<*T*> *ptrName*(new *pointedObjectInitializer*) |  |  |
|  | Shared | more than one pointer can point to 1 object | Reference Counter | shared\_ptr<*T*> *ptrName*(new *pointedObjectInitializer*) |  |  |
|  | Weak |  | No Reference Counter | weak\_ptr<*T*> *ptrName*(new *pointedObjectInitializer*) |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Référence | Équivaut à un pointeur dont on ne peut modifier la valeur, on accède juste à ce qui est pointé. | int x=42;  int \*p=&x; //pointeur  int &r=x; //référence | r | better lisibility  less error | cant modifiy referenced value |

Smart pointer pseudo code :

template <typename T>

class SmartP{

smartP(); //constructeur qui le dfini en nullptr

explicit smartP(T \*p); // initialise avec un aure pointeur

//deplacement

smartP(smartP<T> && other);

smartP operator=(smartP<T> && other);

smartP(smartP<T> & other)=delete; //copie interdite

smartP operator=(smartP<T> & other)=delete; //assignation interdite

~smartP(); //destructeur

//opérateur

T &operator \*(); //déréférencement

T \*operator ->(); //déréférencement puis access element

// comparaison avec nullptr

friend bool operator ==<T>(nullptr\_t, const smart<T>&);

friend bool operator ==<T>(const smartP<T>, nullptr\_t);

}

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| *Std::* List  Name | Elements | | | | | | | | Size | Declare | Init | Append | complexity | |
|  | Objects Types | | Possible Storage other than references | |  | memory locations | Access | |  |  |  |  | Time | Space |
| Array | **Same** any | | Primitive data types |  | | -Contiguous  -individually referenced | *name[index]* |  | *fix* | *elementsType* *name* [*size*] | *elementsType* *name* [*size*] = *{value1, value2,..}* | NONE |  |  |
| Vector | any | |  | *name*.**at**(*index*)  out-of-range exception | dynamic | vector<*type*> *name* | vector<*type*> *name*(*size,value*)  vector<*type*> *name{value1, value2,..}* | *name*.push\_back(*value*) |  |  |
| Tuple | Primitive data types |  | |  | get<*index*>(*name*)  *Name.*first  *Name.*second | | *fix* | tuple <type1,type2> | make\_tuple(value1, value2) | NONE |  |  |
| Pair |  | |  | 2 | pair <type1,type2> | make\_pair(value1, value2) | NONE |  |  |
| Enum |  | |  |  | |  |  | |  |  | enum *type* { *value1, value2,..}*} | NONE |  |  |
|  |  |  |  |  | |  |  | |  |  |  |  |  |  |
|  |  | |  |  | |  |  |  |  |  |  |
|  |  |  | |  |  | |  |  |  |  |  |  |
|  |  | |  |  | |  |  | |  |  |  |  |  |  |
|  |  | |  |  | |  |  | |  |  |  |  |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Stl::  Name | Elements | | | | | | Size | Declare | Init | Append | complexity | |
|  | Objects Types | | Possible Storage other than references | memory locations | Access | |  |  |  |  | Time | Space |
| set | keys | unique | none |  |  | |  |  |  |  | O(log n) |  |
| Unordered set |  |  | |  |  |  |  | O(1) |  |
| Multiset | Unique/duplicate |  |  | |  |  |  |  |  |  |
| Unordered\_map |  | |  |  |  |  |  |  |  |  |  |  |
| Initializer\_list |  | |  |  |  |  |  |  |  |  |  |  |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Passage de paramètre | |  |  | Copie | Modification de  l’objet original |
| Value | | Le plus basique, fait une copie de la variable/l’objet en paramètre dans x | void f(string x); | 1 | 0  Only the copy |
| référence |  | x est le même objet/variable que l’objet/la variable placé(e) en paramètre | void f(string &x); | 0 | yes |
| constante | Combine l’idée du passage par référence (sans copie) où l’objet en paramètre est le  même que x mais ne peut être modifié au sein de la fonction | void f(const string &x); | 0 | 0 |
| R-value | peut être vu comme un objet temporaire où la durée de vie ne dépassera  pas l’expression | void f(const string &&x);  string x='hello';  string y=' world';  f(x+y); // ici le 'hello world' est temporaire | 0 | 0 |
| Pointer | | Passage par valeur d’un pointeur, plus utile pour les grosse données | void f(string \*x); | 0 | yes |

|  |  |  |  |
| --- | --- | --- | --- |
| **Sr. No.** | **Key** | **Inheritance** | **Composition** |
| 1 | Relationship | "is-a" | "has-a" |
| 2 | Scope | provides its features at compile time | easily achieved at runtime |
| 4 | Final | can’t reuse code from the final class | allows code reuse even from final classes |
| 5 | Methods | exposes public and protected method of the parent class | doesn’t expose  interact using public interface. |
|  |  | class Parent0  {  public:  virtual void print(){...};  }  class Parent1 public virtual Parent0  {  public:  virtual void print(){...};  }  class Parent2 public virtual Parent0  {  public:  virtual void print(){...};  }  class Child:public virtual Parent1, public virtual Parent2  {  public:  } | class Source1  {  public:  }  class Source2  {  public:  }  class Composition  {  public:  Source1 instance\_source1;  Source2 instance\_source2;  } |
|  |  | Diamond problem resolved using  1)“virtual”: permet d’appeler une fonction du type de  l’instance plutôt que du type de l’objet pointé  2) objet\_child.Parent1::print() ou  objet\_ child.Parent2::print() |  |

|  |  |
| --- | --- |
| Constructor |  |
| default | C x |
| Default  else : initialisation list | C x{} |
| initialisation list  else : normal | C x{1,2} |
| normal | C x(1,2) |

|  |  |  |
| --- | --- | --- |
| Dynamically allocated memory |  |  |
| Leak |  | int res=0;  int\* p;  int bad1(); // construit 1000 ints  for (int i=0; i<1000; i++){  p=new int{i+i};  res+=p  }  delete p // détruit que un seul  return res |
| Non allocated |  |  |
| Dangling pointer |  | {  char \*dp = NULL;  /\* ... \*/  {  char c;  dp = &c;  }  /\* c falls out of scope \*/  /\* dp is now a dangling pointer \*/  } |

|  |  |  |
| --- | --- | --- |
| Constructeur de copie | Opérateur d’assignation | |
| Stack::Stack(Stack &other){ | Stack::Stack &operator=(Stack other){  while (! isEmpty()) pop(); //se vide | |
| Stack tmp; //stack temporaire  while (!other.isEmpty()){  // push tout dans le stack temp  tmp.push(other.pop());  }  while (!tmp.isEmpty()){  //repush tout dans \*this et other  auto s=tmp.pop();  push(s);  other.push(s);  } | | |
| } | | return \*this; //&operateur a besoin d'une référence en retour  } |

|  |  |
| --- | --- |
| Friend |  |
| Class | class Node {  private:  int key;  Node\* next;  /\* Other members of Node Class \*/  friend class LinkedList; // Now class LinkedList can access private members of Node |
| Function | friend int LinkedList::search(); // Only search() of linkedList can access internal members  }; |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| opérateurs non surchargeable | . | ?: | :: | ;\* | sizeof() |

|  |
| --- |
| Conversion operator overload |
| class Fraction{  ...  //au sein de la classe (déclaration de la méthode)  operator float(); // retourne un float qui représente la fonction  ...  };  //en dehors  Fraction::operator float(){  // comme n et d sont des entier on force la conversion vers float  return static\_cast<float>(n)/static\_cast<float>(d)  } |

|  |  |  |  |
| --- | --- | --- | --- |
| Name | | Constant | int i=5, j=7; |
| Const | Variable | Value | const int k |
| Method | Itself  Its class attributes | void printFr(const Fraction &f){  cout << f.toString() << endl; //erreur  }  void const printFr(const Fraction &f){  cout << f.toString() << endl; //ok  } |
| Pointer | | int const \*p=&j; //pointeur constant vers un entier  \*p=7  p=&i // erreur |
| Pointed value | | int \*const p=&j; //pointeur vers un entier constant  \*p=7 // erreur  p=&i |
| Pointer & Pointed value | | const int \*const p=&j; //pointeur vers un entier constant  \*p=7 // erreur  p=&i // erreur |
| Constexpr | Variable | | value known at compilation |
| function/method | | compilator can determine function / method return value only if constexpr parameters |





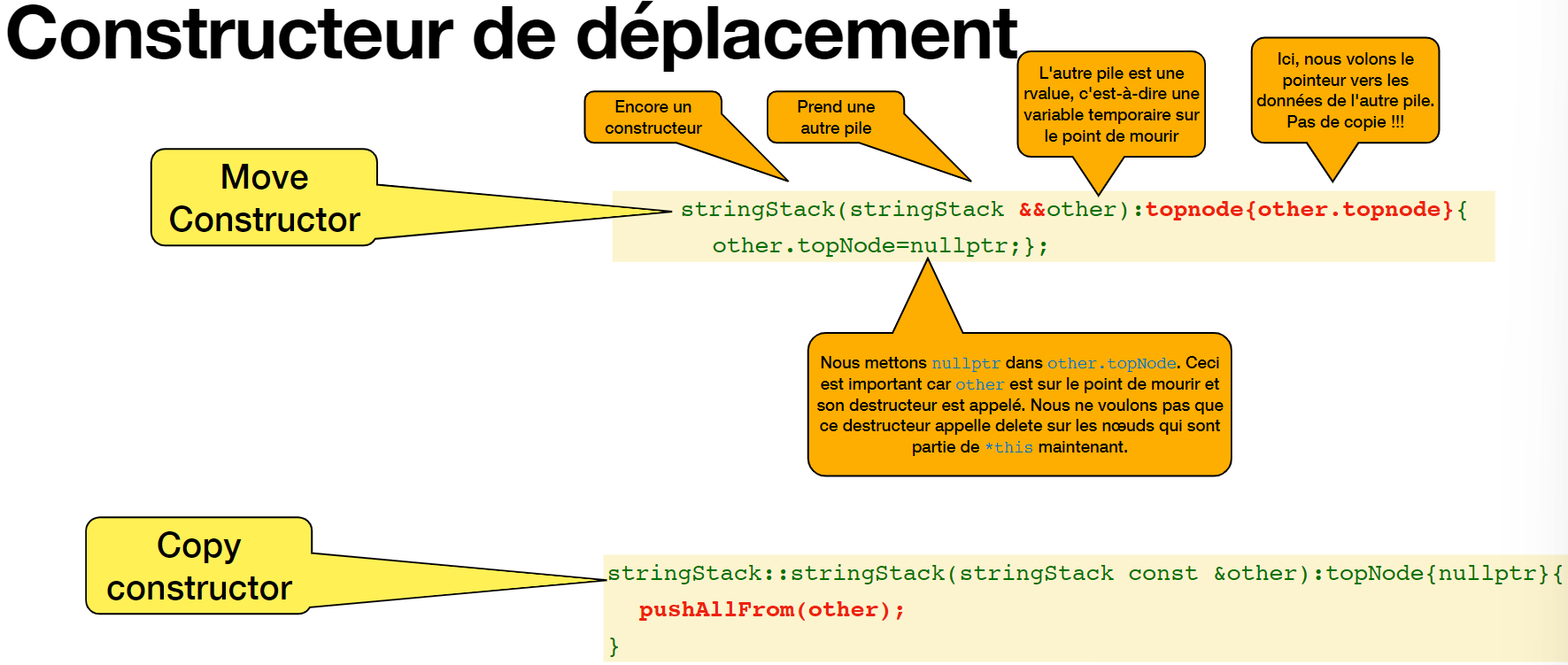
Une image contenant texte

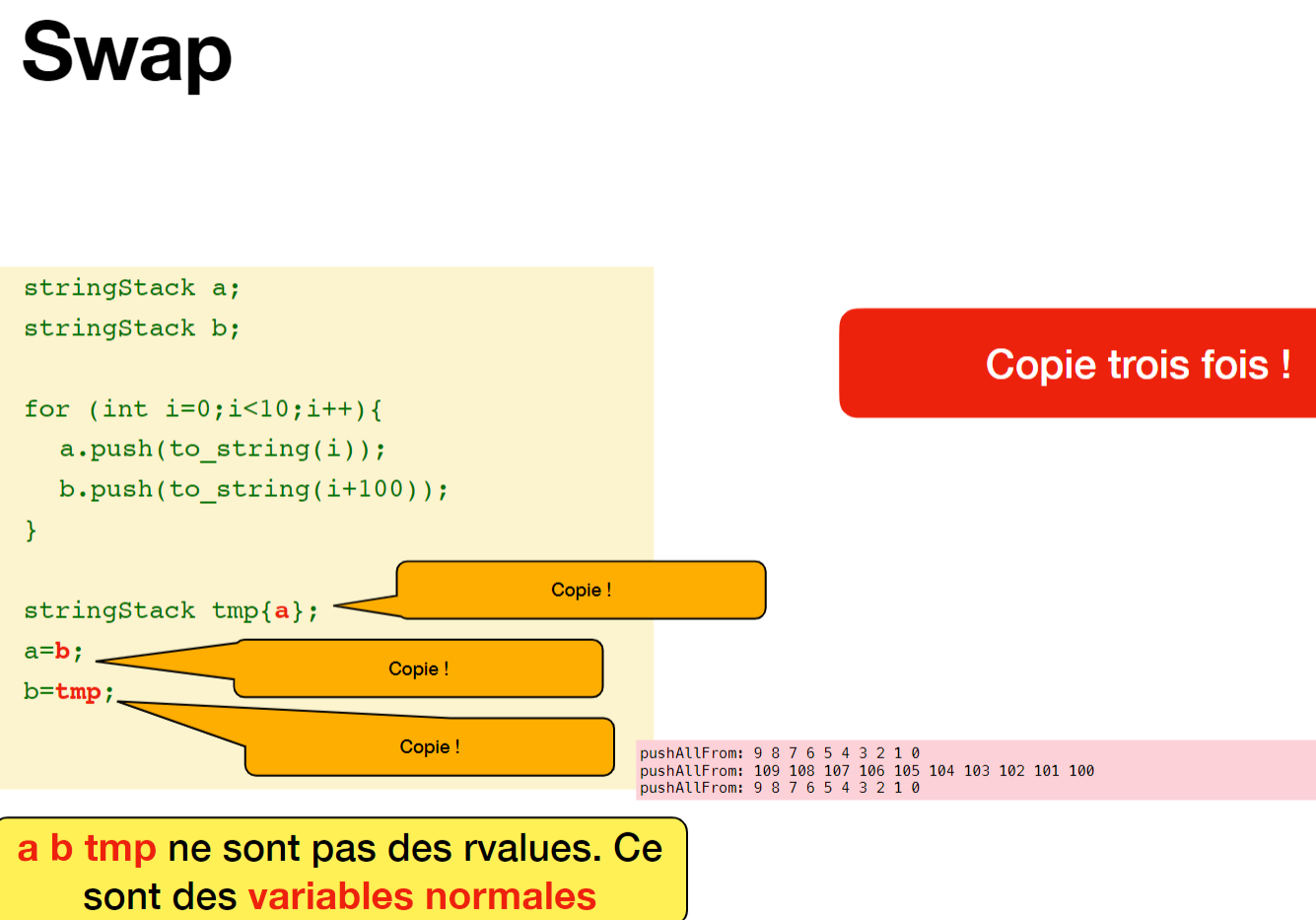
Description générée automatiquement

Une image contenant texte

Description générée automatiquement

|  |  |
| --- | --- |
| Copy = Other not changed | Move = Déplacement d’assignation |





|  |  |  |  |
| --- | --- | --- | --- |
| class using dynamic memory | Constructor | Copy | classname(const &classname) |
| Operator | assignation | operator=(const &classname) |
| If rvalue parameter | Constructor | Move | classname(const &&classname) |
| Operator | assignation move | operator=(const &&classname) |

Une image contenant texte

Description générée automatiquement

Template

Une image contenant texte

Description générée automatiquement

Une image contenant texte

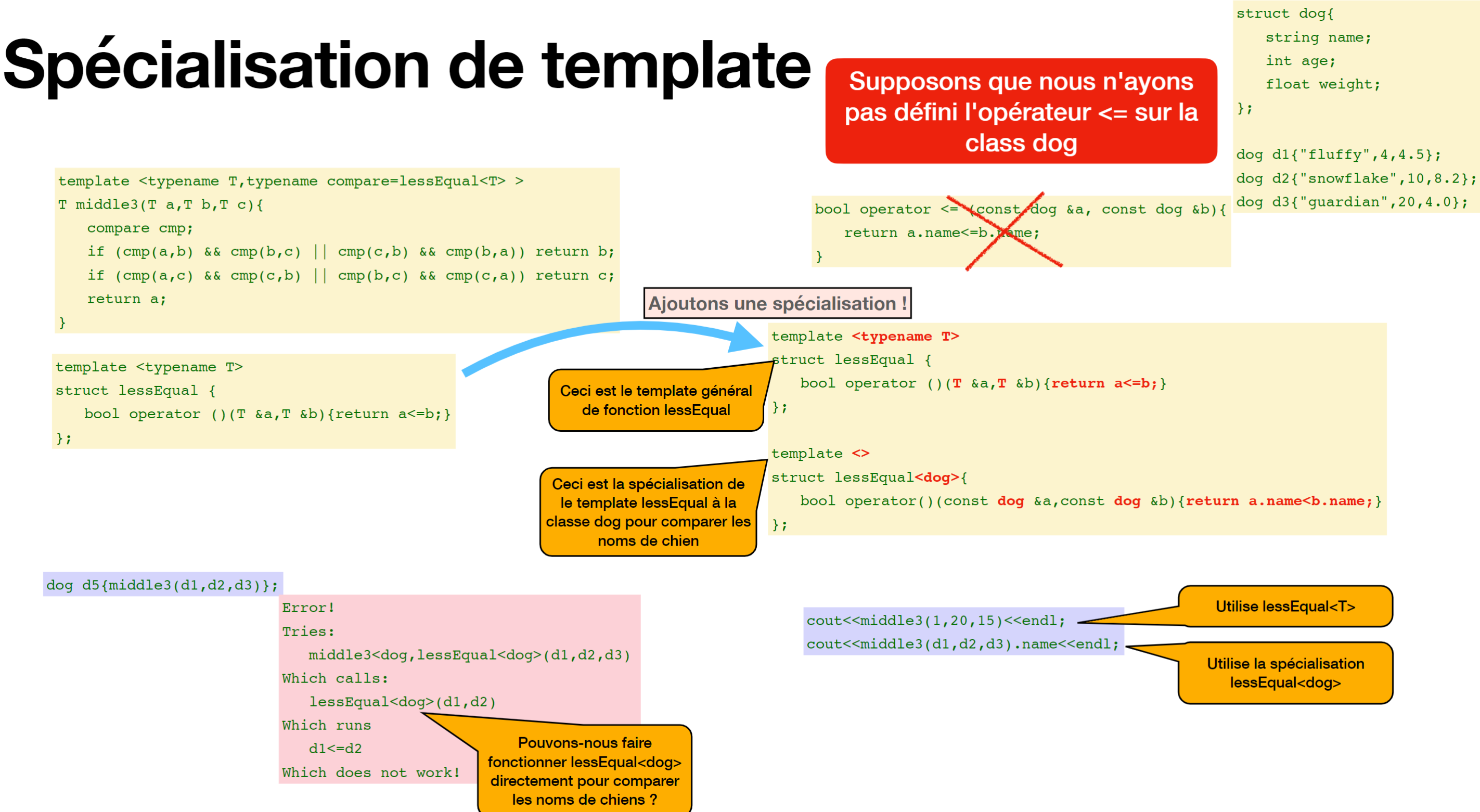
Description générée automatiquement

Class without data

Overloads only operator ()

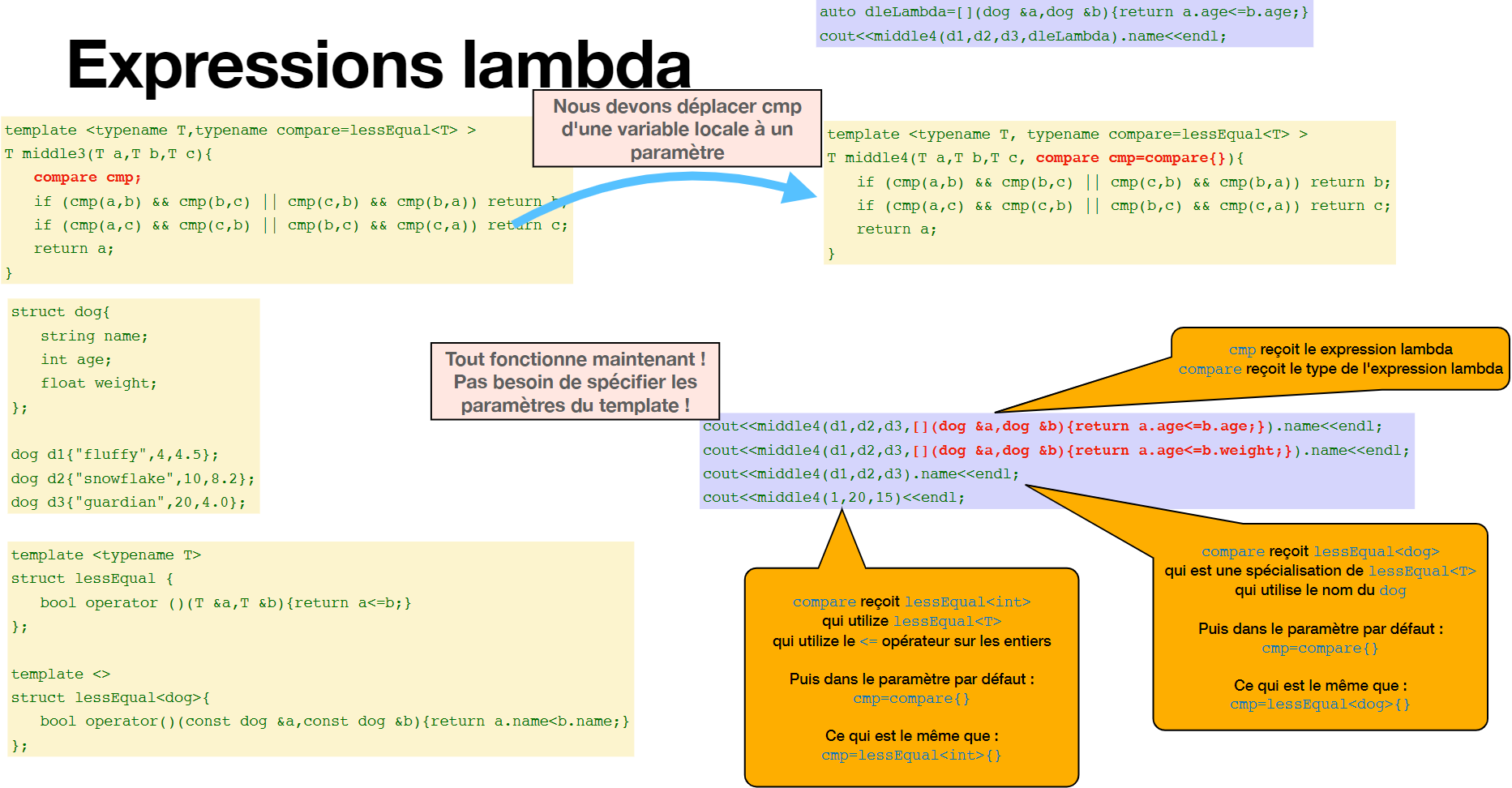
|  |  |  |
| --- | --- | --- |
|  | On ajoute donc une surchage d’opérateur à la classe Dog | Add functor &  un second paramètre, agissant comme une fonction,  au template |
| struct Dog{  int age;  string name;  float weight;  };  template <typename T>  T middle(T a, T b, T c){  // retourne l'élément du "millieu"  if (a<=b && b<=c ||c<=b && b<=a) return b;  if (a<=c && c<=b ||b<=c && c<=a) return c;  }  dog d1{...};  dog d2{...};  dog d3{...}; | struct Dog{  int age;  string name;  float weight;  bool operator <=(const dog da, const dog db){  return a.name <= b.name;  }  }; | struct DogLEage{  bool operator()(const dog &a, const dog &b){return a.age <= b.age};  }  template <typename T, typename compare>  T middle(T a, T b, T c){  // retourne l'élément du "millieu"  compare cmp  if (cmp(a,b) && cmp(b,c)||cmp(c,b) && cmp(b,a)) return b;  if (cmp(a,c) && cmp(c,b)||cmp(b,c) && cmp(c,a)) return c;  }  cout << middle<Dog, DogLEage>(d1,d2,d3) << endl; |
| middle(1,5,7) compiles without errror because  compilator finds T type via parameters but  midle(d1,d2,d3) errror because  operator <= not defined | Only one operator overload so cant compare dogs weight or age | Le code fonctionne sans erreur pour les chiens (mais n’est plus  compatible avec les int, etc) car nous avons surcharger l’opérateur () .  C’est devenu un objet qui se comporte comme une fonction |





Une image contenant texte

Description générée automatiquement



Une image contenant texte

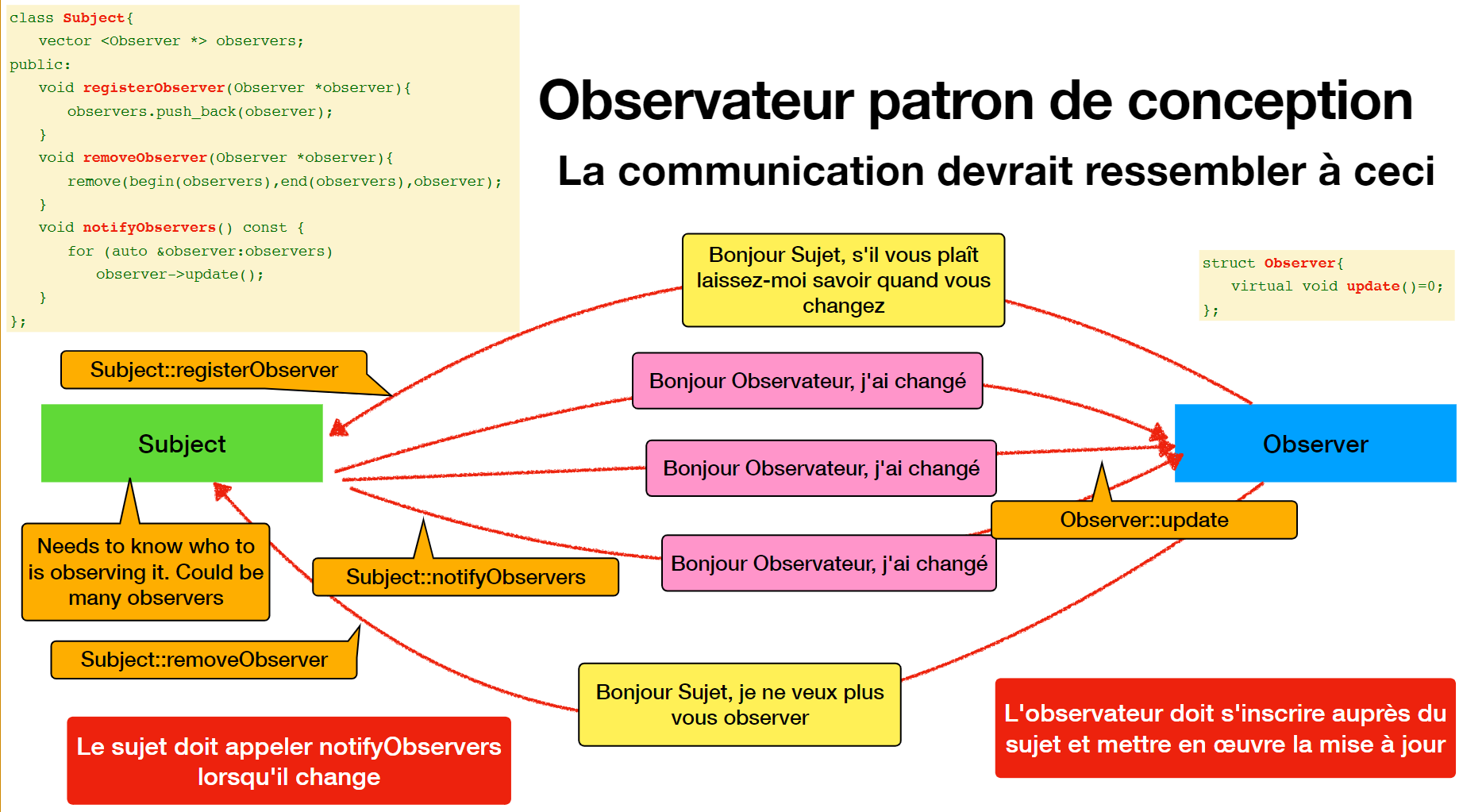
Description générée automatiquement

|  |  |
| --- | --- |
|  | Template |
| Application | Class  method  function |
| Parameters | Types |
| Compilation | Separately with each parameter choice = instanciation |
| Goal | Execute different code on different generic template parameters |
| Function passed as parameter | Lambda expression  =functor  =class with only operator() |

|  |  |
| --- | --- |
| Inheritance  OOP option  Allows class methods reuse  On fait hériter un objet quand on détermine  que ce dernier est un “type de” un autre objet. Par exemple un étudiant est un type  de personne. Nous pouvons donc faire hériter étudiant de personne.  Notez que le mot clé virtual permet d’appeler une fonc�on du type de  l’instance plutôt que du type de l’objet pointé. Ainsi la méthode print() appelée  sur un objet de type étudiant affichera program plutôt que firstname  La conversion d’un objet enfant vers un objet parent est appelé du upcas�ng, son  opposé est appelé du downcas�ng | class Person{  string firstName, lastName;  public:  Person(string firstName, string LastName):  firstName(firstName), lastName(LastName){}  // getters et setters ...  virtual void print() {cout << firstName << endl};  virtual ~Person();  }  class Student: public Person{  string program;  public:  Student(string first,string last, string program):  Person{first, last}, program(program){};  void print(){cout << program << endl; // fonction héritée  } |

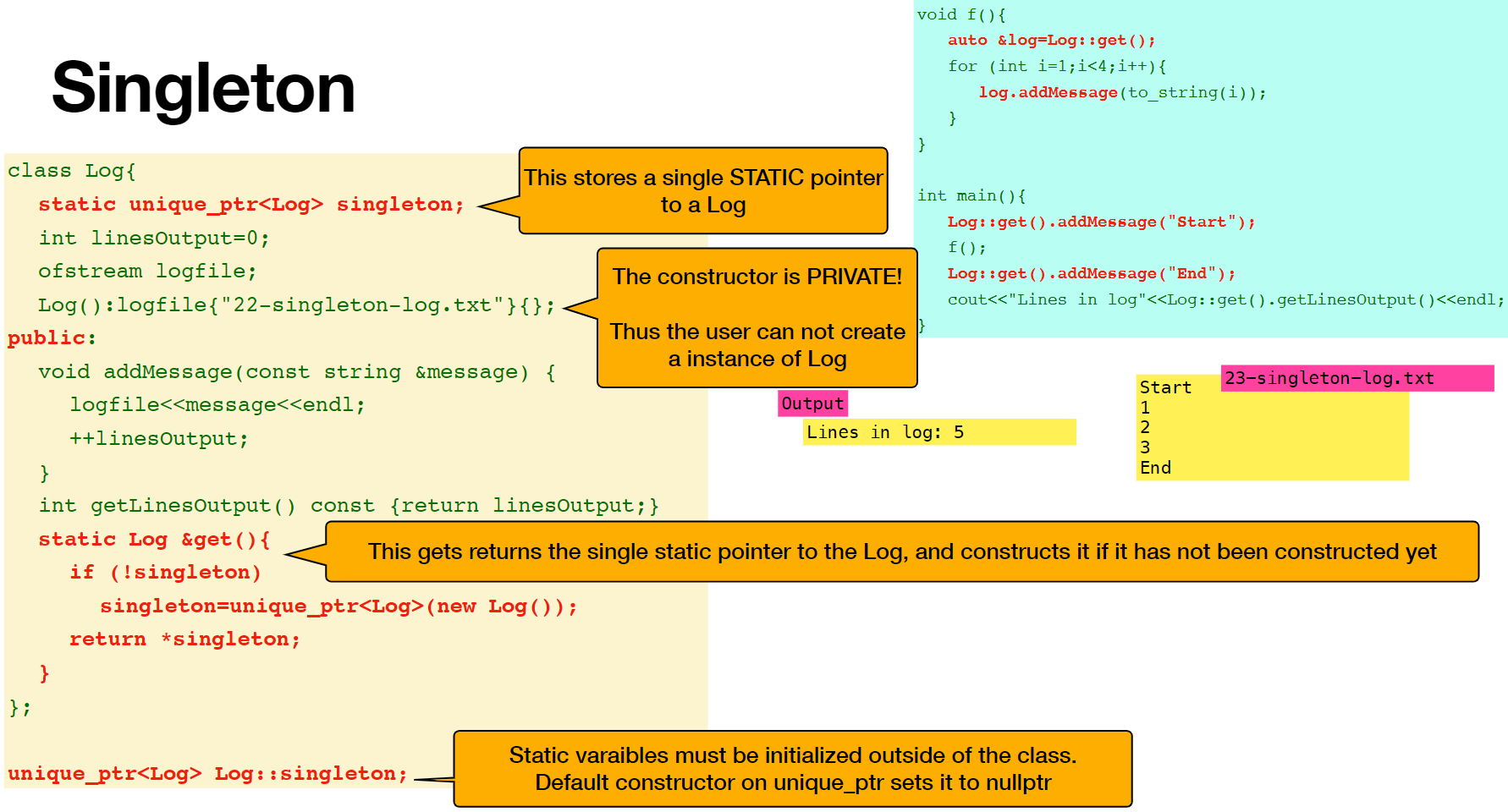
|  |  |  |  |
| --- | --- | --- | --- |
| Class | Person members access control | | |
| Person | Public | Protected | Private |
| class Student : **public** Person {  ...  } | Public | Protected | Private |
| class Student : **protected** Person {  ...  } | Protected | | Not accessible |
| class Student : **private** Person {  ...  } | Private | | Not accessible |

|  |  |
| --- | --- |
| final | devant une méthode indique qu’il s’agit de la dernière version et qu’elle ne peut pas être surchargée via héritage |
| typeid(Classname) | class type |



Une image contenant texte

Description générée automatiquement



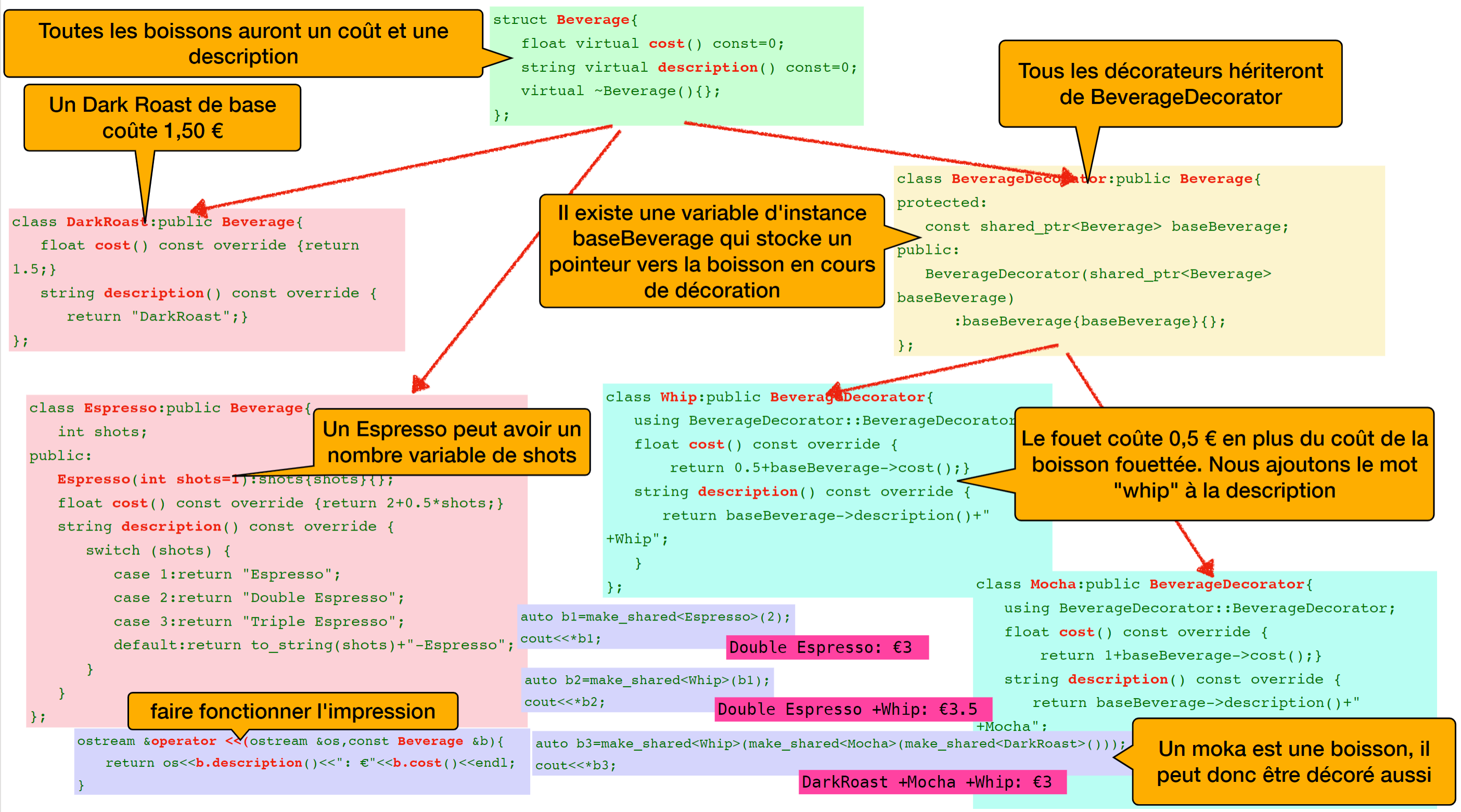
Singleton :

Class unique instance

Constructed only if used

Constructor access : private

Encapsulated class logic



Une image contenant texte

Description générée automatiquement

|  |  |  |
| --- | --- | --- |
| Constructor | Factory method | Builder class |
| construct max 1 class methods | Non constructor  static | Distinct class or class hierarchy |
| methods/functions can return a pointer of  base class pointer type, which could be any base class descendant | can construct and return one or more class instance pointer/reference | Creates other class instances |

MVC