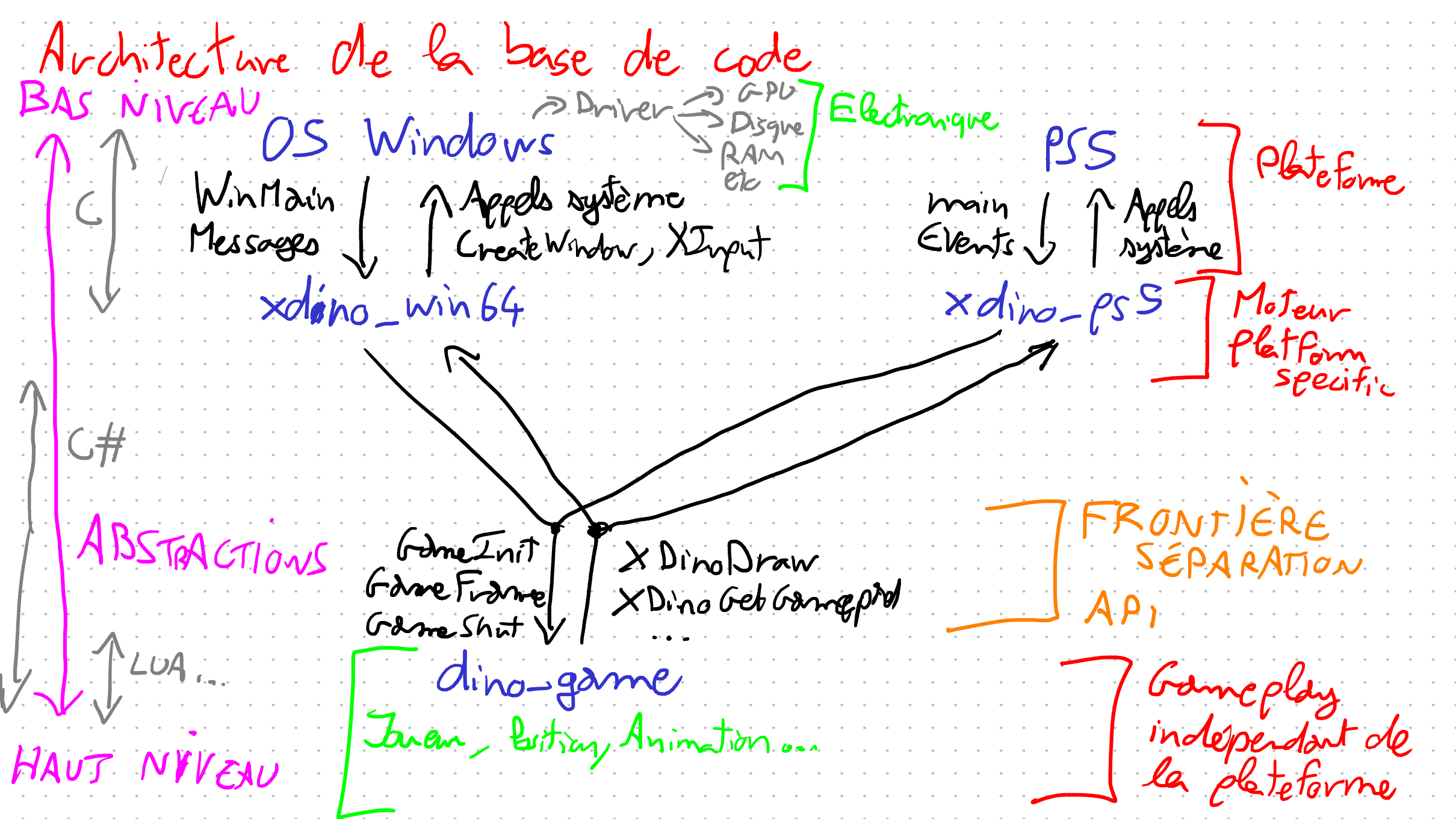
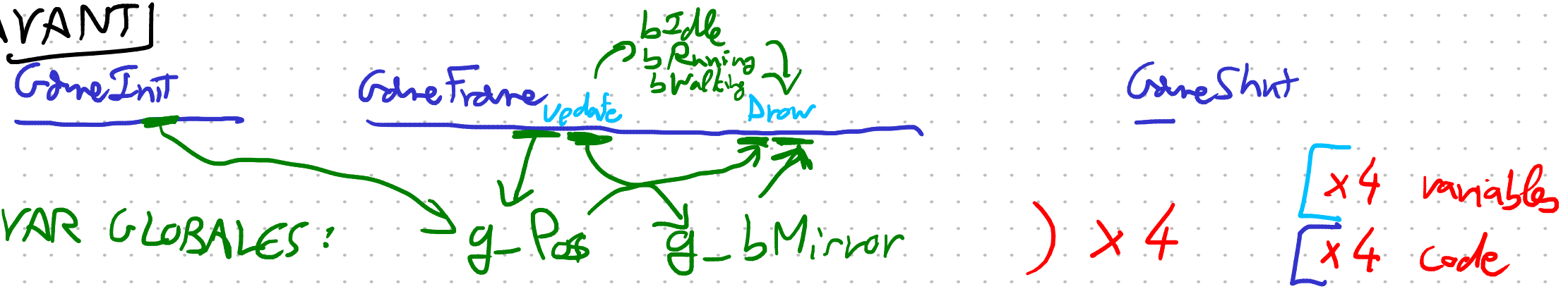
Architecture de la base de code



AVANT



APRÈS

```
Struct DinoPlayer {
    pos
    bMirror
    bIdle
    bRunning
    bWalking
    void UpdatePlayer();
    void DrawPlayer();
}; DinoPlayer p1, p2, p3, p4;
```

```
void DinoPlayer::UpdatePlayer(  )
void DinoPlayer::DrawPlayer(  )

player1.UpdatePlayer(—)
    this -> pos        ou    pos
    this -> bMirror          bMirror
```

CE QU'ON A FAIT

```
void DinoPlayer::UpdatePlayer()
void DinoPlayer::DrawPlayer()
    this->pos.x += 10
```

MEMBRE — IMPLICITE

] FONCTIONS MEMBRES = MÉTHODES

CE QU'ON AURAIT PU FAIRE

```
void UpdatePlayer(DinoPlayer* p, ...)
    p->pos.x += 10
```

EXPLICITE

(Par pointeur pour ne pas copier)

] FONCTIONS LIBRES

⟺ ÉQUIVALENT

struct VS classes (C++ s'appelait "C with classes")

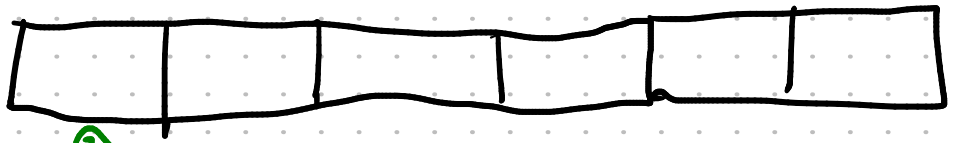
Une classe est comme une structure (en C++), sauf qu'on rajoute une frontière, une séparation, entre le comportement et les détails d'implémentation.

[ex: comportement = je me déplace
détail = le sprite choisi pour l'anim

```
class DinoPlayer {
public:
    void UpdatePlayer();
    void DrawPlayer();
    void Init(color, posInit);
private:
    m_pos;  m_bMirror
    m_bWalking  m_bRunning
                m_bIdle
    m_color
};
```

Public = comportement
Private = implémentation

std::vector et les algorithmes C++



begin()

end()

Itérateurs

++ ++ ++ ++ ++ ++

VERSION INDICE

```
for (int i=0; i<vec.size(); ++i){
    vec[i] = ...
}
```

VERSION FOR-RANGE

```
for (int& i : vec){
    i = ...
}
```

VERSION ITÉRATEUR

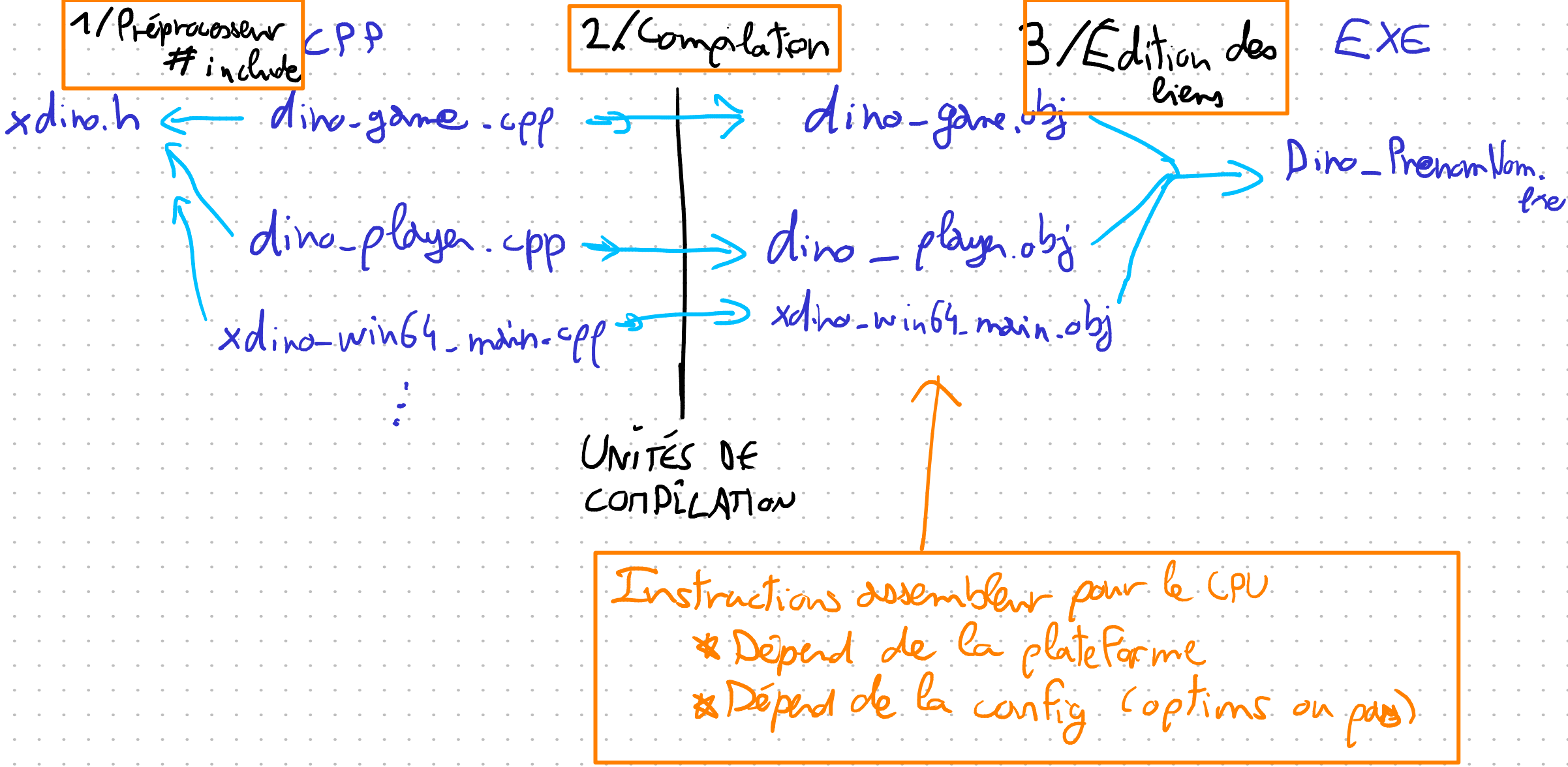
```
it = vec.begin()
itend = vec.end()
while (it != itend) {
    *it = ...   } MODIFIE LA CASE RÉFÉRÉE
    it++   } CHANGE DE CASE
}
```

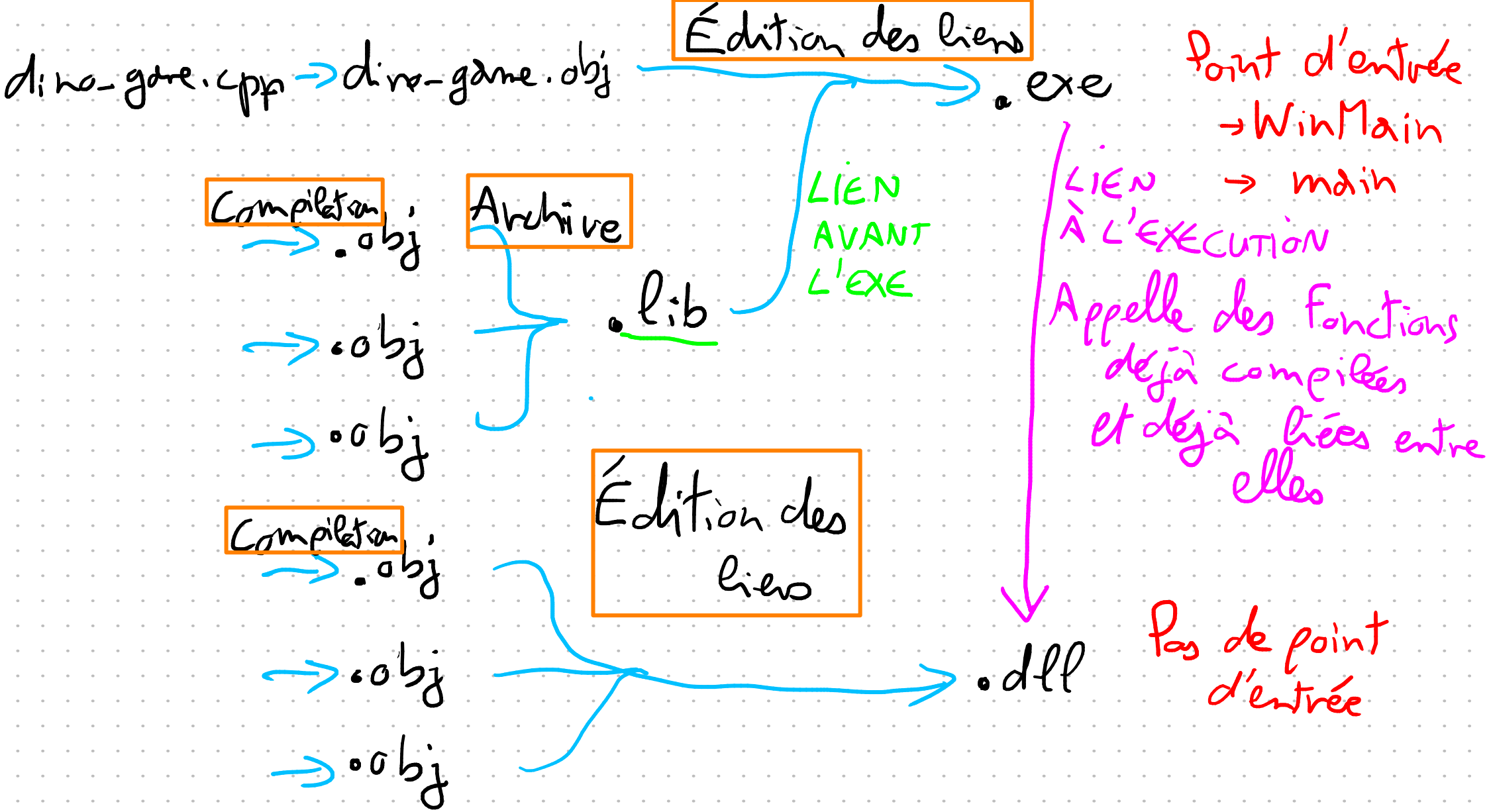
std::sort plus petit au plus grand

- start ← g_Players.begin()
- last ← g_Players.end()
- Foncteur ← suivant quel critère A < B ?

```
bool DinoPlayer::IsAbove(DinoPlayer& p2){
    return m_pos.y < p2.m_pos.y
}

bool CompareDinosPos(DinoPlayer& a, DinoPlayer& b){
    return a.IsAbove(b);
}
```

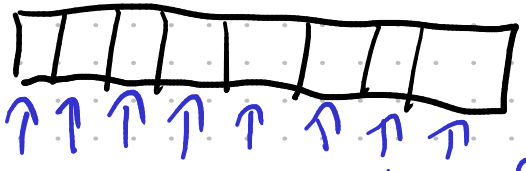

1/ Préprocesseur
#include
CPP
2/ Compilation
3/ Edition des liens
EXE
xdino.h
dino_game.cpp
dino_game.obj
Dino_PrenomNom.exe
dino_player.cpp
dino_player.obj
xdino_win64_main.cpp
xdino_win64_main.obj
UNITÉS DE COMPILATION
Instructions assembleur pour le CPU
* Dépend de la plateforme
* Dépend de la config (optims ou pas)


dino-game.cpp → dino-game.obj
Édition des liens
.exe
Point d'entrée
→ WinMain
Compilation
.obj
.obj
.obj
Archive
.lib
LIEN AVANT L'EXE
LIEN → main
À L'EXÉCUTION
Appelle des fonctions déjà compilées et déjà liées entre elles
Compilation
.obj
.obj
.obj
Édition des liens
.dll
Pas de point d'entrée

Mémoire

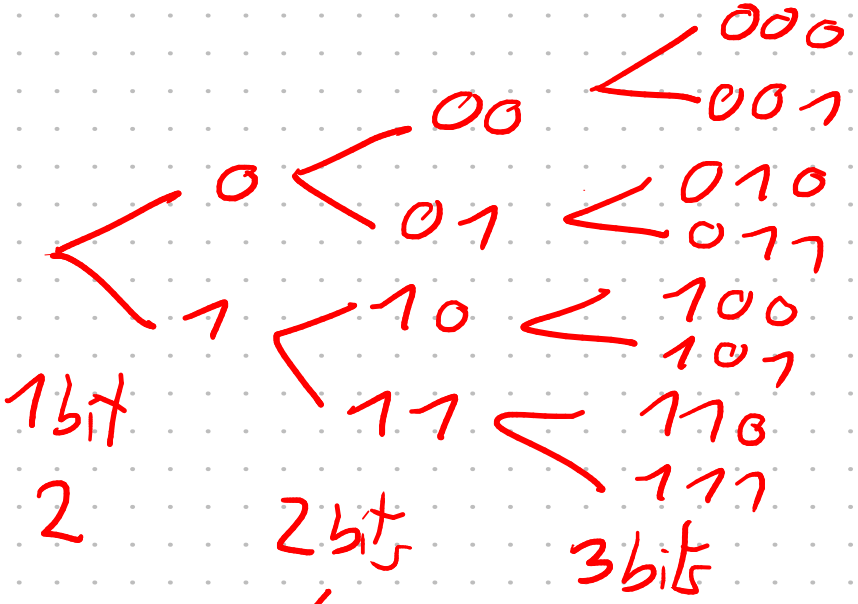
Par exemple, une machine avec 32 GB RAM

Type * → entiers qui identifient une case



Pointeurs = adresses

Machine 64-bits : adresses sont stockés dans des entiers 64 bits



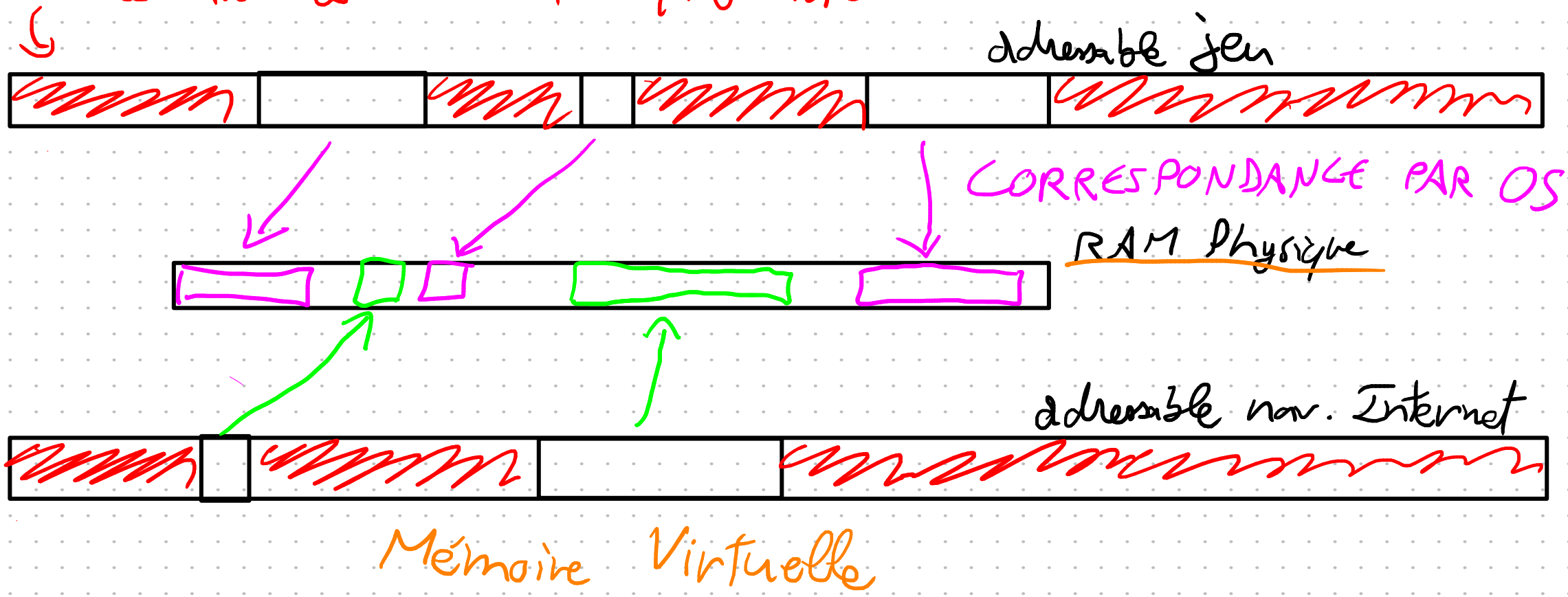
n bits → 2^n adresses possibles

Pointeurs 64 bits → 2^{64} octets dans l'espace adressable

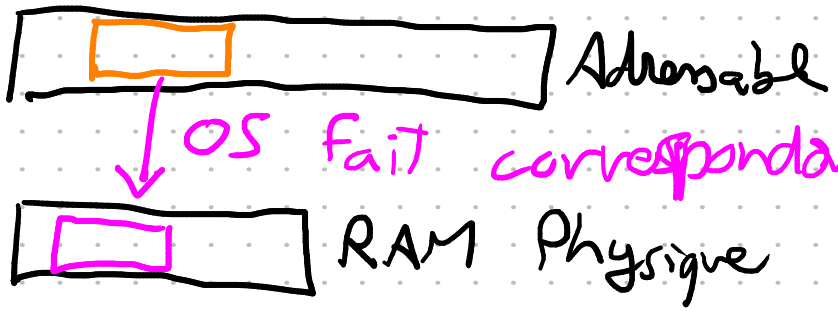
~16 000 000 000 GB adressable

32 GB RAM physique

Adresses invalides = Crash du programme



Allocation



→ Discuter entre l'OS, le programme, la RAM, et le CPU

~ quelques microsecondes

à en 60 Fps

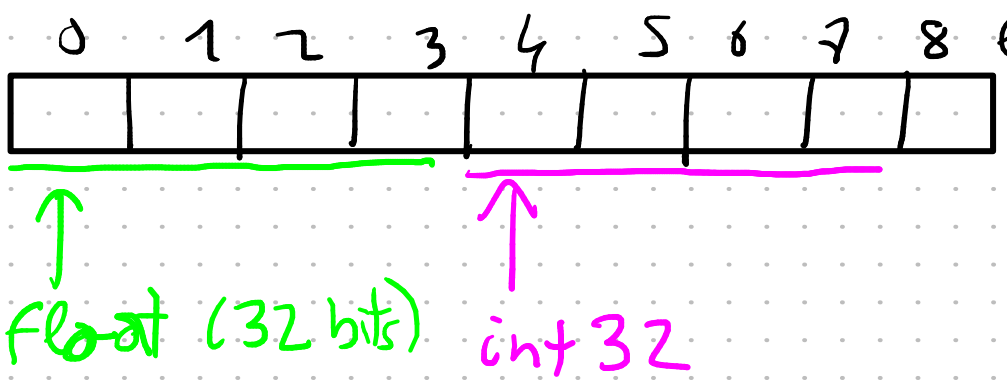
→ 16 000 microsecondes

~ 5000 allocations si on fait que ça

CPU 4 GHz

4 000 000 000 ops/s

4 000 ops/microseconde




```
int A = 5;
int* pA = &A;
```

pA

Une même séquence de bit peut être interprété différemment

0111 1111 1000 0000 0000 0000 0000 0000

R V B A

Couleurs RGBA → R = 127 V = 128 B = 0 A = 0

Float → $+\infty$

int32 → 2 139 095 040

Type = Outil du langage de programmation pour garder l'information de comment interpréter une suite de bits et quelles opérations sont autorisées

Typage statique : le compilateur connait les types associés aux emplacements mémoire, donc il peut détecter des erreurs logiques.

```
/// Représente une coordonnée
struct DinoVec2 {
    float x;
    float y;
};

/// Représente une couleur de
union DinoColor {
    struct {
        uint8_t r;
        uint8_t g;
        uint8_t b;
        uint8_t a;
    };

    uint32_t rgba;
};
```

→ 64 bits
→ 32 bits
→ 32 bits

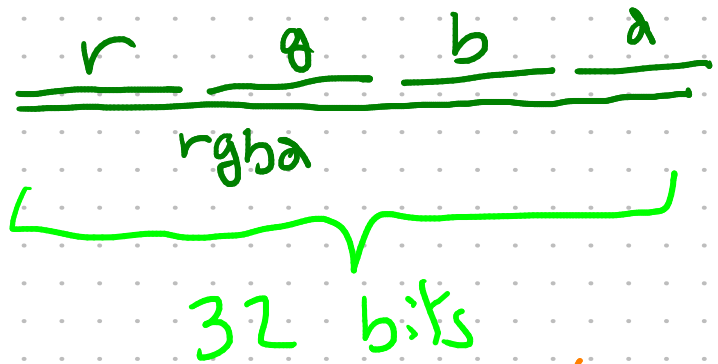
DinoVec2 = x (32) + y (32) = 64 bits

Structure = Ensemble de variables

4 uint8_t → 4 octets → 32 bits

} 32 bits

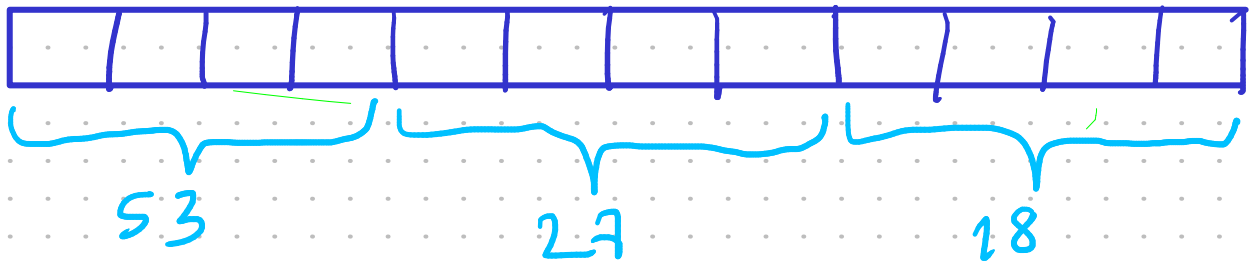
DinoColor



Union = Même emplacement mémoire, qui au cours de l'exécution, va y stocker des types différents.

Connaître la taille d'un type en mémoire sizeof (Type)

std::vector<int32> = { 53, 27, 18 };



↑ ptr début ↑ ptr fin

class vector {
64 bits — ptrdebut
64bits — ptrfin
};

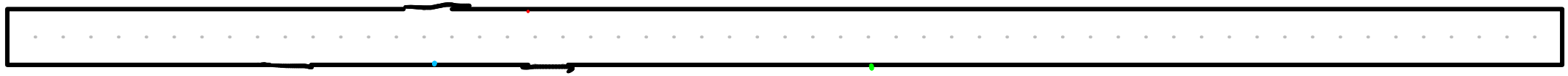
vector ~ 128 bits

[0] ptrdébut
[1] ptrdebut + sizeof (int32)
[2] ptrdébut + sizeof (int32) x 2

1. OS commence le programme EXE

→ Dans le fichier EXE, il y a la taille nécessaire pour les variables globales

→ OS alloue la stack = la pile



WinMain
↘ CreateWindow
↓
GameFrame
↘ std::sort
↘ ComparePlayerPos
↘ IsAbove

stack = 1 grosse allocation

à chaque appel de fonction
"on grignote" dedans

à chaque return
"on rend la mémoire"

```
std::vector<DinoAnimal> g_Animals;
```