Cours de C++ (M3103)

{ A. Casali / M. Laporte

Plan

A. Récursivité

B. Adresse mémoire

C. Pointeurs en mémoire dynamique

A. Récursivité

A.1 Définition

un sous-programme est dit récursif si sa définition fait appel à lui-même, directement ou pas.

A.2 Formulation informatique

```
void [T'] f (T [&] x, …)
{

    ...
    f (y);
    ...

}
```

chaque appel à `f()` résout une partie du problème sur le "paramètre" `x` puis confie le reste du problème, plus simple, ... à `f()` sur le "paramètre" `y`.
C'est le principe de la stratégie « diviser pour régner » (divide & conquer).

Et on sort quand de l'appel récursif ?

➡ Il doit toujours y avoir une condition de sortie / terminaison (même si elle est cachée).

A.3 Récursivité directe simple

la récursivité est directe simple lorsqu'un sous-programme s'appelle directement lui-même une seule fois.

Exemple : Calcul de `n!`

Formulation sans récurrence :

$$n! = \begin{cases} \prod_{i=1}^{n} n \; si \; n \geq 1 \\ 1 \; si \; n = 0 \end{cases}$$

Formulation avec récurrence :

$$n! = \begin{cases} n * (n-1)! \; si \; n \geq 1 \\ 1 \; sinon \end{cases}$$

```
unsigned Facto (unsigned n)
{
    if (n <= 1) return 1;
    return n * Facto (n-1);

}
```

A.5 Attention avec la récursivité

La récursivité n'est pas la solution à tous nos problème.

Exemple : afficher le contenu d'une string caractère par caractère

Non récursif :

```
Procedure AfficheChaine (Chaine : in str)
debut
    pour (i vrariant_de 0 à taille(Chaine) - 1)
    faire
        Afficher (Chaine [i]);
    ffaire
fin
```

Récursif	Oui	Non
Nb Contexte	N + 1	1
Nb Objets (utilisés)	2N + 2	2

Avec récursivité :

```
Procedure AfficheChaine (Chaine : in str)
debut
    AffichCarac (Chaine, 0);
fin

procedure AffichCarac (Str :in string, Pos : in entier_nat)
debut
    si (Pos vaut taille (Chaine) return;
    afficher (Chaine [Pos]);
    AffichCarac (Str, Pos +1);
fin
```

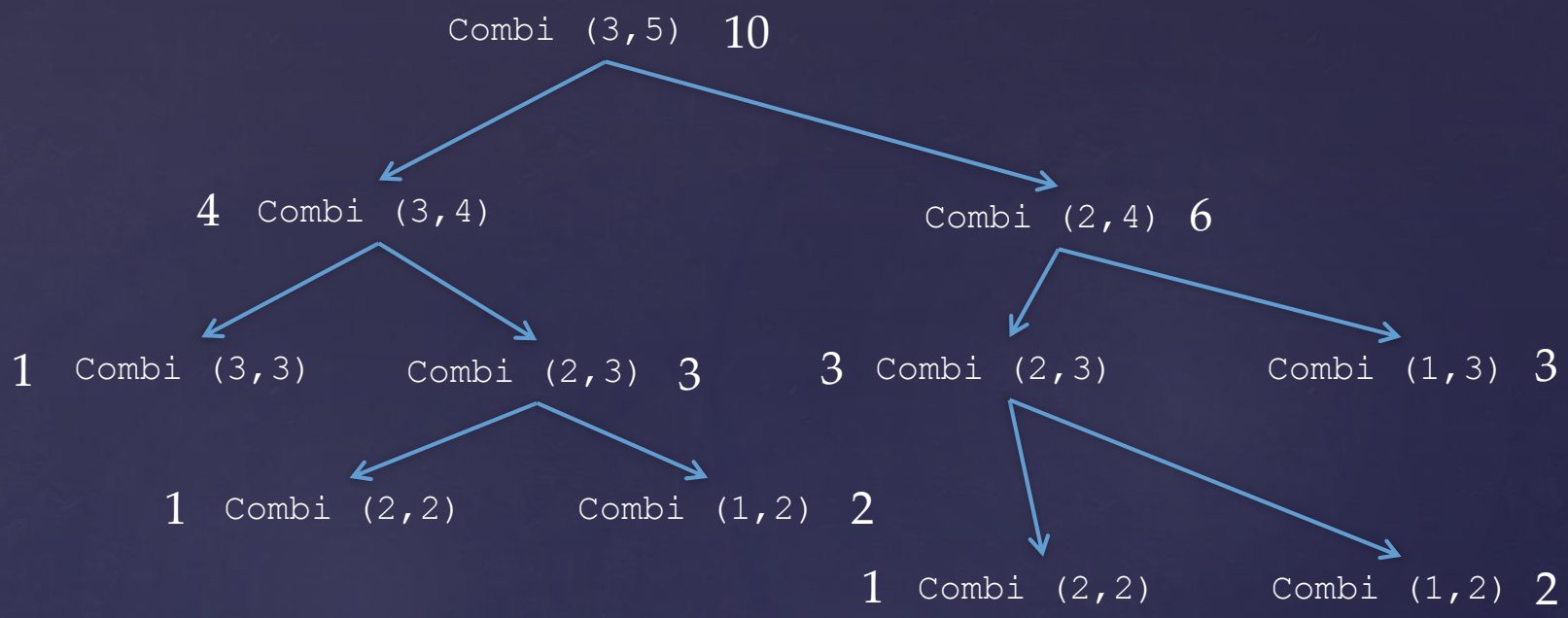
A.5 Récursivité directe multiple

la récursivité est directe multiple lorsqu'un sous-programme s'appelle directement lui-même une ou plusieurs fois.

Exemple : Calcul du nombre combinatoire

$$C_n^k = \begin{cases} 1 \; si \; k \; vaut \; n \\ k \; si \; k \; vaut \; 1 \\ C_{n-1}^k + C_{n-1}^{k-1} sinon \end{cases}$$

```
unsigned Combi (unsigned k, unsigned n)
{
    if (n == k) return 1;
    if (1 == k) return k;
    return Combi (k, n-1) + Combi (k-1, n-1);
}
```



La trace de l'algorithme forme un arbre binaire : à chaque embranchement de l'arbre (nœud), on a au plus deux sous – arbres (fils).

Exemple : trier un vecteur

```
void QuickSort (vector <T> &V, unsigned Min, unsigned Max)
{
    if (Max == Min) return;
    unsigned Pivot = V[Min];
    unsigned PosPivot (Ventiler (V, Pivot, Min, Max));
    QuickSort (V, Min, PosPivot);
    QuickSort (V, PosPivot + 1, Max);
}
```

5	1	2	12	24	13	10	2

Appel de `Ventiler ()`

1	2	2	5	24	13	10	12

- Tous les chiffres plus petits que `Pivot` (ici 5) sont dans le sous vecteur d'indice plus petit que `PosPivot` (ici 3), les autres sont dans le sous vecteur d'indice plus grand.
- On trie chacun des sous vecteurs indépendamment l'un de l'autre.

A.6 Récursivité croisée / mutuelle

Deux algorithmes sont mutuellement récursifs si l'un fait appel à l'autre, et l'autre fait appel à l'un.

Exemple : parité d'un entier

```
bool Pair (unsigned n)
{
   return ((0 == n) ? true : Impair (n-1));
}

bool Impair (unsigned n)
{
   return ((0 == n) ? false: Pair (n-1));
}
```

```
Pair (2) => Impair (1) => Pair (0) => true

Impair (2) => Pair (1) => Impair (0) => false
```

A.7 Récursivité terminale

Un sous programme est récursif terminale lorsqu'aucun traitement ne suit l'appel récursif.

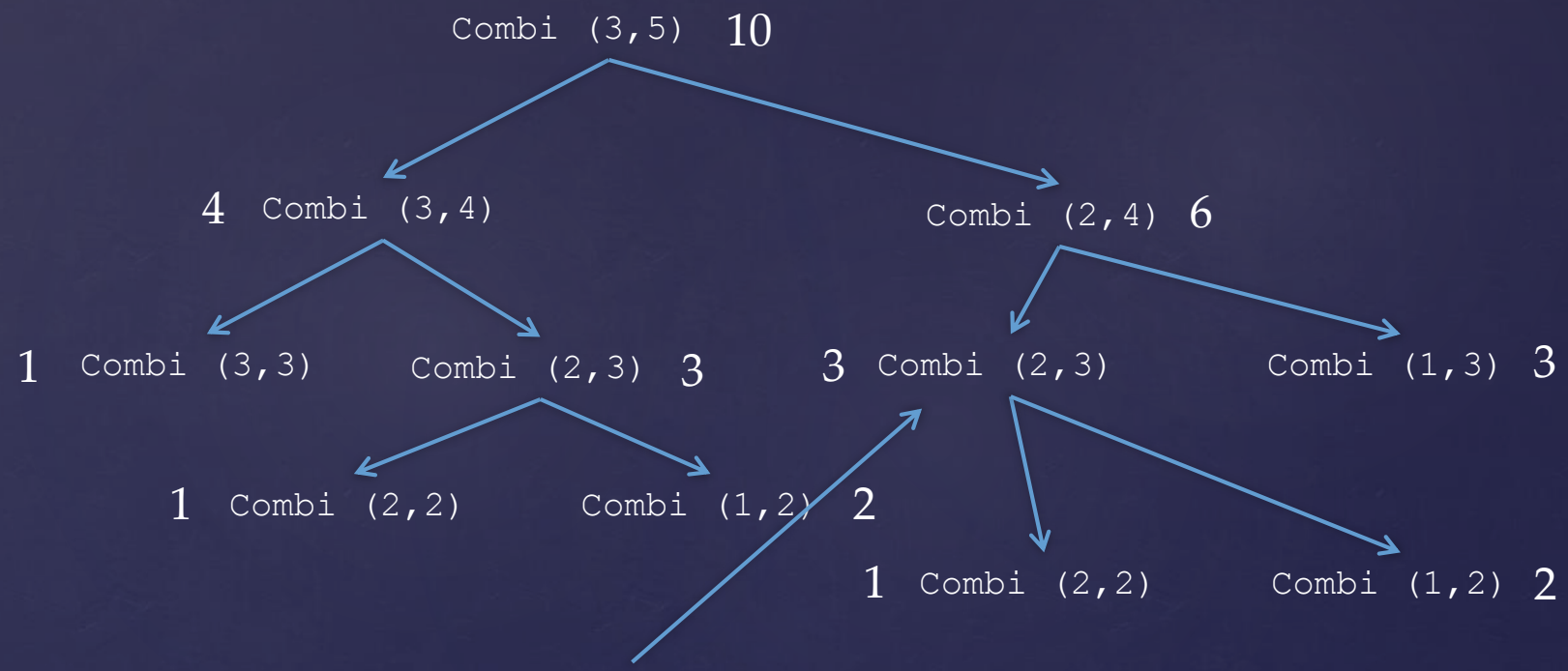
Exemple :

```
procedure AffichCarac (Str :in string, Pos : in entier_nat)
debut
    si (Pos vaut taille (Chaine) return;
    afficher (Chaine [Pos]);
    AffichCarac (Str, Pos +1);
fin
```

Pour ce type de sous programme, on peut facilement casser l'appel récursif en utilisant des instructions itératives (boucles).
Pour les autres, on doit passer par l'instanciation d'une pile qui permet d'enregistrer les différentes informations renvoyées / modifiées lors des appels récursifs successifs.

A.8 Mémoïsation

Technique d'optimisation de codage permettant de réduire le temps d'exécution d'un sous programme en mémorisant ses résultats



On a aucun intérêt à recalculer Combi (2,3)

On stocke dans un tableau les valeurs déjà trouvées, sinon on trouve 0.

```
unsigned Combi (unsigned k, unsigned n, Matrice & TP)
{
    if (TP [n][k] != 0) return TP [n][k];
    TP [n][k] = Combi (k, n-1) + Combi (k-1, n-1);
    return TP [n][k];
}
```

Reste à savoir comment on initialise la matrice TP!

```
void InitTP (Matrice & TP, unsigned n)
{
    TP.resize (n + 1);
    for (unsigned i (1); i < TP.size (); ++i)
    {
        TP[i].resize (i + 1);
        TP [i][1] = i; //if (1 == k) return k;
        TP [i][i] = 1; // if (n == k) return 1;
    }
}
```

A.9 Limiter le nombre d'appels récursifs

Pb : les derniers appels récursifs du tri rapide sont très couteux (mémoire + temps), alors que la taille du sous-tableau est petite. Dans ce cas, on trie le sous-tableau avec une autre méthode.

```
void QuickSort (vector <T> & V, unsigned Min, unsigned Max)
{
    if (Max <= Min + 15) {
        BubleSort (V, Min, Max);
        return;
    }
    unsigned Pivot = V[Min];
    unsigned PosPivot (Ventiler (V, Pivot, Min, Max));
    QuickSort (V, Min, PosPivot);
    QuickSort (V, PosPivot + 1, Max);
}
```

Plan

A. Récursivité

B. Adresse mémoire

C. Pointeurs en mémoire dynamique

B. Adresse mémoire

B.1 Adresse des variables

Chaque variable possède sa propre adresse mémoire.
Accès : `&VarIdent`

```
const int CstInt = 10;
double    Double;
bool      Bool;
string    Str;

cout << &CstInt            0x7fff5c8c9a1c
     << &Double            0x7fff5c8c9a10
     << &Bool              0x7fff5c8c9a0f
     << &Str;              0x7fff5c8c9a00
```

B.2 Partage d'adresse mémoire

Plusieurs variables peuvent partager la même adresse mémoire. La condition est qu'elles soient toutes de même type.

```
int Int = 10;
int &RefInt = Int;

cout << Int << '\t'<< sizeof (Int) << '\t'<< &Int   << '\n'
     << RefInt  << '\t'<< sizeof (RefInt)  << '\t'<< &RefInt << '\n';
```

```
10  4   0x7fff505e4a14
10  4   0x7fff505e4a14
```

```
++Int;
cout << Int << '\t'<< sizeof (Int) << '\t'<< &Int   << '\n'
     << RefInt  << '\t'<< sizeof (RefInt)  << '\t'<< &RefInt << '\n';
```

```
11  4   0x7fff505e4a14
11  4   0x7fff505e4a14
```

La modification d'une des variables entraine celle des autres variables qui partagent la même zone mémoire.

B.3 Passage de paramètres par référence dans les fonctions

```
void f1 (int & i)
{
    cout << "&i = " << &i << '\n';
}

void f2 (int i)
{
    cout << "&i = " << &i << '\n';
}
…
int   A = 10;
cout << "&A = " << &A << '\n';
f1 (A);
f2 (A);
```

```
0x7fff5854fa2c
0x7fff5854fa2c
0x7fff5854fa0c
```

Lorsque l'objet est passé par référence, on travaille bien sur la même adresse mémoire.

B.4 Passage de tableaux en paramètre de fonctions

```
template <typename T>
void AffichTab (T Tab [])
{
    cout << sizeof (Tab)
         << Tab << endl;
}
…
float TabFloat [] = {1.2, 2.3, 3.4};
AffichTab (TabFloat);            8 0x7fff536dda20
char TabCar [] = {'a', 'b', 'c'};
AffichTab (TabCar);              8 abcY?
char TabCar2 [] = {'a', 'b', 'c', '\0'};
AffichTab (TabCar2);             8 abc
```

Remarque : on aurait pu remplacer la généricité du C++ par `void*`

Plan

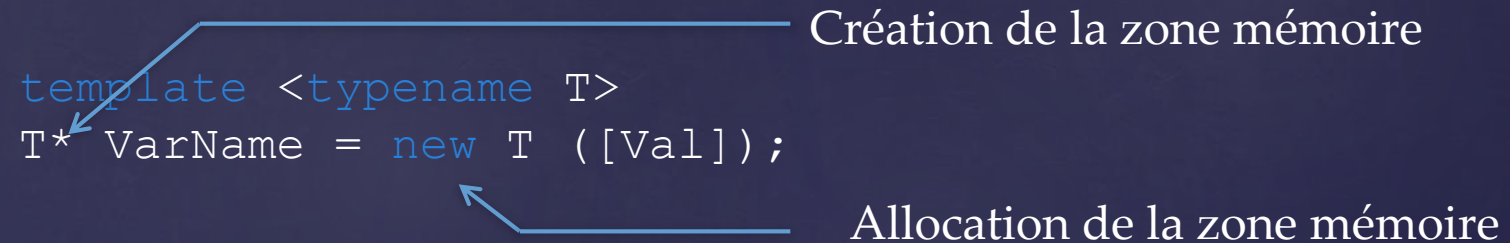
A. Récursivité

B. Adresse mémoire

C. Pointeurs en mémoire dynamique

C. Pointeurs en mémoire dynamique

C.1 Déclaration en mémoire dynamique



Exemple :

```
int* pInt = new int (10);
```

C.2 Accès en Lecture / Ecriture

```
*VarName = SomethingCompatibleWithT;
AVarIdentCompatibleWithT = *VarName;
```

Exemple :

```
*pInt = 10;
int a = *pInt;
```

```
int* i;
cout << "adresse de i : " << i
     << " valeur de i : " << *i
     << endl;
```

```
Segmentation fault: 11
```

On a bien déclarer un pointeur, mais il pointe sur quoi??

```
int k = 5;
int* j =  &k;
cout << "adresse de j : " << j          0x7fff5032ba0c
     << "valeur de j : "  << *j         5
     << endl;

cout << "adresse de k : " << &k          0x7fff5032ba0c
     << "valeur de k : "  << k           5
     << endl;
```

C.3 Destruction

```
delete VarName;
```

Désalocation de la zone mémoire

Exemple :

```
delete pInt;
```

Le pointeur existe toujours, seul l'objet pointé est détruit!

```
int* i;
i = new int (10);
cout << i
     << *i << endl;
delete i;

cout << i;
```

```
0x7fbae2c000e0
10

0x7fbae2c000e0
```

C.4 Réallocation d'une zone mémoire

```
VarName = new T ([Val]);
```

Allocation d'une nouvelle zone mémoire

```
int* i = new int (10);
cout << "adresse de i : " << i          0x7fd830404c80
     << " valeur de i : " << *i         10
     << endl;

i = new int (20);
cout << "adresse de i : " << i          0x7fd830404c90
     << " valeur de i : " << *i         20
     << endl;
```

C.5 Pointeur à contenu constant

```
template <typename T>
const T* VarName = new T ([Val]);
```

```
const int* i = new int (10);
cout << "adresse de i : " << i          0x7fd830404c80
     << " valeur de i : " << *i         10
     << endl;

i = new int (20);
cout << "adresse de i : " << i           0x7fd830404c90
     << " valeur de i : " << *i          20
     << endl;

*i = 30;
```

Erreur de compilation :

```
File.cxx:XX:8: error: assignment of read-only
location '* i'
```

C.6 Pointeur à adresse constante

```
template <typename T>
T* const VarName = new T ([Val]);
```

```
int* const i = new int (10);
cout << "adresse de i : " << i          0x7fd830404c80
     << " valeur de i : " << *i         10
     << endl;

*i = 30;
cout << "adresse de i : " << i          0x7fd830404c80
     << " valeur de i : " << *i         30
     << endl;

i = new int (20);
```

Erreur de compilation :

```
File.cxx:XX:8: error: error: assignment of read-only
variable 'i’
```

C.7 Opérateur ->

```
CPersonne* P1 = new CPersonne ("Casali", "Alain");
cout << *P1.GetNom () << endl;
```

Erreur de compilation : l'opérateur `'.'` a une priorité supérieure à l'opérateur `'*'`

Solution 1 : parenthéser le pointeur

```
cout << (*P1).GetNom () << endl;
```

Solution 2 : utiliser l'opérateur ->

```
cout << P1->GetNom () << endl;
```

C.8 Pointeur et Polymorphisme

```
class CPersonne
{
    …
    void Afficher ()
    {
        cout << "Je suis "
       "une  personne" << endl;
    }
};
```

```
class CProf : public CPersonne
{
    …
    void Afficher ()
    {
        cout << "Je suis un "
             "Prof" << endl;
    }
};
```

```
CPersonne* P1 = new CPersonne ("Casali", "Alain");
P1->Afficher ();          Je suis une personne
CPersonne* P2 = new CProf ("Casali", "Alain");
P2->Afficher ();          Je suis une personne
```

PB : comment utiliser le polymorphisme?

Il faut qualifier chaque fonction utilisant le polymorphisme de virtuelle

```
class CPersonne
{
    …
    virtual void Afficher ()
    {
        cout << "Je suis "
       "une  personne" << endl;
    }
};
```

```
class CProf : public CPersonne
{
    …
    virtual void Afficher ()
    {
        cout << "Je suis un "
             "Prof" << endl;
    }
};
```

```
CPersonne* P1 = new CPersonne ("Casali", "Alain");
P1->Afficher ();        Je suis une personne
CPersonne* P2 = new CProf ("Casali", "Alain");
P2->Afficher ();        Je suis un prof
```

C.8 Pointeur, assignement et test d'égalité

```
CPersonne* P1 = new CPersonne ("Casali", "Alain");
CPersonne* P2 = P1;

if (P2 == P1)
    cout << "même adresse" << endl;
```

C.9 Fuite mémoire

```
CPersonne* P1 = new CPersonne ("Casali", "Alain");
CPersonne* P2 = P1;

if (P2 == P1)
    cout << "même adresse" << endl;

delete P1;
cout << P2->GetNom ();
```

```
terminate called after throwing an instance of
'std::length_error'
  what():  basic_string::_S_create
Abort trap: 6
```

La case sur laquelle pointait P1 a été détruite, et par conséquent celle de P2 aussi.

Solution :

Il faudrait créer sa propre classe de pointeur :

- Un entier naturel désigne le nombre d'instance d'objet qui pointe sur la case mémoire;
- Chaque fois qu'on passe dans le constructeur, on incrémente cet entier;
- Chaque fois qu'on passe dans le destructeur ou qu'on fait appel à `delete`, on décrémente cet entier.
- S'il est nul, on peut supprimer la case mémoire sans problème.

Les *smart pointers* sont la pour vous ☺