

LANGAGE C++

MEMO COURS N°1

Voilà le travail qu'il faudrait savoir faire seul après avoir suivi assidûment un cours et l'avoir travaillé avec le support de cours (disponible sur Campus).

Les Memos des cours suivants seront à votre charge.

ON CONNAIT DEJA BIEN PEU DE CHOSES

```
#include <iostream>
using namespace std;
cin >> x
cout << x << endl
new
delete
delete []
```

Commentaires sur plusieurs ou une seule ligne

```
/*
*
*/
/*          */
//
```

Pointeur générique du C : void *

top niveau en C pour assurer la généricité
plus vraiment utile en P.O.O : programmation orientée objet
il existe des templates pour la programmation générique (à suivre ...)

Pour les constantes :

#define MAX 10	ON OUBLIE
const int MAX = 10;	A FOND
int tab[MAX];	OK

les const : ils sont partout!!! ET C'EST TANT MIEUX!

char * p	==> pointeur variable et zone pointée modifiable
const char * p	==> zone pointée constante (non modifiable, écriture interdite)
char * const p	==> pointeur constant (comme un tableau)
const char * const p	==> PERSONNE NE BOUGE!

Les tentatives d'écriture sont vérifiées à la compilation ==> COMPILE TIME ERROR
On travaille donc avec un garde-fou / un filet : PROGRAMMATION PAR CONTRAT
IDEE MAJEURE : NE PAS ROMPRE UN CONTRAT!

Utilisation la plus pertinente des const: passage de paramètres à une fonction

void f(const int *p1, int *p2)

DECLARATION DE VARIABLES

juste à l'endroit où l'on en a besoin ==> on évite le recours à la lecture YOYO

PREMIER CONCEPT MAJEUR DU C++ :

LES REFERENCES

alias d'une variable : notation & (à ne pas confondre avec & adresse d'une variable)

utilisation comme une variable ordinaire

```
int a = 3
int &b = a;    // une référence doit être initialisée lors de sa déclaration
// b vaut 3    pas de notation *p comme en C si p = &a
b = b + 4;    // b et a valent 7
```

Utilisation en passage de paramètres

OBJECTIF 1 : plus facile à utiliser que les pointeurs

//passage par copie des valeurs des pointeurs : OK CA MARCHE!

```
void echange(int *a, int *b)
{
    int tmp = *a; *a = *b; *b = tmp;
}
```

//passage par copie des valeurs des références : OK CA MARCHE!

```
void echange(int& a, int& b)
{
    int tmp = a; a = b; b = tmp;
}
```

Concept de surcharge de fonctions

2 fonctions echange de même nom, mais types de paramètres différents
le compilateur sait faire la différence lors d'un appel

```
echange(&x,&y)  version pointeurs
echange(x,y)   version références
```

remarque : les 2 fonctions suivantes ne peuvent cohabiter

```
void echange(int a, int b)
void echange(int& a, int& b)
```

appel ambigu : echange(x, y) quelle fonction echange choisir?

OBJECTIF 2 : minimisation des informations à recopier lors du passage de paramètres

COUTEUX un objet Point est recopié : C et C++

```
void affiche(Point p)
```

PEU COUTEUX, RAPIDE juste un pointeur : C et C++

```
void affiche(Point* p) MAIS DANGER PAS DE CONTRAT DE NON MODIFICATION
void affiche(const Point* p) OK SECURE CONTRAT DE NON MODIFICATION
```

PEU COUTEUX, RAPIDE juste une référence : uniquement C++

```
void affiche(Point& p) MAIS DANGER PAS DE CONTRAT DE NON MODIFICATION
void affiche(const Point& p) OK SECURE CONTRAT DE NON MODIFICATION
```

remarques de style : Point* <==> Point * Point& <==> Point &

Référence comme valeur de retour de fonction

DU JAMAIS VU : f(6) = 12; !!!!

voir exemple support de cours p15

des exemples dans le TP N°1

Fonctions avec des arguments à valeur par défaut

```
void afficher(const Point& p, int couleur=0, const char *nom="Point");
```

appels possibles pour un Point p1

```
affiche(p1) ==> couleur vaut 0 et nom vaut "Point"
```

```
affiche(p1, 124) ==> couleur vaut 124 et nom vaut "Point"
```

```
affiche(p1, 124, "Origine") ==> couleur vaut 124 et nom vaut "Origine"
```

dès qu'une valeur par défaut est précisée, les autres arguments doivent avoir des valeurs par défaut :

```
void afficher(const Point& p, int couleur=0, const char *nom); ILLEGAL:
```

SECOND CONCEPT MAJEUR DU C++ :

LA SURCHARGE DE FONCTIONS

Si pas mêmes noms de fonction

alors

SURCHARGE NON

sinon

si nombre d'arguments différents

alors

SURCHARGE OK

sinon

si type des paramètres différents

alors

SURCHARGE OK

sinon

SURCHARGE NON

- le type de retour de la fonction n'est pas prise en compte lors de la surcharge de fonctions

```
int somme(int n1, int n2) OK
```

```
double somme(int n1, int n2) ERROR : même si types de retour différents
```

```
double somme(double d1, double d2) OK
```

```
int somme(int n1, int n2, int n3) OK
```

```
int somme(const Point & p1, const Point & p2) OK
```

FIN DU COURS Page 20

sans operator+ (plus tard ...)