REFERENCES (1)

➢ Définition

- **Synonyme pour un objet informatique** (une variable, une instance d'une classe...).
- Utilisation indifférente de la référence ou de l'objet informatique initial, quelque soit l'action réalisée.
- La référence et la chose ont même adresse, même valeur ...

➢ Syntaxe

```
type & identificateur = <variable> ;
// identificateur est une variable
// de type « reference vers type»
```

Une référence s'initialise toujours au moment de la déclaration.
On ne peut jamais changer ensuite cette initialisation.

➢ Exemple : variable

```
int main() { int i, *pi=NULL, j=2;
    int & ri = i;          // ri ⇔ i
    ri = 9 ;               // affecte 9 a ri... donc a i !
    ri = j ;               // affecte 2 a ri... donc a i !
    cin >> i;              // ⇔ cin >> ri;
    cout << ri;            // ⇔ cout << i;
    pi = &ri;              // ⇔ pi = &i;
    ri++;                  // Incrémente i
    &ri=j;                 // Interdit de référencer une
                           // autre variable !
}
```

➢ Exemple: Paramètre de fonction

```
// a et b sont des références à des objets
// extérieurs à la fonction, passés à la fonction !
swap( int &a, int &b){ int inter=a; a=b; b=inter; }
int main() { int i,j;
    cin >> i >> j;         // Entrée de i et j
    cout << i << j;        // Affichage de i et j
    swap(i, j);
    // la fonction utilise des paramètres référence.
    // On l'exécute avec des références à i et j
    // La fonction travaille donc bien sur i et j !
    cout << i  << j;       // Affichage de i et j :
    // les valeurs de i et j ont bien été inversées !
}
```

REFERENCES (2)

➢ Référence et pointeurs ? Référence, ou pointeur ?

Les notions de référence et de pointeur sont très proches.

La plupart des compilateurs C++ travaillent en fait dans votre dos avec des pointeurs lorsque votre code utilise des références.

« Quand une référence est utilisée, un pointeur pourrait l'être »

Exemple :

```
int main() { int i;
     int &ri = i; // ri est la même chose que i
     ri =5 ;
     /* Le code équivalent est :
     int *ri_assembleur = &i;
     *ri_assembleur = 5;   */
     return 0;
}
```

Pourquoi alors à la fois des références et des pointeurs en C++ ?

- Dans certains cas, les références permettent un code **beaucoup** plus « naturel » et « lisible ».
- Dans d'autres cas, toujours pour la clarté et la lisibilité du code, il faut au contraire éviter les références et préférer les pointeurs.

Conseils à venir plus tard dans ce cours.

REFERENCES (3)

➢ Remarques

On peut prendre l'adresse d'une référence : c'est l'adresse de la variable référencée :

```
int i = 9 ;
int &ri = i;
int *pi = &ri ;   // pi prend l'adresse... de i !
*pi = 10 ;        // i vaut 10
```

Attention : pas de références à des références, ni de références à des champs de bits, ni de tableaux de références (pb d'initialisation), ni de pointeurs sur des références (une référence n'est pas un objet « physique » : elle n'a pas d'adresse ou valeur au niveau assembleur).

➢ Référence et retour de fonction

Une référence à un objet peut être retournée par une fonction.

=> La fonction renvoie un synonyme de la variable retournée (au lieu d'une valeur)

Cela permet par exemple d'écrire :

```
int & f(int & x) {...; return x ; }
// x est retourné par reference !
f(i)=a+b; //=> modifie x via sa référence !
```

- **Exemple 1**

L'opérateur >> de cin est défini par :

```
istream & operator>> (istream &s, int &i)
{ // code lisant un entier
   return s;
}

ce qui permet     cin >> i;
// Transcrit en istream::>>(cin,i)
cin >> i >> j;
// Transcrit en istream::>>(istream::>>(cin,j),i);
```

REFERENCES (4)

- **Exemple 2**

```
// f(int*, int) retourne une reference sur le premier
// élément nul du tableau supposé existé.
int & f (int *t, int n) {
    for (int i=0; i<n; i++)
      if (t[i]==0)
          return t[i] ;
}
// Le type de retour de la fonction est une
// référence, donc ce qui est retourné n'est pas
// la valeur de t[i] mais une référence à t[i] !

int main() {
    int tab[10]={1,2,3,4,0,5,6,7,8,9};
    for (int i=0; i<10; ) cout << tab[i++]<< " ";
    // Affiche 1,2,3,4,0,5,6,7,8,9

    f(tab,10) = 15580; /* change la valeur du premier
                          élément du tableau qui est nul */
    for (int i=0; i<10; ) cout << tab[i++] << " ";
    // Affiche 1,2,3,4,15580,5,6,7,8,9
    return 0;
}
```

Attention : ne jamais retourner de référence à une variable locale à une fonction !

Ceci serait équivalent à retourner l'adresse d'une variable locale.

```
int & f (int n)  {
    int i;  // i est locale et automatique
    •••;
    return i ;// destruction de i. ERREUR de conception !
}

int main()    { int a,b;
  a=f(10);
    // a est une référence à un objet qui
    // n'existe plus => erreur !
    return 0 ;
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (1)

const : nouveau mot clé pour imposer la « constance » de la valeur d'un objet informatique.

➢ Constantes - Variable const

Deux syntaxes :

```
int main(){
    const int i = 2;          // i est un entier constant
    double const x = 3.14;    // x est un double constant
    i = 3 ;                   // INTERDIT : i est const
    scanf("%d", &i) ;         // => ERREUR : i est const
    printf("%lf\n", x);       // OK
    double pi2 = x * x ;        // OK
    return 0;
}
```

Intérêt : **garantit le typage fort** vérifié à la compilation.

➢ Const et type des variables

Le mot clé "const" fait partie du type d'une variable : une "variable entière const" n'est pas du meme type qu'une "variable entière".
Le caractère const ou non const est vérifié à la compilation.
Exemple de messages d'erreur du compilateur :

```
error: assignment of read-only location
error: invalid conversion from 'const int*' to 'int*'
```

➢ Pointeur const

Comme toute variable, un pointeur peut être constant :
<type> * **const** ptr = <adresse> ;
=> plus le droit de changer la valeur de ptr (l'endroit ou il pointe).

```
int main(){
    int i = 2, j = 3;
    int * const p_i = &i ;  // p_i est un ptr constant
    printf("%d\n", *p_i);   // OK
    *p_i = 8 ;              // OK
    p_i = &j ;              // INTERDIT
    return 0;
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (2)

➢ Pointeur et référence sur une variable const

Deux syntaxes équivalentes :

pointeurs	*références*
<type> **const** * ptr;	<type> **const** & ref = <adresse>;
const <type> * ptr;	**const** <type> & ref = <adresse>;

=> On a pas le droit d'utiliser le pointeur (ou la reference) pour *modifier* la variable

```
int main(){
    int i = 2;
    const int * p_i = &i;
    // equivalent à int const * p_i = &i ;
    // p_i pointe sur int constant
    i = 9 ;                  // OK
    printf("%d\n", *p_i);    // OK
    *p_i = 8 ;               // INTERDIT

    const int & r_i = i;
    // equivalent à int const & r_i = i ;
    // r_i est une référence sur int constant
    printf("%d\n", r_i);     // OK
    r_i = 8 ;                // INTERDIT
    return 0;
}
```

➢ Pointeur, références et vérification de la *constness*

Un pointeur sur variable "const" peut pointer une variable "non const".
Un pointeur sur variable "non const" ne peut pas pointer une variable "const".

```
int main(){
    int i = 2;
    const int * p_i = &i;   // OK
    const int & r_i = i;    // OK
    *p_i = 2 ;              // INTERDIT : *pi read_only
    r_i = 9 ;               // INTERDIT : ri read_only
    const int j = 9 ;
    const int * p_j = &j;   // OK
    int * p_jnonconst = &j; // INTERDIT
    const int & r_j = j;    // OK
    int & r_jnonconst = j;  // INTERDIT
    return 0 ;
}
```

MOT CLE *CONST ET* VARIABLES *CONST* (3)

➢ Paramètre « const pointeur » d'une fonction

La syntaxe *const ptr* pour un paramètre interdit la modification dans la fonction de la variable pointée.

Exemple : paramètre tableau "const"

```
// équivalent à void afficherTab(int const * t, int n)
void afficherTab(const int * t, int n){
    for (int i=0; i<10; ) cout << t[i++]<< " ";
    cout << endl ;
    t[5] = 9 ; // INTERDIT du fait du const
}
```

=> une fonction qui travaille sur un tableau ou un objet *sans le modifier* devrait toujours declarer ce parametre "const" pour signifier, dans le prototype de la foncton, que le tableau (ou l'objet) ne sera pas modifié.

Exemple :

```
// signature de strcpy() ; voir man strcpy
char *strcpy(char* s1, const char* s2);
```

=> une fonction qui travaille sur un pointeur *sans modifier la variable pointée* devrait toujours declarer ce paramètre pointeur const.

➢ Paramètre « const référence » d'une fonction

La syntaxe *const reference* pour un paramètre interdit la modification dans la fonction de la variable référencée.

Exemple :

```
// équivalent à void afficherString(string const & str)
void afficherString(const string & str){
    cout << "str vaut : " << str << endl ; // OK
    str = 8 ; // INTERDIT du fait du const
}
```

Nous reviendrons sur cette notion dans la suite du cours.

MOT CLE *CONST ET* VARIABLES *CONST* (4)

➢ Exemple

Exemple :

```
 #1                    #2           #3
const int * function(const int * const p_i);
```

est équivalent à :

```
     #1                    #2       #3
int const * function(int const * const p_i);
```

#3 indique que le pointeur à gauche est *const* : **dans la fonction**, on ne pourra pas changer l'endroit ou il pointe.

```
// INTERDITS dans le code de la fonction:
p_i = <une autre adresse> ;
```

#2 indique que l'entier pointé est *const.* Permet de savoir que la fonction ne modifiera pas l'entier pointé. D**ans la fonction**, on ne pourra pas changer la valeur de cet entier

```
// INTERDITS dans le code de la fonction:
*p_i = 9 ;
p_i[6] = 2 ;
```

#1 indique que l'adresse retournée par la fonction pointe un entier const : après de l'appel de la fonction, le cone de pourra pas changer la valeur de cet entier.

```
int main() {
    int k = 9 ;
    * ( function(&k) ) = 3;              // INTERDIT
      // car le ptr retourné pointe un entier const :
      // "assignment of read-only location"
    int * ptr1 = function(&k) ;          // INTERDIT
    // invalid conversion from 'const int* const' to 'int*'
    int * const ptr2 = function(&k) ;    // INTERDIT
    const int * ptr3 = function(&k) ;    // OK
    printf("%d\n", *ptr3);               // OK
    *ptr3 = 9;                           // INTERDIT
    return 0 ;
}
```

ECHANGE DE FONCTIONS ENTRE LE C ET LE C++

➢ Fonction C -> appel C++ :

Pour utiliser une librairie compilée avec un compilateur C dans un programme C++, il suffit de déclarer les prototypes des fonctions C avec la directive

`extern "C"`

```
extern "C" { double mafct_C (double); }
int main() {
    double rest = mafct_C(4) ; //appel depuis du code C++
}
```

➢ Handler C qui appelle du code C++

Pour qu'une fonction d'une librairie C puisse utiliser du code C++ : définir de nouvelles fonctions qui feront le lien avec les méthodes d'un objet particulier et imiteront ces méthodes.

Exemple : utiisation de l'algorithme qsort() de la librairie C.
Prototype de qsort (man qsort) :

```
void qsort(void *base, size_t nel, size_t width,
                int (*compar)(const void *, const void *));
```

ou *compar()* est un *pointeur sur une fonction C* qui doit retourner la comparaison des deux arguments passés en paramètre.

```
#include <stdlib.h>
int monComparateur_C(const void * p1, const void * p2) {
    const string * s1 = static_cast<const string *> (p1);
    const string * s2 = static_cast<const string *> (p2);
    return s1->compare(*s2); //methode string::compare()
}

int main() {
    string tab[3];
    tab[0] = "ef"; tab[1] = "ab" ; tab[2] = "cd";
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;

    qsort( tab, 3, sizeof(string), monComparateur_C);
    for(int i = 0 ; i < 3 ; i ++) cout << tab[i] << " " ;
    cout << endl;
    return 0;
}
```

CONVERSION ; CAST

Nouveaux opérateurs de conversion, plus sécurisés et adaptés à la programmation objet.

➢ Conversion prédéfinie : `static_cast<T> (expr)`

- idem anciennes conversions
- conversions résolues à la compilation
- pas de vérifications à l'exécution
- doit être utilisé pour des conversions non-ambiguë

➢ Suppression de la constance : `const_cast<T> (expr)`

- résolu à la compilation
- Exemple :

```
void affStr(string & str ) {
    cout << "Chaine : " << str << endl;
}

void g(const string & s) {
    affStr(s);          // Erreur: s constant
                        // et affStr n'attend pas un const
    affStr(  const_cast< string & >(s)  );  // Ok
}

//Remarque : affStr(), en fait, devrait prendre un
//const string& en paramètre puisqu'elle ne modifie pas s
```

➢ Conversion dynamique : `dynamic_cast<T> (expr)`

- Utilisé sur des pointeurs ou des références dans une hiérarchie de classes

➢ Conversion de pointeur : reinterpret_cast<T> (expr)

- Permet de transformer n'importe quel pointeur en n'importe quel pointeur.
- Pas de vérification à l'exécution.
- Peut être utilisé sur des objets sans liens : dangereux

FONCTIONS INLINE

Une fonction (fonction simple ou fonction membre) déclarée **inline** est expansée par le compilateur : l'appel de la fonction est remplacé par son code.

Efficacité :

- Pas de passage de paramètres sur la pile, pas d'appel de fonction, pas de restauration de la pile.
- Très utile dans les méthodes d'accès aux membres privés.

Déclaration explicite :

```
inline int uneFctInline() { .... } ;
```

Déclaration implicite : les fonctions membres définies dans la déclaration de la classe (=> en général, dans le fichier header), sont inline .

Le corps doit être mis dans le fichier header (.h) et non pas dans le .cpp :

Fichier header MaClasse.h

```
// fonctions C
inline int fac(int n) {return i<2 ? i : i* fac(i-1) }

// fonctions membres
class A {
    //implicite dans la déclaration de la classe
    int f() { return 2 ; }
    int g() ;
} ;

// déclaration explicite de g en inline
inline int A::g() { return 3 ; }
```

Peut être ignorée par le compilateur (en particulier s'il y a des boucles à l'intérieur)
L'mplantation réelle dépend des compilateurs.
Exemple :

```
inline fac(int n) {return i<2 ? i : i* fac(i-1) }
```

L'appel se fait par `fac(5);`
Le compilateur générera :

- soit 720
- soit 6*5*4*3*2*1
- soit 6*fac(5)