

# C++

## Les références

001

## Passage d'argument

Rappel: Les paramètres sont passés par **copies** en C et en C++

```
#include <iostream>
int ma_fonction(int b)
{
    b=b+2;
    return b;
}
int main()
{
    int a=5;
    int c=ma_fonction(a);
    std::cout<<a<<" "<<c<<std::endl;
    return 0;
}
```

b est une copie de a  
(même valeur, emplacement mémoire différent)

modification locale sur b  
n'affecte pas a

c est une copie de b

002

## Passage d'argument

Rappel: Pour modifier un paramètre dans une fonction en C, on passe son **adresse**.

```
#include <iostream>
void ma_fonction(int *b)
{
    *b=*b+2;
}
int main()
{
    int a=5;
    int *pa=&a;
    ma_fonction(pa);
    std::cout<<a<<std::endl;
    return 0;
}
```

b est une copie de pa  
b contient la copie de l'adresse de a  
donc b pointe vers a

on modifie la valeur pointée  
=> on modifie a

003

## Pointeurs = problèmes

Pointeurs vers valeurs non prévues

- Segmentation Fault
- Modification de variables imprévues

Coder (proprement) avec des pointeurs demande beaucoup de contraintes:

- Faire attention à l'initialisation `int *p=NULL;`
- Vérifier si pointeur non NULL `assert(p!=NULL);`
- Changement de la syntaxe du code (beaucoup d'\*) `*p=*p+1;`

Utilisation principale des pointeurs:

- Passer un argument par son adresse pour le modifier `f(int *p) { *p=*p+1; ... }`
- Passer une struct par son adresse const sans la modifier `f(const int* p) { int a=*p; ... }`

Sous-utilisation de la puissance des pointeurs

004

## Les références

C++ introduit la notion de **référence**

Référence ~ alias vers une variable

Référence permet

- Passer un argument et de le modifier dans la fonction appelante
- Passer un argument de manière *const* sans le copier

Avec la même syntaxe que la variable initiale !

```
void ma_fonction(int& b)
{
    b=b+2;
}
int main()
{
    int a=5;
    ma_fonction(a);
    std::cout<<a<<std::endl;
    return 0;
}
```

int& = symbol d'une référence (alias) vers un int  
 Pas d'\*, on a l'impression de manipuler a directement (syntactic sugar)  
 b vaut 7 !

005

## Les références

```
int main()
{
    int a=5;
    int& ref_a=a;

    ref_a=9;
    ref_a++;

    std::cout<<a<<" "<<ref_a<<std::endl;

    return 0;
}
```

ref\_a est un alias sur a  
 modifier ref\_a revient à modifier a

⚠ Symbole des références très mal choisi!

Confusion courante entre

```
int a=8;
int& ref_a=a;
int a=8;
int* ptr_a=&a;
```

l'analyse du symbole "&" dépend du contexte  
 - declaration: reference  
 - devant variable: adresse

006

## Les références: initialisation

Une référence doit toujours être initialisée lors de la déclaration

Permet d'éviter le pointeur ref non initialisé

```
int main()
{
    float a;
    float& ref_a=a; OK

    float& ref_b; KO
    float& ref_c=8; KO
    float& ref_d=NULL;KO

    return 0;
}
```

declared as ref but non initialized  
 invalid initialization of non-const ref of type 'float&' from an rvalue of type 'int'

- rvalue = valeur temporaire ne peut être qu'à droite d'une affectation
- lvalue = variable stockable peut être à gauche d'une affectation

007

## Les références constantes

Une référence constante permet de créer un alias non modifiable sur une autre valeur. Même une rvalue.

```
int main()
{
    float a=4.0f;
    const float& ref_a=a;

    a=a+1.0f;
    ref_a=ref_a+7.0f;

    const float& ref_b;
    const float& ref_c=8.0f;

    return 0;
}
```

ref\_a est un alias constant sur a  
 a peut être modifié. ref\_a est également modifié.  
 ref\_a ne peut pas être modifié par cette intermédiaire  
 ref\_b doit toujours être initialisé  
 ref\_c peut être un alias constant sur la rvalue 8.0f

008

## Intérêt des références

```
struct vec4
{
    double x,y,z,w;
};

void multiply(vec4& v, double s)
{
    v.x *= s;
    v.y *= s;
    v.z *= s;
    v.w *= s;
}

int main()
{
    vec4 v={1.1,7.4,8.8,9.1};

    multiply(v,4.0);

    std::cout<<v.x<<" "<<v.y<<" "<<v.z<<" "<<v.w<<std::endl;

    return 0;
}
```

passage d'une référence

modification de la variable  
comme une variable locale  
pas d\*, pas de ->

009

## Intérêt des références

```
struct vec4
{
    double x,y,z,w;
};

void print(const vec4& v)
{
    std::cout<<v.x<<" "<<v.y<<" "<<v.z<<" "<<v.w<<std::endl;
}

int main()
{
    vec4 v={1.1,7.4,8.8,9.1};

    print(v);

    return 0;
}
```

référence constante  
pas de copie inutile (rapide)  
pas de syntaxe pointeur inutile

010

## Exemple concret de référence

Fonction de multiplication par un scalaire

```
struct vec4
{
    double x,y,z,w;
};

vec4 mul(const vec4& v, double s)
{
    vec4 v_out={v.x*s,v.y*s,v.z*s,v.w*s};
    return v_out;
}

int main()
{
    vec4 v1={1.1,7.4,8.8,9.1};

    vec4 v2=mul(v1,3.5);

    std::cout<<v2.x<<" "<<v2.y<<std::endl;

    return 0;
}
```

évite la copie

011

## Accesseur

Accesseur de type 'get'

```
class vec50
{
private:
    float T[50];
public:
    void init()
    {
        for(int k=0;k<50;++k)
            T[k]=static_cast<float>(k);
    }
    float value(unsigned int i) const
    {
        assert(i<50);
        return T[i];
    }
};

int main()
{
    vec50 v;
    v.init();
    std::cout<<v.value(10)<<std::endl;
    return 0;
}
```

Indique que cette fonction  
ne modifie pas T

012

## Accesneur

### Accesneur de type 'set'

```
class vec50
{
private:
    float T[50];
public:
    void init()
    {
        for(int k=0;k<50;++k)
            T[k]=static_cast<float>(k);
    }
    float& value(unsigned int i)
    {
        assert(i<50);
        return T[i];
    }
};

int main()
{
    vec50 v;
    v.init();

    v.value(10)=15;
    v.value(10)++;

    std::cout<<v.value(10)<<std::endl;

    return 0;
}
```

retourne une référence!

On peut manipuler la valeur de l'extérieur

Utile pour des vecteurs/matrices!  
m(4,8)=18;  
Equivalent Java:  
m.set(4,8,18); ou m.set(18,4,8);

013

## Précautions avec les références

Une référence peut devenir invalide si la variable n'existe plus.  
(Même problème qu'avec les pointeurs)

```
int& f(int a)
{
    int b=2*a;
    return b;
}

int main()
{
    int a=10;
    int& ref=f(a);

    std::cout<<ref<<std::endl;

    return 0;
}
```

b est une variable locale

retourne une ref sur une variable locale (Warning)

variable locale n'existe plus  
Fuite mémoire, segfault, etc..

014

## Limiter les références non const

Ne pas abuser des modifications de paramètres par référence  
Difficile à lire

```
int f(int a,int& b,int c,int& d,int e,int f)
{
    b++;d++;e+=a+c+f;
}

int main()
{
    int x0=1,x1=3,x2=8,x3=9,x4=12,x5=15;
    f(x0,x1,x2,x3,x4);
    return 0;
}
```

*Qui est modifié?*

Il faut des règles de programmations  
Il faut éviter tout doute sur les variables modifiées

- Ex.
- Documenter le code
  - Références in/out en début/fin de fonction
  - Variables modifiées passées par pointeurs

...

015