

21 sept. 04

Rappel C++

C → Structure – Champs (pas de nouveau type par défaut)

C++ → Langage orienté objet → classe → données : membres – variables - membre
 → fonctions : méthodes – fonctions – membre

classe

mot clef : *class* (par défaut, accès privé)

- Modificateur d'accès : public,
private,
protected.
- Méthodes (2) : → constructeur
→ destructeur

Syntaxe : constructeur : * Même nom que la classe

* Pas de type de retour

Syntaxe : destructeur :

* Même nom que la classe + '~'

* Pas de type de retour

Ex :

```
#include <string.h>
class fiche
{
    public : ← modificateur d'accès
        char nom[50] ;
        char prenom [50] ;
};
void main ()
{
    fiche a ;
    strcpy (a.nom, »test ») ;
}
```

- Appels : implicites
 - Constructeur appelé lors de la création de l'objet.
 - Destructeur appelé lors de la destruction de l'objet.
 - Surcharge :
 - opérateurs :
 - surcharge est vue comme une méthode de la classe
 - surcharge est vue comme une fonction amie
- ex : classe permettant la gestion des nombres complexes. Avec *operator* surcharge de l'opérateur * et =.

```
//prog nonoff 21 sept 2004
#include <stdio.h>
#include <iostream.h>
#include <conio.h>
class COMPLEXE
{
    public:
        COMPLEXE (int reel=1, int imaginaire=1);
        COMPLEXE (COMPLEXE &c);
```

```

    friend COMPLEXE operator * (COMPLEXE C1, COMPLEXE C2);
    friend int operator == (COMPLEXE C1, COMPLEXE C2);
    void affiche();
private:
    int nreel;
    int nimaginaire;
};

COMPLEXE::COMPLEXE (int reel, int imaginaire)
{
    nreel=reel;
    nimaginaire=imaginaire;
}

COMPLEXE::COMPLEXE(COMPLEXE &c):nreel(c.nreel),nimaginaire(c.nimaginaire)
{
}

COMPLEXE operator * (COMPLEXE C1, COMPLEXE C2)
{
    C1.nreel = C1.nreel * C2.nreel;
    C1.nimaginaire = C1.nimaginaire * C2.nimaginaire;
    return C1;
}

int operator == (COMPLEXE C1, COMPLEXE C2)
{
    if( ( C1.nreel == C2.nreel) && (C1.nimaginaire == C2.nimaginaire) )
        return 1;
    return 0;
}

void COMPLEXE::affiche ()
{
    cout<<"Reel ="<<nreel<<endl<<"Imaginaire ="<<nimaginaire<<endl;
    getch();
}

void main()
{
    int reel1, reel2, imaginaire1, imaginaire2;
    int comp;
    cout<<"Entrez reel1, imaginaire1, reel2, imaginaire2"<<endl;
    cin>>reel1>>imaginaire1>>reel2>>imaginaire2;
    COMPLEXE c1(reel1,imaginaire1);
    COMPLEXE c2(reel2,imaginaire2);
    COMPLEXE c3=c1*c2;
    c3.affiche();
    comp= (c1==c2);
    if (comp)

```

```

    cout<<"C1 et C2 sont egaux"<<endl;
else
    cout<<"C1 et C2 sont diff,rents"<<endl;
getch();
}

```

- fonctions : plusieurs fonctions peuvent posséder le même nom.
 - Nombre de paramètre
 - Type de paramètre
- Sachant qu'une méthode est une fonction, on peut surcharger les méthodes (dont le constructeur).

Ex :

```

class fiche
{
    char nom [50] ;
    char prenom [50] ;
public :
    fiche(char*nom, char*prenom) ;
}
void main ()
{
    fiche a ;
    fiche b (« Dupont », « Bernard ») ;
}
fiche ::fiche(char*nom,char*prenom)
{
    strcpy(this->nom,nom) ;
    strcpy(this->prenom,prenom) ;
}
fiche ::fiche()
{
    strcpy(this->nom, »nomnom ») ;
    strcpy(this->prenom, »nomnom ») ;
}

```

- Héritage
 - Simple :
 - A : classe de base
 - ↓ ← modifieur : public, private, protected
 - B : classe dérivée
- Ex :

```

class etudiant
{
//protected
    char nom[50] ;
    char prenom[50] ;
public :
    etudiant (char*nom, char*prenom) ;
    void print() ;
};

```

```

class 3info : public etudiant
{
    char stage[50];
public:
    3info(char*nom,char*prenom,char*stage);
}
3info::3info(char*nom,char*prenom,char*stage):etudiant(nom,prenom)
{
    strcpy(this->stage,stage);
}
etudiant::etudiant(char*nom,char*prenom)
{
    strcpy (this->nom,nom) ;
    strcpy (this->prenom,prenom) ;
}
void 3info ::print()
{
    printf(“%s\n”,nom);
    printf(“%s\n”,prenom);
    printf(“%s\n”,stage);
}
//ou
    etudiant::printf();
}
//ou
void etudiant::print()
{
    printf(“%s\n”,nom);
    printf(“%s\n”,prenom);
    printf(“%s\n”,stage);
}

```

Appel !

ne marche que si l'on met *protected*.

- Multiple : pas en C# !