



## Tests/débugage Notions avancées de programmation

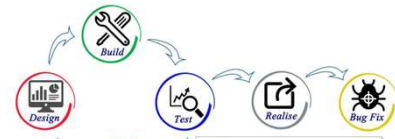
- Test de code (test de non régression)
- Valgrind
- Module Image (voir TD)
- Arguments de main
- Début du C++ avancé
  - Operator
  - Notion de template → STL
  - Notions d'héritage

<http://licence-info.univ-lyon1.fr/LIFAPCDA>

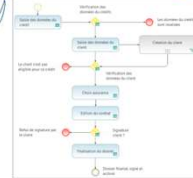
Alexandre Meyer

1

## Vérification de code, test de non-régression



Conception visuelle des  
scénarios de test en  
projet agile



2

2

## Valgrind

- Valgrind
  - <http://valgrind.org/>
  - 6 outils
    - Outils de vérification mémoire
    - Profiler : graphe d'appel
    - Profiler : cache et saut
    - Profiler : tas
    - Thread : detection d'erreur

3

3

## Valgrind : exemple 1

// exemple.cpp : g++ -g exemple.cpp -o myprog

```
1 #include <stdlib.h>
2
3 void f(void)
4 {
5     int* x = new int[10];
6     x[10] = 614;
7 }
8
9 int main(void)
10 {
11     f();
12     return 0;
13 }
```

Problème?



4

4

## Valgrind : exemple 1

// exemple.cpp : g++ -g exemple.cpp -o myprog

```
1 #include <stdlib.h>
2
3 void f(void)
4 {
5     int* x = new int[10];
6     x[10] = 614; // problème 1: débordement
7 } // problème 2: memory leak -- x not freed
8
9 int main(void)
10 {
11     f();
12     return 0;
13 }
```



5

5

## Valgrind : exemple 1

- Compiler avec option de debug : g++ -g
- valgrind --leak-check=yes myprog arg1 arg2

```
==19182== Invalid write of size 4
==19182==   at 0x804838F: f (exemple.c:6)
==19182==   by 0x80483AB: main (exemple.c:11)
→ Écriture en mémoire non allouée, ligne 6
```

```
==19182== Address 0x1BA45050 is 0 bytes after a block of size 40 alloc'd
==19182==   at 0x1B8FF5CD: malloc (vg_replace_malloc.c:130)
==19182==   by 0x8048385: f (exemple.c:5)
==19182==   by 0x80483AB: main (exemple.c:11)
→ Mémoire non libérée, allouée en ligne 5
```

6

6

## Valgrind : exemple 2

```
// exemple2.cpp : g++ -g exemple.cpp -o myprog
1 #include <stdlib.h>
2
3 int* p = new int[10];
4 for(int i=0;i<10;++i)
5     p[i] = i;
6 delete[] p;
7
8
9 for(int i=0;i<10;++i)
10     cout<<"p["<<i<<"]="<<p[i]<<endl;
11
```

Affichage ?

7

## Valgrind : exemple 2

```
// exemple2.c : gcc -g exemple.c -o myprog
1 #include <stdlib.h>
2
3 int* p = new int[10];
4 for(int i=0;i<10;++i)
5     p[i] = i;
6 delete[] p;
7
8
9 int *x = new int[20];
10 for(int i=0;i<10;++i)
11     cout<<"p["<<i<<"]="<<p[i]<<endl;
12
13
```

Et là ?

8

7

8

## Valgrind : exemple 2

- 2 exemples précédents
  - Affiche le tableau car la zone mémoire est simplement marquée à disponible au moment du free
- Valgrind détecte ce type d'erreur

9

9

## Valgrind

- Valgrind
  - Plus de chose à expérimenter en TD/TP
  - Prenez le temps de bien regarder, vous en gagnerez plus tard
  - Voir la doc : <http://valgrind.org/docs/manual/manual.html>
- Attention avec des lib externes
  - Souvent elles produisent des erreurs avec Valgrind
  - Les msg peuvent être long
  - Difficile d'extraire ses erreurs de celles des libs
  - s'habituer au msg de valgrind sur des prog courts

10

10

## Tests de (non) régression

- Tests de régression : à chaque fois que le logiciel est modifié, s'assurer que "les choses qui fonctionnaient avant fonctionnent toujours"
- Pourquoi modifier le code déjà testé ?
  - correction de défaut
  - ajout de fonctionnalités
- Quand ?
  - en phase de maintenance / évolution
  - ou durant le développement

11

11

## TestRegression : Module

- Dans chaque module vous ajouterez une fonction de test *modTestRegression* qui vérifie
  - des éléments/constantes de base
    - par ex. `var_taille>=0`
  - des fonctionnements
    - Appel une fonction
    - Vérifie qu'elle a bien fait ce qu'elle prétend
- Utilisez des assert
- Cette fonction
  - Sera longue car de nombreux tests sont toujours à faire
  - peut bien sûr être découpée en plusieurs sous-fonctions
- Chaque module viendra avec un exécutable qui appelle cette fonction

12

12

## Exo : Ecrivez la fonction de TestRegression qui

- teste un module Nombre Complexe
- teste un module TabDyn
  - plusieurs Ajout à un TabDyn
  - plusieurs suppression à un TabDyn
  - la fonction qui trie un TabDyn
- teste un module ListeChaine
- teste un module ArbreBinaireDeRecherche

13

## Test régression : module NbComplexe

```
class Complex
{
private:
    float x,y;
public:
    NbComplexe(const float re, const float im);
    ...
    bool estReel();
    bool estImaginaire();
    void testRegression();
};
```

14

13

14

## Test régression : module TabDyn

```
class TabDyn
{
private:
    Element* tab;
    int taille_u, taille_alloc;
public:
    TabDyn();
    void ajouter(const Element& e);
    const Element& getConst(int i) const;
    Element& get(int i);
    int taille();
    void supprimer(int i);
    int trouver(const Element& );
    void trier(); // suppose que on sait comparer Element
    ~TabDyn();
    void testRegression();
};
```

15

15

## void TabDyn::testRegression()

```
{
    TabDyn td;
    Element e;

    assert( td.taille_u==0 );
    assert( td.taille()==td.taille_u );
    assert( td.tab==NULL);

    td.ajouter( e );
    assert( td.taille()==1 );
    assert( td.tab[0] == e );
    assert( getConst(0) == e );
    assert( get(0) == td.tab[0] );
    ...
}
```

16

16

## void TabDyn::testRegression()

*... on teste toutes les fonctions ...*  
*... + faire un test avec 10000 insertions, des suppressions ...*

```
}

-----
// TabDyn_TestRegression.cpp
#include <TabDyn.h>
int main()
{
    TabDyn td;
    td.testRegression();
    return 0;
}
```

17

17



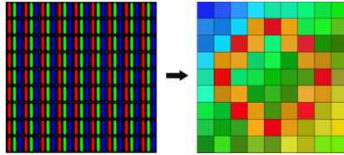
## Vers du C++ avancé ...

- Passage de paramètres
- Constructeur et initialisation de paramètres
- Arguments de main
- Operator en C++
- Notion de template
- STL
- Notions d'héritage

18

## Retour sur le module Image

Lien entre ces cours et la réalisation en TD et TP



19

## Module Image

- En TD machine réalisation d'une classe Image
  - Une image est 2D est un tableau 2D de Pixel
  - Pixel contient 3 composantes R,G,B (unsigned char)
  - En C++, new ne permet que de faire des allocations 1D

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

Pixel = R,G,B

20

## Module Image

- En TD machine réalisation d'une classe Image
  - Une image est 2D, c'est un tableau 2D de Pixel

```
class Pixel
{
private:
    unsigned char r,g,b;
public:
    Pixel(unsigned char rr, unsigned gg, unsigned char bb)
        : r(rr), g(gg), b(bb) {}
};

class Image
{
private:
    int dmix, dimy;
    Pixel* pix;
public:
    ...
}
```

21

## Module Image

- En TD machine réalisation d'une classe Image
  - Une image est 2D, c'est un tableau 2D de Pixel
  - En C++, new ne permet que de faire des allocations 1D
- Il faut donc avoir une classe Image avec la bonne interface
  - setPixel(x,y,pix)
  - Pixel getPixel(x,y)
  - ces deux accesseurs font la conversion (x,y) → indice du tableau

	0	1	2	3	4	5	6	7
0	0	1	2	3	4	5	6	7
1	8	9	10	11	12	13	14	15
2	16	...	...	...	...	...	...	...
3								
4								
5								
6								
7								

Indice = y x dimx + x

22

## Module Image

- En TD machine réalisation d'une classe Image

```
class Pixel
{
private:
    unsigned char r,g,b;
public:
    Pixel(unsigned char rr, unsigned gg, unsigned char bb)
        : r(rr), g(gg), b(bb) {}
};

class Image
{
private:
    int dmix, dimy;
    Pixel* pix;
public:
    const Pixel& getPixel(int x, int y) const;
    Pixel& getPixel(int x, int y);
    void setPixel(int x, int y, const Pixel& p);
    ...
}
```

23

## Module Image

- En TD machine réalisation d'une classe Image

```
class Pixel
{
private:
    unsigned char r,g,b;
public:
    Pixel(unsigned char rr, unsigned gg, unsigned char bb)
        : r(rr), g(gg), b(bb) {}
};

class Image
{
private:
    int dmix, dimy;
    Pixel* pix;
public:
    const Pixel& getPixel(int x, int y) const;
    Pixel& getPixel(int x, int y);
    void setPixel(int x, int y, const Pixel& p);
    ...
}
```

**Remarquer**

- Les initialisations de r,g,b
- les const
- Les passages de paramètres

24

19

20

21

22

23

24

## Module Image

- Pile et tas
- Exemple

```
class Pixel ...
class Image ...

int main()
{
    Image im(128,128);
    Image* im2 = new Image(128,128);
}
```

- Quelle différence ?

25

## Module Image

- Faire des vérifications dans le code

```
void setPixel(int x, int y, const Pixel& p)
{
    assert( x>=0 && y>=0 );
    assert( x<dimx && y<dimy );
    ...
}
```

- Tester les exécution avec Valgrind !!!
  - Le seul outil qui fasse des vérifications mémoires même quand le code semble marcher

26

26

## Module Image

- Pile et tas
- Exemple

```
class Pixel ...
class Image ...

int main()
{
    Image im(128,128);
    Image* im2 = new Image(128,128);
}
```

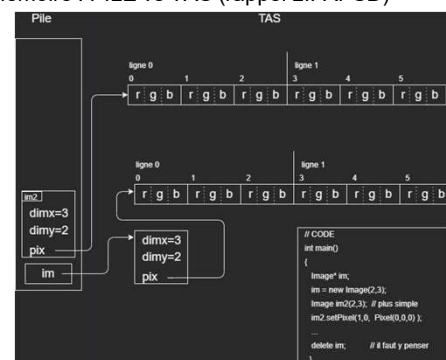
- Quelle différence ?

- **im** est sur la pile
  - la classe Image uniquement, pas le tab de Pixel
- Le pointeur **im2** est sur la pile
  - La classe Image pointée par im2 est sur le tas
  - Le tab de Pixel est sur le tas
- Dans les deux cas le tableau de pixel est sur le tas
- En C++ on choisit si pile ou tas
- Sauf cas particulier, im2 n'est pas la bonne pratique

27

## Module Image

- Mémoire : PILE vs TAS (rappel LIFAPSD)



28

28

## Module Image

- Mémoire : PILE vs TAS (rappel LIFAPSD)
- Accesseur / Mutateur
  - **Ne pas avoir d'accesseur/mutateur partout, tout le temps**
  - Raisonner plutôt en action de haut niveau
  - Quelles actions j'aimerais faire sur la classe

29

29

## Module Image

- Mémoire : PILE vs TAS (rappel LIFAPSD)
- Accesseur / Mutateur
  - **Ne pas avoir d'accesseur/mutateur partout, tout le temps**
  - Raisonner plutôt en action de haut niveau
  - Quelles actions j'aimerais faire sur la classe
- class Image
  - On donne accès à des actions (pas aux données !!!)
  - void resize(int new\_dimx, int new\_dimy);
  - void rectangle(...)
  - void setPix / getPix → **fait les vérifications qu'il convient !**

30

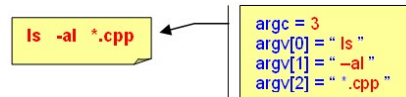
30

## Module Image

- Pour le module Image (et votre projet) de vous devrez écrire du code
  - **'propre'**, lisible et maintenable dans le temps par toutes votre équipe de développement
    - Passage de paramètres avec ( const TYPE & param)
    - Fonctions avec le décorateur Const à la fin
    - Initialisation des données avec constructeur
    - Destructeur
  - **Documenté** (avec doxygen par exemple), voir l'autre cours
    - Faites le dès le début sinon cela ne sert à rien
  - **Testé**
    - avec Valgrind (gestion mémoire)
    - par des tests de (non-)regression → chaque module a sa longue fonction de test

31

## Arguments de main



32

## Démo d'un bash

- ls -a

```
alex@nohu(:):~$ ls -a
./          .cache/    .ipython/
../         .conda/    .jupyter/
.AWK_d/     .condarc*  .keras/
.anaconda/  .config/   .lessht
.aptcyp/    .dbus/     .linphone-history.db
.bash_history .gitconfig* .linphonerc
.bashrc@    .gvfs/     .local/
alex@nohu(:):~$ cp texte.cpp src
```

- mkdir src
- vi truc.cpp
- cp truc.cpp src

33

## Arguments de main

- int main ()
- **int main (int argc, char \*argv[])**
  - Le plus classique
    - ./prog histoire.txt
  - Permet de récupérer les arguments d'un appel du programme depuis le shell
- int main (int argc, char \*argv[],char \*\*envp)
  - Idem mais ajoute les variables du shell

34

## Arguments de main

- **int main (int argc, char \*argv[])**
  - ./prog histoire.txt
  - Permet de récupérer les arguments d'un appel du programme depuis le shell

argc

contient le nombre d'arguments qui ont été passés lors de l'appel du binaire exécutable (nombre de mots dans la ligne de commande) ;

argv ...

35

## Arguments de main

**int main (int argc, char \*argv[])**

argc : nombre d'arguments

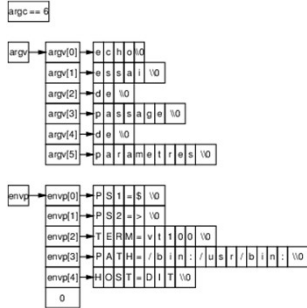
argv : contient les arguments de la ligne de commande. Ces arguments sont découpés en mots par le shell et chaque mot est référencé par un pointeur dans le tableau. Il y a toujours au moins un argument qui correspond au nom du binaire exécutable appelé ;

- le nombre de pointeurs valides dans le premier tableau est donné par le contenu de la variable entière (nombre d'arguments argc) ;
- la chaîne pointée par la première entrée du tableau argv contient le nom de la commande elle-même.

36

## Arguments de main

- ```
• int main (int argc, char *argv[],char **envp)
> echo essai de passage de parametres
```



37

## Arguments de main

```
int main(int argc, char** argv)
{
    int i;
    cout<<"nb arg="<<argc<<endl;

    for(i=0;i<argc;i++)
    {
        cout<<i<<" "<<argv[i]<<endl;
        if (strcmp(argv[i], "bonjour")==0)
            cout<<"BONJOUR"<<endl;
    }

    return 0;
}
```

38

## Arguments de main

```
#include <stdio.h>

int main (int argc, char *argv[],char **envp)
{
    int i;
    printf("Nous avons %d arguments : \n",argc);
    for(i=0;i<argc;i++)
        printf("argument %d = %s \n",i,argv[i]);

    printf("Nous affichons les 5 premieres variables d'environnement \n");
    printf("mais nous les comptons toutes. \n");

    i=0;
    while(envp[i])
    {
        if(i<5) printf("Envp[%d] : %s \n",i,envp[i]);
        ++i;
    }
    printf("Il y a %d variables d'environnement : \n    , i);
    return 0;
}
```

39

# Les operateurs en C++

operator+  
operator-  
...

40

# Operator+ en C++

```
class Complex
{
Public:
    Complex(float _x=0, float _y=0) : x(_x), y(_y) {}
private:
    float x,y;
};

int main()
{
    Complex a(1,2);           // a est le complexe 1+2i
    Complex b;                 // b est initialisé avec (0,0)
```

41

## Et en C++

```
Complex comAdd(Complex a, Complex b)
{
    Complex r;
    r.x = a.x+b.x;
    r.y = a.y+b.y;
    return r;
}

int main()
{
    Complex c;
    c = comAdd(c1,c2);

    // Il serait vraiment pratique
    // c = c1 + c2;
}
```



42

## C++ et les operator+, ...

```
Complex operator+(const Complex& a, const Complex& b)
{
    Complex r;
    r.x = a.x+b.x;
    r.y = a.y+b.y;
    return r;
}

int main()
{
    Complex c1(2,5);
    Complex c2(1,1);
    Complex c;
    c = c1 + c2;    // On peut grâce à la
                  // fonction particulière
                  // opérateur+
    ...
}
```

43

## C++ et l'operator-

```
...
Complex operator-(const Complex& a, const Complex& b)
{
    Complex r;
    r.x = a.x - b.x;
    r.y = a.y - b.y;
    return r;
}

int main()
{
    Complex c1(2,5);
    Complex c2(1,1);
    Complex c;
    c = c1 + c2 + c1 - c1; // C'est bien pratique
    ...
}
```

44

## C++ et l'operator\*

```
...
Complex operator*(float a, const Complex& b)
{
    Complex r;
    r.x = a * b.x;
    r.y = a * b.y;
    return r;
}

int main()
{
    Complex c1(2,5);
    Complex c2(1,1);
    Complex c;
    c = c1 - 5.0f * c2;    // C'est bien pratique
    // attention : c = c2 * 5.0f ne marchera pas !!!
    // car les arguments sont (complex, float)
    ...
}
```

45

## C++ et l'operator\*

```
...
Complex operator*(Complex a, Complex b)
{
    Complex r;
    r.x = a.x*b.x - a.y*b.y;
    r.y = a.x*b.y + a.y*b.x;
    return r;
}

int main()
{
    Complex c1(2,5);
    Complex c2(1,1);
    Complex c;
    c = c1 * c2;    // C'est bien pratique
    ...
}
```

46

## operator<< en C++

```
...
Class Complex
{
    ...
public:
    friend ostream& operator<<(ostream& os, const Complex& c);
};

ostream& operator<<(ostream& os, const Complex& c)
{
    os<<c.x<<" + i."<<c.y;
    return os;
}

int main()
{
    Complex c1(2,5);
    cout<<c1<<endl;
    ...
}
```

47

## Notion de template<> en C++

48



## C++ mécanisme de template

```
int max(int a, int b)
{
    if (a>b) // opérateur de comparaison
        return a;
    else
        return b;
}

float max(float a, float b)
{
    if (a>b) // opérateur de comparaison
        return a;
    else
        return b;
}
```



Réécriture de  
quasiment la même chose

49

## C++ mécanisme de template

Pire encore pour les conteneurs : TabDyn, Liste, File, etc.

```
class TabDynEntier
{ ...
private:
    int taille_allouee, taille_utilisee;
    int *p;
};

class TabDynCamion
{ ...
private:
    int taille_allouee, taille_utilisee;
    Camion *p;
};
```



Réécriture de  
quasiment la même chose

50

## C++ mécanisme de template

// Définition de fonction template

```
template <typename T> T max(T a, T b)
{
    T res;
    if (a>b) // opérateur de comparaison
        return a;
    else
        return b;
}

int main()
{
    int mi = max( 5, 8 );
    float mf = max(5.8f, 8.234f);
}
```



51

## C++ mécanisme de template

définition de fonction template :

```
template <typename T> T max(T a, T b)
{
    ...
}

int main()
{
    int i = max( 12, 34);           // max<int>()
    float f = max( 13.56f, 0.223f); // max<float>()
    Etudiant c = max(
        Etudiant("Albert", 14),
        Etudiant("Albert", 18) );
    // marche si la class Etudiant sait
    // répondre à a>b
}
```



52

## C++ mécanisme de template

• Avec des classes

```
template <typename T> class TabDyn
{
private:
    int taille_allouee, taille_utilisee;
    T* tab;
};

int main()
{
    TabDyn<int> tab;
    TabDyn<Camion> tabC;
}
```

53



Templates, Containers et STL

une bibliothèque C++, normalisée par l'ISO (document  
ISO/CEI 14882) et mise en œuvre à l'aide des



54

## STL



- Structures de données de base
  - tableau, liste chaînée file, pile, hashtable collection, ensemble, etc.
- 2 caractéristiques
  - Ne pas recoder à chaque fois
  - Indépendant du type/classe contenu

55

## STL



- Comportements **génériques** décrits de manière algorithmique
  - dans une pile d'entiers, on empile, on dépile
  - dans une pile d'objets de classe A, on empile, on dépile
- même algorithme qui effectue cela !
- Idem pour tableau dyn, liste chaînée, etc.

56

## STL



- un ensemble de classes conteneurs, telles que les vecteurs (vector), les tableaux associatifs (map), les listes chaînées (list), qui peuvent être utilisées pour contenir n'importe quel type de données à condition qu'il supporte certaines opérations comme la copie et l'assignation.
- une abstraction des pointeurs : les itérateurs. Ceux-ci fournissent un moyen simple et élégant de parcourir des séquences d'objets et permettent la description d'algorithmes indépendamment de toute structure de données.
- des algorithmes génériques tels que des algorithmes d'insertion/suppression, recherche et tri.
- une classe string permettant de gérer efficacement et de manière sûre les chaînes de caractères.

57

## #include<ctime>

```
clock_t t;

t = clock();
printf ("Operation à chronométrer...\n");
t = clock() - t;

cout <<"Temps en clicks="<<t<<endl;

cout<<"Temps en secondes="
    <<((float)t)/CLOCKS_PER_SEC
    <<endl;
```

58

## #include<cstring> std::string

```
string s1 = "Anatoliy";
cout << "s1 is: " << s1 << endl;

// first argument C string, second number of characters
string s4 (line,10);
cout << "s4 is: " << s4 << endl;

// 1 - C++ string, 2 - start position, 3 - number of characters
string s5 (s3,6,4); // copy word 'line' from s3
cout << "s5 is: " << s5 << endl;

// 1 - number characters 2 - character itself
string s6 (15,*);
cout << "s6 is: " << s6 << endl;

// you can instantiate string with assignment
s6 = "Anatoliy";
cout << "s6 is: " << s6 << endl;      + concat + substring + etc.
```

59

## Tableau dynamique : vector

```
std::vector<int> mon_vecteur;
mon_vecteur.push_back(4);
mon_vecteur.push_back(2);
mon_vecteur.push_back(5);

// Pour parcourir un vector (même const) on peut utiliser les iterators ou les index
for(std::size_t i=0;i<mon_vecteur.size();++i)
    std::cout << mon_vecteur[i] << ' ';

std::vector<int> mon_vecteur(5,69);
// crée le vecteur 69,69,69,69,69

v[0] = 5;
v[1] = 3;
v[2] = 7;
v[3] = 4;
v[4] = 8;
```



60

## Liste chaînée



```
std::list<int> ma_liste;
ma_liste.push_back(4);
ma_liste.push_back(5);
ma_liste.push_back(4);
ma_liste.push_back(1);

for(lit=ma_liste.begin();lit!=ma_liste.end();++lit)
    std::cout << *lit << " ";
std::cout << std::endl;
```

61

## Ensemble : set



```
std::set<int> s;
// équivaut à std::set<int, std::less<int> >

s.insert(2); // s contient 2
s.insert(5); // s contient 2 5
s.insert(2); // le doublon n'est pas inséré
s.insert(1); // s contient 1 2 5

for(sit=s.begin();sit!=s.end();++sit)
    std::cout << *sit << ' ';
std::cout << std::endl;
```

62

## #include <stack>

```
1 // stack::empty
2 #include <iostream> // std::cout
3 #include <stack> // std::stack
4
5 int main ()
6 {
7     std::stack<int> mystack;
8     int sum (0);
9
10    for (int i=1;i<=10;i++) mystack.push(i);
11
12    while (!mystack.empty())
13    {
14        sum += mystack.top();
15        mystack.pop();
16    }
17
18    std::cout << "total: " << sum << '\n';
19
20    return 0;
21 }
```

63

## Programmation Orientée Objets Notion d'héritage

Nécessaire pour introduire Qt

64

## Classes et objets

- Les classes sont les éléments de base de la programmation orientée objet (POO) en C++. Dans une classe, on réunit
  - des données variables : les données membres, ou les attributs de la classe
  - des fonctions : les fonctions membres, ou les méthodes de la classe
- Une classe A apporte un nouveau type A ajouté aux types (pré)définis de base par C++
- Une variable 'a' créée à partir du type de classe A est appelée une instance (ou objet) de la classe A.

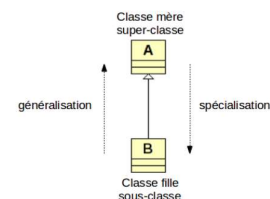
```
class A {
protected:
    float poids;
    bool a_afficher;
Public:
    void print() const;
};

A a1;
A a2,a3;
```

65

## Héritage

- L'héritage (ou spécialisation, ou dérivation)
  - permet d'ajouter des propriétés à une classe existante pour en obtenir une nouvelle plus précise
  - "un B est un A avec des choses en plus"

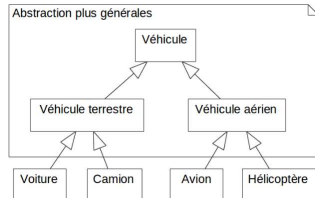


- Exemple : Un étudiant est une personne, et a donc un nom (et un prénom, ...). De plus, il a un numéro INE

66

## Héritage

- L'héritage permet
  - la réutilisation du code déjà écrit
  - l'ajout de nouvelles fonctionnalités
  - la modification d'un comportement existant (redéfinition)



67

## Héritage : un exemple

```

class Personne // Classe de base
{
private:
    string nom;
public:
    string getNom() const;
    void afficher() const
    {
        cout << nom << endl;
    }
};

class Etudiant : public Personne // Classe dérivée
{
private:
    string ine;
public:
    string getINE() const;
    // ajout de nouvelles fonctionnalités
    void afficher() const // modification d'un comportement existant
    {
        Personne::afficher();
        // réutilisation du code déjà écrit
        cout << ine << endl;
    }
};
  
```

68

## Types d'héritage

Remarque : Les constructeurs, le destructeur, de même que l'opérateur= de la classe de base ne sont pas hérités dans la classe dérivée.

| mode de dérivation | Statut dans la classe de base | Statut dans la classe dérivée |
|--------------------|-------------------------------|-------------------------------|
| public             | public                        | public                        |
|                    | protected                     | protected                     |
|                    | private                       | inaccessible                  |
| protected          | public                        | protected                     |
|                    | protected                     | protected                     |
|                    | private                       | inaccessible                  |
| private            | public                        | private                       |
|                    | protected                     | private                       |
|                    | private                       | inaccessible                  |

69

## Héritage : un exemple

```

class Personne // Classe de base
{
private:
    string nom;
public:
    string getNom() const;
    void afficher() const
    {
        cout << nom << endl;
    }
};

class Etudiant : public Personne // Classe dérivée
{
private:
    string ine;
public:
    string getINE() const;
    // ajout de nouvelles fonctionnalités
    void afficher() const // modification d'un comportement existant
    {
        Personne::afficher();
        // réutilisation du code déjà écrit
        cout << ine << endl;
    }
};
  
```

70

## Redéfinition et la surcharge

- Il ne faut pas mélanger la redéfinition et la surcharge
  - Une surdéfinition ou **surcharge** (overloading) permet d'utiliser plusieurs méthodes qui portent le même nom au sein d'une même classe avec une signature différente
  - Une **redéfinition** (overriding) permet de fournir une nouvelle définition d'une méthode d'une classe ascendante pour la remplacer. Elle doit avoir une signature rigoureusement identique à la méthode parente.
- Un objet garde toujours la capacité de pouvoir redéfinir une fonction pour la réécrire ou la compléter

71

## Surcharge

```

class Personne // Classe de base
{
private:
    string nom;
public:
    string getNom() const;
    void afficher() const
    {
        cout << nom << endl;
    }
    void afficher(int n) const // Surcharge
    {
        for(int i=0; i<n; i++) cout<<" ";
        cout << nom << endl;
    }
};
  
```

72

## Redéfinition

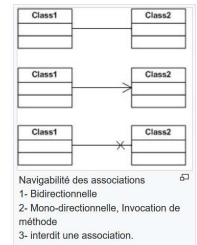
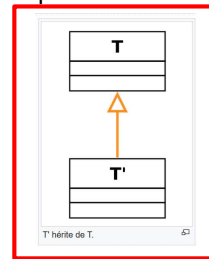
```
class Personne // Classe de base
{
private:
    string nom;
public:
    string getNom() const;
    void afficher() const
    {
        cout << nom << endl;
    }
};

class Etudiant : public Personne // Classe dérivée
{
private:
    string ine;
public:
    string getINE() const;
    void afficher() const // redéfinition
    {
        Personne::afficher();
        // réutilisation du code déjà écrit
        cout << ine << endl;
    }
};
```

73

## Héritage et UML

- Dans un diagramme des classes
  - Flèche avec un triangle
  - Ne pas confondre avec une association

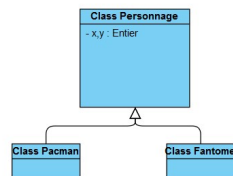


[https://fr.wikipedia.org/wiki/Diagramme\\_de\\_classes](https://fr.wikipedia.org/wiki/Diagramme_de_classes)

74

## Pacman et héritage

- Tous les personnages partagent des données et fonctionnalités
  - Pacman
  - Fantome
 spécialisent leur comportement



75



## Conclusion

**Beaucoup de choses à expérimenter durant le projet**

- Test de code (non regression) / Mémoire Valgrind
- Arguments de main
- C++ avancé
  - operator+, -, \*, <, ==, etc.
  - Notion de template
  - STL
  - Notions d'héritage

76