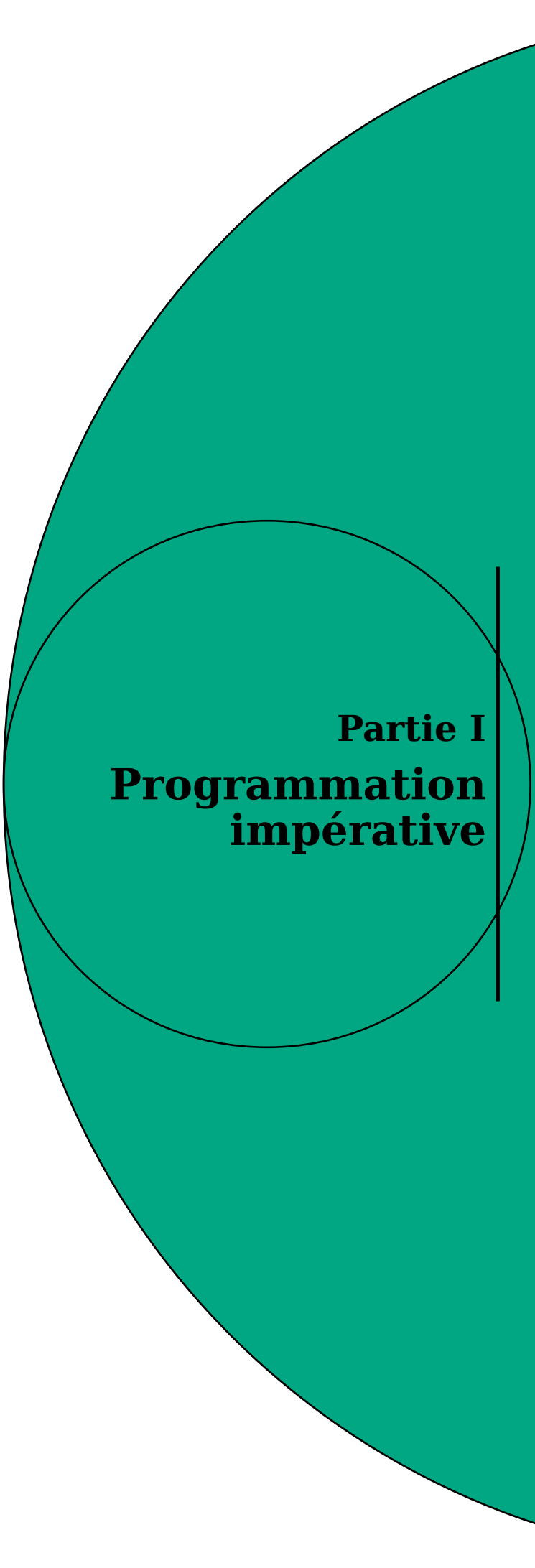
Programmation en C++
Agrégation génie électrique

François Schwarzentruber



Partie I

Programmation impérative

Chapitre 1
Bases

Le C++ n'est pas tout à fait un surensemble du C mais on traite dans ce chapitre les notions qui ne sont pas de la programmation objet.

1.1 Algorithmique de base

1.1.1 Déclarer une variable

```
int x;
```

1.1.2 Structure conditionnelle

```
if(x == 2)
{
    x = 3;
```

```
}
else
{
    x = x + 1;
    y = 0;
}
```

```
switch(x)
        {
        case 0:
            {
                cout << "miaou";
                break;
            }
       case 1:
            {
                cout << "waf";
                break;
            }
      default:
            {
                cout << "rien";
            }
    }
```

1.1.3 Boucle while

```
while(x == 2)
{
    x = x + 1;
    y = y - 1;
}
```

1.1.4 Boucle for

```
for(i = 0; i < 100; i++)
{
    x = x + i;
    y = y - 1;
}
```

1.1.5 Procédures et fonctions

```
void f(int x)
{
    y = 2*x;
}
```

```
int f(int x)
{
    return 2*x;
}
```

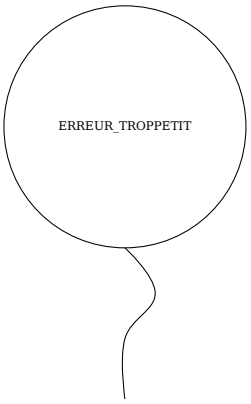
1.2 Mon premier programme

```
#include <iostream>

using namespace std;

int main()
{
    int x = 5;
    x = x + 1;
    cout << "Hello world" << x << endl;
    return 0;
}
```

1.3 Manipuler des fichiers

```
#include <iostream>
#include <fstream>

using namespace std;

int main()
{
    ifstream fichier("cjk-decomp-0.4.0.txt", ios::in);
    string contenu;
    getline(fichier, contenu);
    cout << contenu << endl;
    return 0;
}
```

1.4 Exceptions



On peut traiter des événements "exceptionnels" en lançant une exception : par exemple 'ERREUR_TROPPETIT'. L'erreur est "intercepté" par le premier 'catch' ou alors le programme plante :

```
2 [sig] consoleapp 5000 open_stackdumpfile:
Dumping stack trace to consoleapp.exe.stackdump
```

```
#define TOOSMALL 0
#define TOOSBIG 1

double getElement(vector<double> & v, int i) {
    if (i < 0)
```

```
    {
        throw(ERREUR_TROPPETIT);
    }
    else if (i >= v.size())
    {
        throw(ERREUR_TROPGRAND);
    }
    return v[i];
}


bool isElementPositif(vector<double> & v, int i) {
    try
    {
        return (getElement(v, i) > 0);
    }
    catch (int error)
    {
        return false;
    }
}
```

1.5 Classe

```
class Point
{
    public:
        int x;
        int y;

};
```

L'utilisation se fait ainsi :

```
    Point p;
    p.x = 4;
    p.y = 3;
```

1.6 Spécification d'une fonction

Mauvaise pratique : fonction non spécifiée donc vous ne savez pas ce que fait la fonction.

```
double getElementSafe(vector<double> & v, int i) {
   return v[i];
}
```

Bonne pratique : spécifier les entrées, les sorties et les effets de bord (les variables modifiées par la fonction) à l'aide de commentaires et d'assertions.

Bonne pratique :

```
#include <cassert>

 //! function that returns the i^th element of the vector v
```

```
/*! it supposes that i is between 0 and v.size() - 1
/*!
\param v a vector
\param i an integer
\return The i^th element of the vector v
*/
double getElement(vector<double> & v, int i) {
    assert(i >= 0);
    assert(i < v.size());

    return v[i];
}
```

assert vérifie la condition et le programme plante si l'assertion n'est pas vérifié.

Si les paramètres d'une fonction ne vérifient pas la spécification, le responsable est l'appelant et le bug est dans l'appel.

Si la fonction ne retourne pas le bon résultat, alors le responsable est celui qui a écrit la fonction. Le bug est dans la fonction.

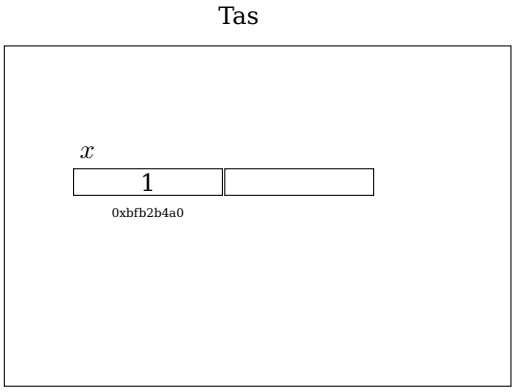
Chapitre 2
Pointeurs

2.1 Adresse mémoire

```
int x = 1;
```

x est l'entier 1.

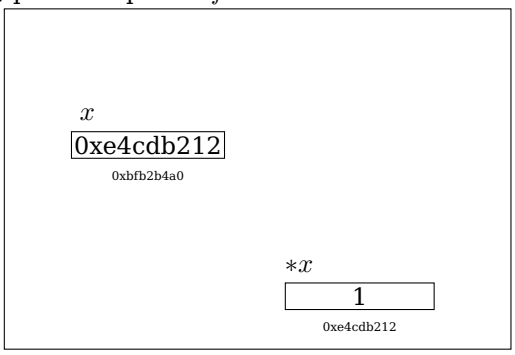
$\&x$ est l'adresse de x, par exemple $0xbfb2b4a0$. C'est là que la valeur 1 est inscrite.



2.2 Déclarer un pointeur

```
int *x;
```

x est une variable qui contient une adresse (par exemple 0xe4cdb212) vers un entier $*x$. $\&x$ est l'adresse de x, par exemple $0xbfb2b4a0$.



Example 1

```
int a = 3;
int *x = &a;
```

a est une variable contenant 3. Son adresse est $\&a$. x est un pointeur vers un entier. On fait pointer x vers la case qui contient a.

A la fin, x et $\&a$ dénotent l'adresse de la case qui contient a (3).

$*x$ représente la valeur inscrite dans la case où pointe x.

Après exécution du programme suivant :

```
int a = 3;
int *x = &a;
*x = 4;
```

$*x$ et a dénotent la valeur dans la case de a c'est à dire 4.

Exercice 1 *Que vaut a à la fin du programme suivant ?*

```
int a = 3;
int *x = &a;
*x = a+1;
a = *x + 1;
```

Remark 1 *x (c'est à dire une adresse vers une case qui contient un entier), $*x$ (cet entier), a sont modifiables. $\&a$ (l'adresse de a), $\&x$ ne sont pas modifiables.*

2.3 Allocation

Ce code crée une case mémoire pour un entier, y place 5 et x pointe vers cette case.

```
int *x = new int(5);
```

Pour désallouer :

```
delete x;
x = 0;
```

On affecte 0 à x pour dire que x pointe sur rien (il pointe sur la case d'adresse 0 qui ne correspond à rien).

2.4 Objet

```
Point *p = new Point();
(*p).x = 4;
(*p).y = 3;
```

Mais en fait, on écrit plutôt :

```
Point *p = new Point();
p->x = 4;
p->y = 3;
```

D'ailleurs on peut écrit des constructeurs d'objets et des méthodes :

```
class Point
{
    public:
        int x;
        int y;

        int distanceOrigine()
        {
            return sqrt(this->x * this->x + this->y * this->y);

        }

        Point(int argx, int argy)
        {
            this->x = argx;
            this->y = argy;
        }
```

```
        ~Point()
        {
                cout << "Point␣detruit";
        }

};
```

this est un pointeur vers l'objet courant.
L'utilisation se fait ainsi et appelle constructeur Point(int argx, int argy) :

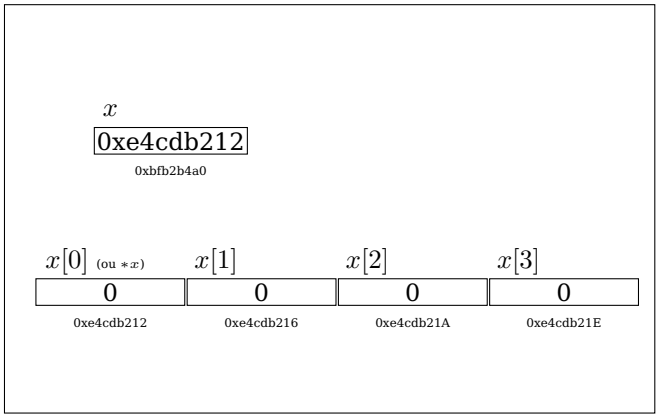
```
Point *p = new Point(3, 4);
cout << p->distanceOrigine() << endl;
```

À la destruction de l'objet, le destructeur ~Point() est automatiquement appelé.

2.5 Tableau

Pour allouer un tableau, par exemple de 4 cases :

```
int *x = new int[4];
```



Les valeurs des 4 cases sont $x[0], \ldots, x[3]$.
$x[4]$ est illicite.
$*x$ représente en fait la valeur dans la première case du tableau, c'est à dire $x[0]$.
Pour désallouer :

```
delete[] x;
x = 0;
```

Pour allouer un tableau de tableau, on peut faire :

```
#define HEIGHT 5
#define WIDTH 3

double **p2DArray;

  p2DArray = new double*[HEIGHT];
  for (int i = 0; i < HEIGHT; ++i)
       p2DArray[i] = new double[WIDTH];
```

2.6 Appels de fonctions

2.6.1 Par valeur

Si on définit :

```
void f(int a)
{
    a = 4;
}
```

après

```
int x = 3;
f(x);
```

x vaut toujours 3.

2.6.2 Par référence

Si on définit :

```
void f(int &a)
{
    a = 4;
}
```

après

```
int x = 3;
f(x);
```

x vaut toujours 4.

Exercice 2 *Avec :*

```
void f(int * a)
{
    ...
}
```

quels appels sont corrects ?

`int *x = 5;` `f(x);`	
`int x = 5;` `f(&x);`	
`int x = 5;` `f(x);`	

2.6.3 Protéger ses paramètres

```
void f(const int a)
{
    a = 4;
}
```

Le mot *const* empêche la modification du paramètre : c'est une garantie. Le code précédent ne compile pas.

De même :


```
void f(const int &a)
{
    a = 4;
}
```

ne compile pas.

Conseil : utilisez le mot *const* au maximum.

2.7 Référence

Le code suivant fait que a et b sont les même cases mémoires.

```
int a;
int& b = a;
```

Exercice 3

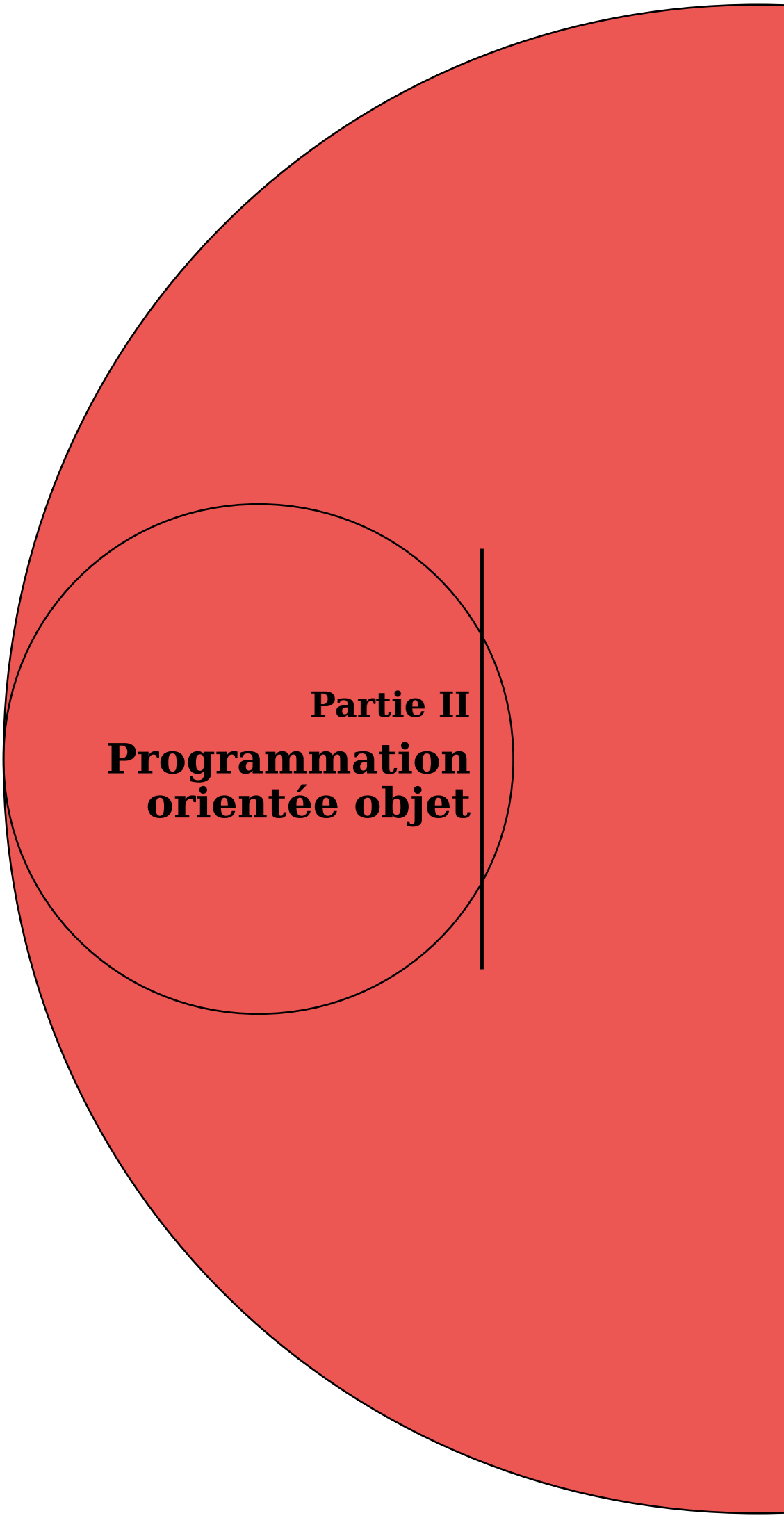
```
int a = 3;
int& b = a;
a++;
```

Que vaut b ?

Exercice 4 *A-t-on le droit de faire ?*

`int& g(int a)`
`{`
`  int b = 2*a;`
`  return b;`
`}`
	oui - non

`int& g(int a)`
`{`
`  int* b = new int(2*a);`
`  return *b;`
`}`
	oui - non



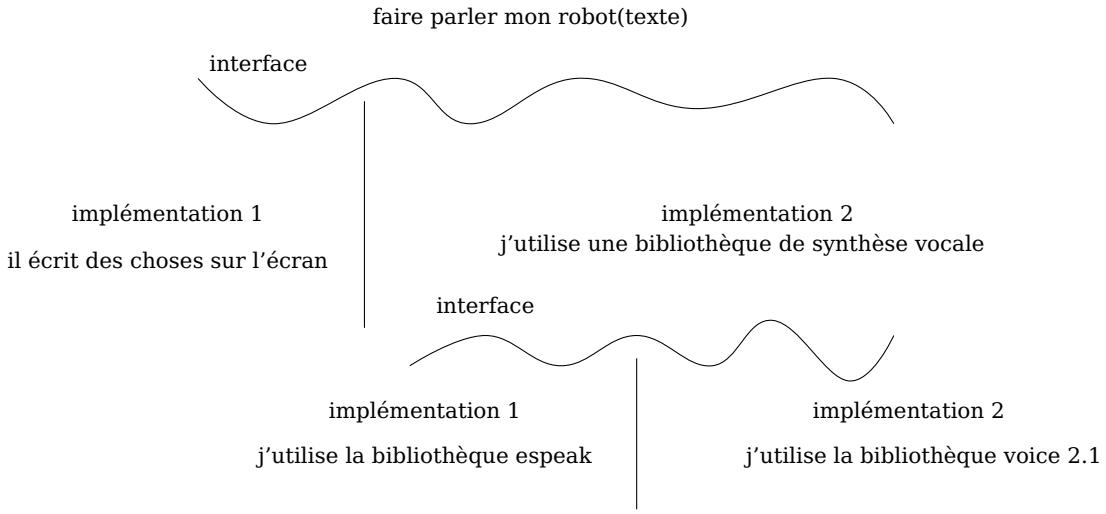
Partie II

Programmation orientée objet

Chapitre 1
Concepts

1.1 Interface/implémentation

Abstraire c'est pouvoir revenir sur des choix sans remettre en cause le programme. C'est aussi être indépendant : des morceaux de programme sont communs.



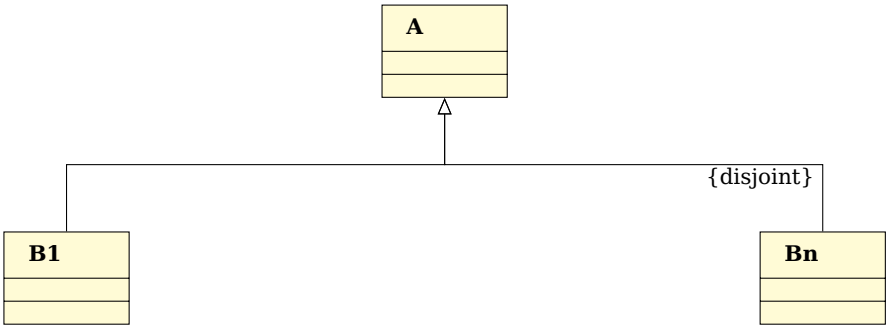
1.2 Héritage

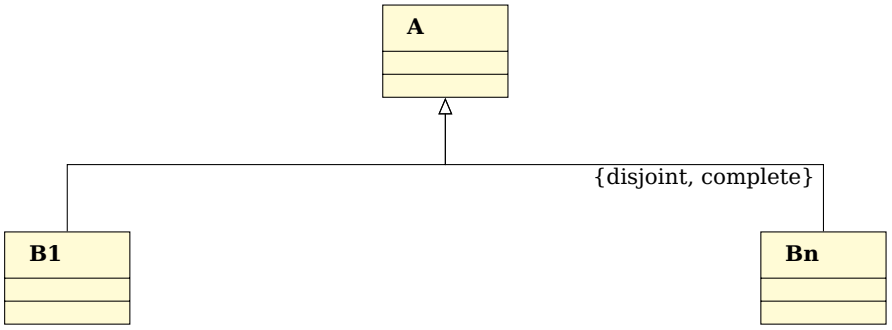
Tous les mammifères sont des animaux. Ce syllogisme peut se représenter à l'aide d'un diagramme de classes UML :

En C++, on écrit :

```
class Animal {
    ...
}


class Mammifere: public Animal {
    ...
};
```

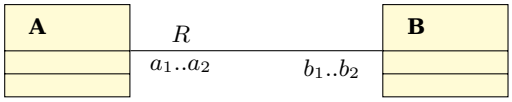
La notation UML permet de représenter que des classes sont disjointes (mais C++ ne permet pas de le faire) :



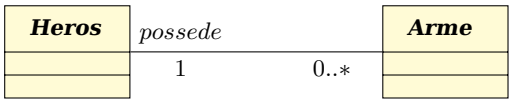


1.3 Associations

Une association est une relation abstraite entre deux classes.

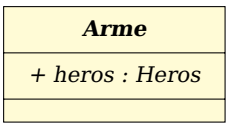


Example 2



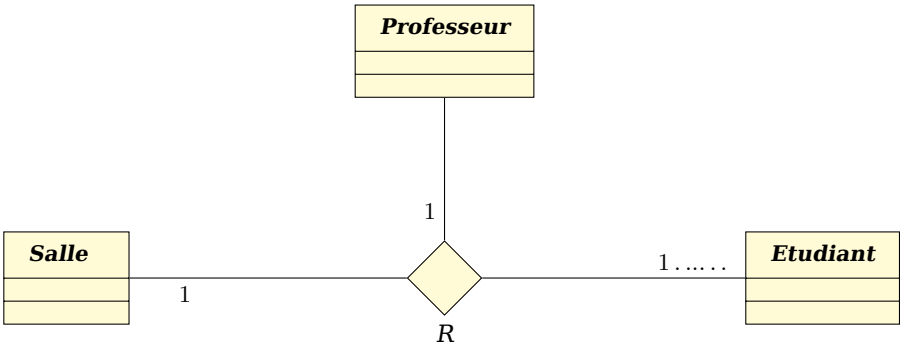

```
class Heros {
    std::set<Arme> armes;
}


class Arme {
    Heros heros;
};
```

On peut aussi représenter un champ de la manière suivante :



Associations n-aires

Example 3

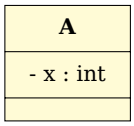


Concepts

1.4 Visibilité

La visibilité s'applique aux champs et aux méthodes.

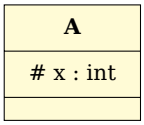
1.4.1 Privé




```
class A {                          OK
    private:                       OK
->      int x;                     OK
        A* a;                      OK
                                   OK
    int f();                       OK
};                                 OK

class B: public A {
    private:
    int g();
};

class C {
    private:
        int f();
};


int A::f() {                       OK
    return x + a->x;               OK
}

int B::g() {
    ...
}
```

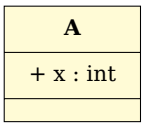
```
int C::f() {
    ...
}
```

1.4.2 Protégé




```
class A {                          OK
    protected:                     OK
–>      int x;                     OK
        A* a;                      OK
                                   OK
    int f();                       OK
};                                 OK

class B: public A {                OK
    private:                       OK
    int g();                       OK
};                                 OK

class C {
    private:
        int f();
};


int A::f() {                       OK
    return x + a->x;               OK
}

int B::g() {                       OK
    ...                            OK
}                                  OK

int C::f() {
    ...
}
```


1.4.3 Public



```
class A {                          OK
    public:                        OK
->      int x;                     OK
        A* a;                      OK
                                   OK
    int f();                       OK
};                                 OK

class B: public A {                OK
    private:                       OK
    int g();                       OK
};                                 OK

class C {                          OK
    private:                       OK
        int f();                   OK
};                                 OK


int A::f() {                       OK
    return x + a->x;               OK
}

int B::g() {                       OK
    ...                            OK
}                                  OK

int C::f() {                       OK
    ...                            OK
}                                  OK
```

Exercice 5

```
class A {
        private:
                int x;

        protected
                int y;

        public
                int z;
}

class B: public A {
        ...
}
```

- ☐ *Je peux utiliser x dans toutes les méthodes de B.*
- ☐ *Je peux utiliser y dans toutes les méthodes de B.*
- ☐ *Je peux accéder aux champs x d'objet n'importe où dans le programme.*
- ☐ *Je peux accéder aux champs y d'objet n'importe où dans le programme.*
- ☐ *Je peux accéder aux champs z d'objet n'importe où dans le programme.*

1.5 Le mot clé $Friend$

Le mot clé $Friend$ permet un contrôle plus fin sur les aspects public et privé. Dans l'exemple suivant, a est un membre privé mais la fonction g définie en dehors de la classe a le droit d'utiliser des membres privés d'objets de type X.

```
class X {
      int a;
      friend void g( int );
   };



void g(int a)
{
        ...
}
```


Remark 2 *La syntaxe n'est pas terrible. On ne rajoute pas de méthode g à X !*

Dans l'exemple suivant, a et $f()$; sont privés sauf pour la classe B, qui, dès qu'elle manipule un objet de type A peut utiliser à son gré les membres privés.

```
class B;

 class A {
        private:
               int a;
               f();
        friend B;
 };

  class B {
     void h (A*p)  {  p->a = 0; p->f(); }
 };
```

On peut être encore plus précis ! Par exemple, on peut n'autoriser qu'une méthode de la classe B à accéder aux champs privés de A :

```
class B;
class A{
    private: int a;
    friend void B :: f();
};
```

1.6 Étendre une classe

1.6.1 Virtual

Le mot clé $Virtual$ fait en sorte que l'héritage se passe bien.

```
#include <iostream>
```

```
using namespace std;
class Chat
{
public:
    void griffer() { cout << "griffe!\n"; }
    virtual void miauler() { cout << "miaou!\n"; }
};

class ChatMagique : public Chat
{
public:
    void griffer() { cout << "griffemagique!\n"; }
    void miauler() { cout << "MIAOUMIAOU!\n"; }
};

int main()
{
    Chat a;
    a.griffer(); // affiche "griffe!"
    a.miauler(); // affiche "miaou!"

    cout << endl;

    ChatMagique b;
    b.griffer(); // affiche "griffemagique!"
    b.miauler(); // affiche "MIAOUMIAOU!"

    cout << endl;
    // copie non polymorphe
    a = b;
    a.griffer(); // affiche "griffe!"
    a.miauler(); // affiche "miaou!"

    cout << endl;
    // utilisation polymorphe de B (par pointeur)
    Chat * pa = &b;
    pa->griffer(); // affiche "griffe!"
    pa->miauler(); // affiche "MIAOUMIAOU!" <-- grace a virtual

    cout << endl;
    // utilisation polymorphe de B (par reference)
    Chat & ra = b;
    ra.griffer(); // affiche "griffe!"
    ra.miauler(); // affiche "MIAOUMIAOU!" <-- grace a virtual
}
```

Concepts

1.6.2 Classe abstraite

Là la méthode *miauler* n'est pas défini.

```
class ChatAbstrait
{
public:
```

```
    virtual void miauler() = 0;  // = 0 signifie "virtuelle pure"
};
```

Elle sera définie dans les classes qui hérite de ChatAbstrait. ChatAbstrait s'appelle une classe abstraite.

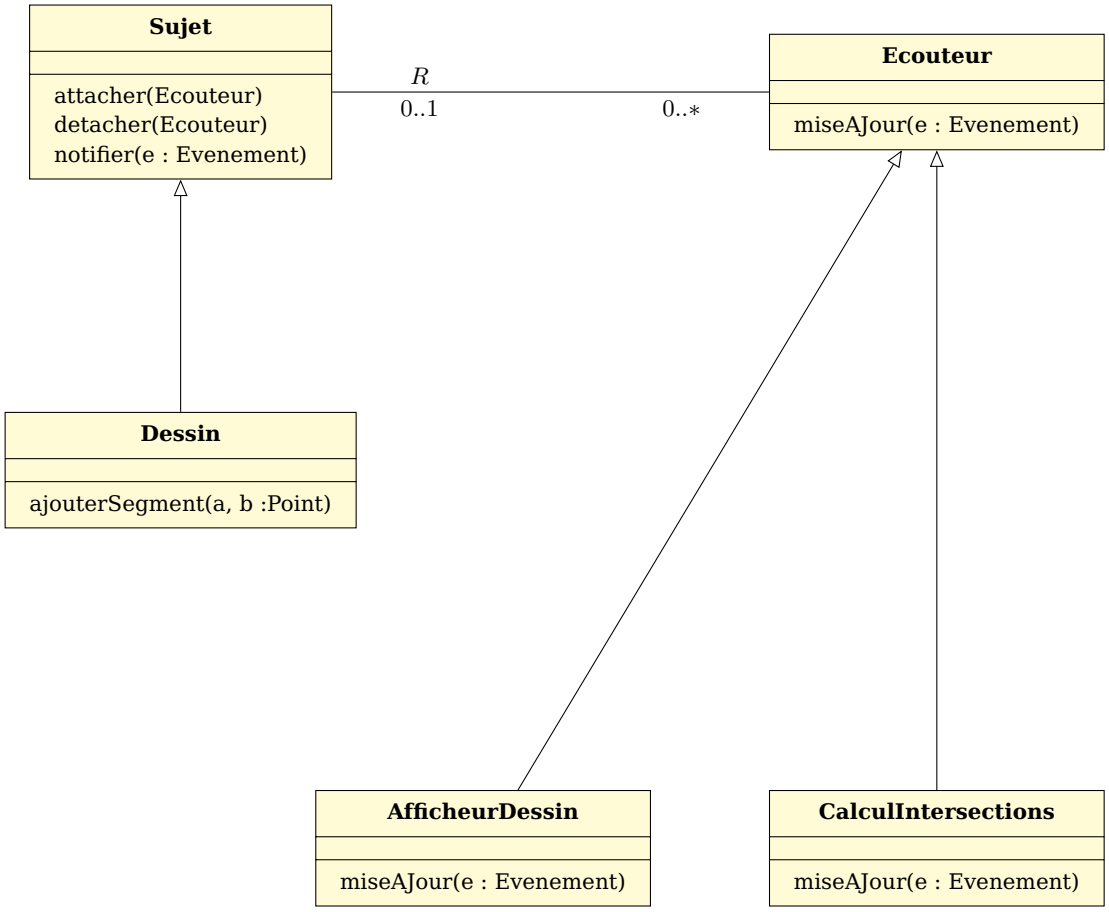
Chapitre 2
Patrons de conception

2.1 Travailler à plusieurs

Sur certains projets, 2000 personnes contribuent. Ajouter une fonctionnalité ne doit pas bouleverser toute la structure du programme. Les patrons de conception sont des schémas généraux de bonne architecture de programme.

2.2 Écouteur

Ce patron perd de garantir que l'affichage et autres calculs sur les données sont toujours à jour. De plus, il facilite l'ajout de fonctionalité de traitement des données.



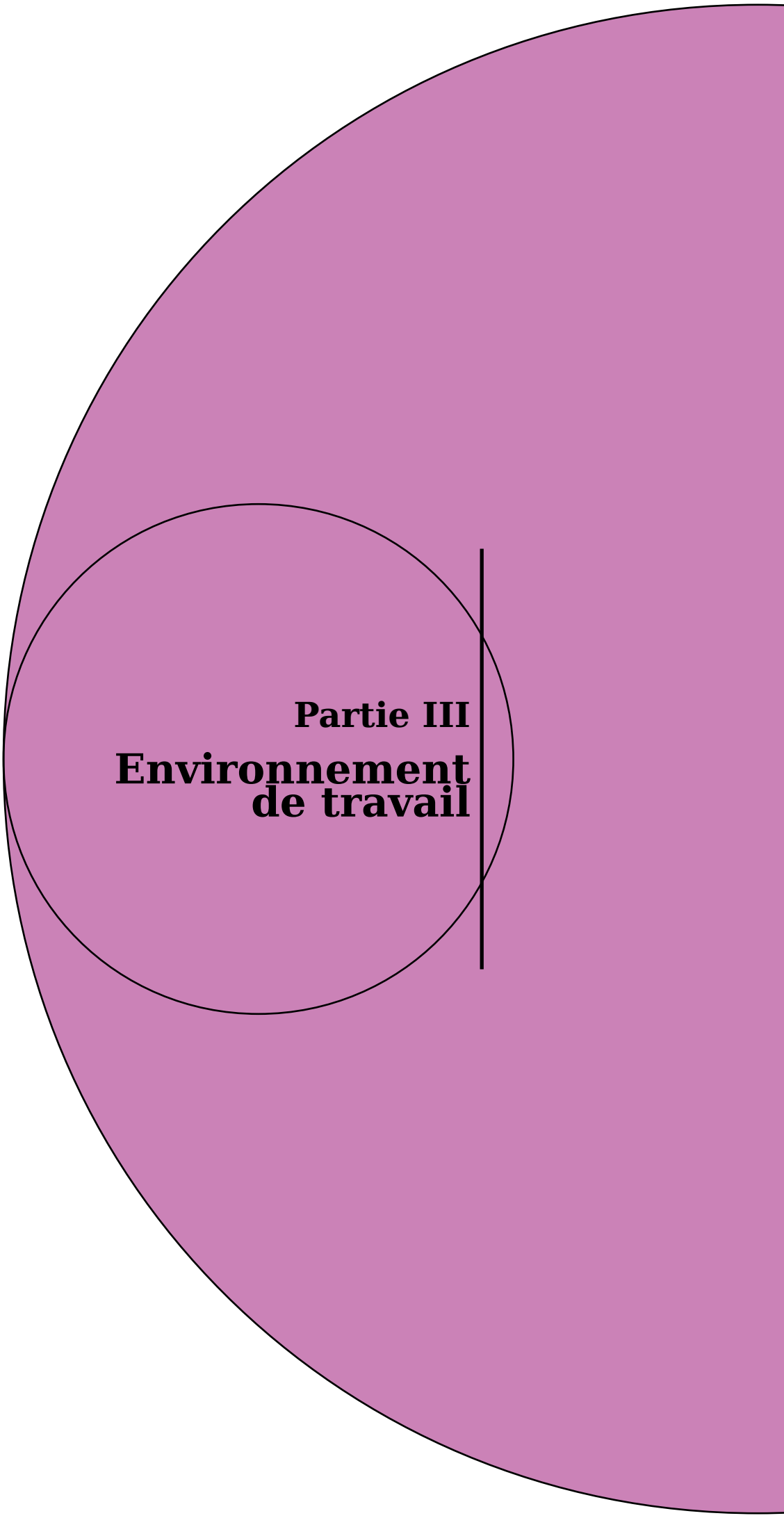

```
void Dessin::ajouterSegment(Point a, Point b)
{
    ...
    notifier(new Evenement(a, b));

}



Sujet::attacher(Ecouteur ecouteur)
{
   ...

}
```

2.3 Fabrique abstraite

Le patron Fabrique abstraite permet de s'abstraire des créations d'objets. Cf. google.

2.4 Commande

Le patron Commande permet de se souvenir des commandes effectuées afin de pouvoir les annuler (typiquement un Ctrl + Z dans un logiciel de texte), les rejouer ou sauvegarder une séquence d'actions. Cf. google.



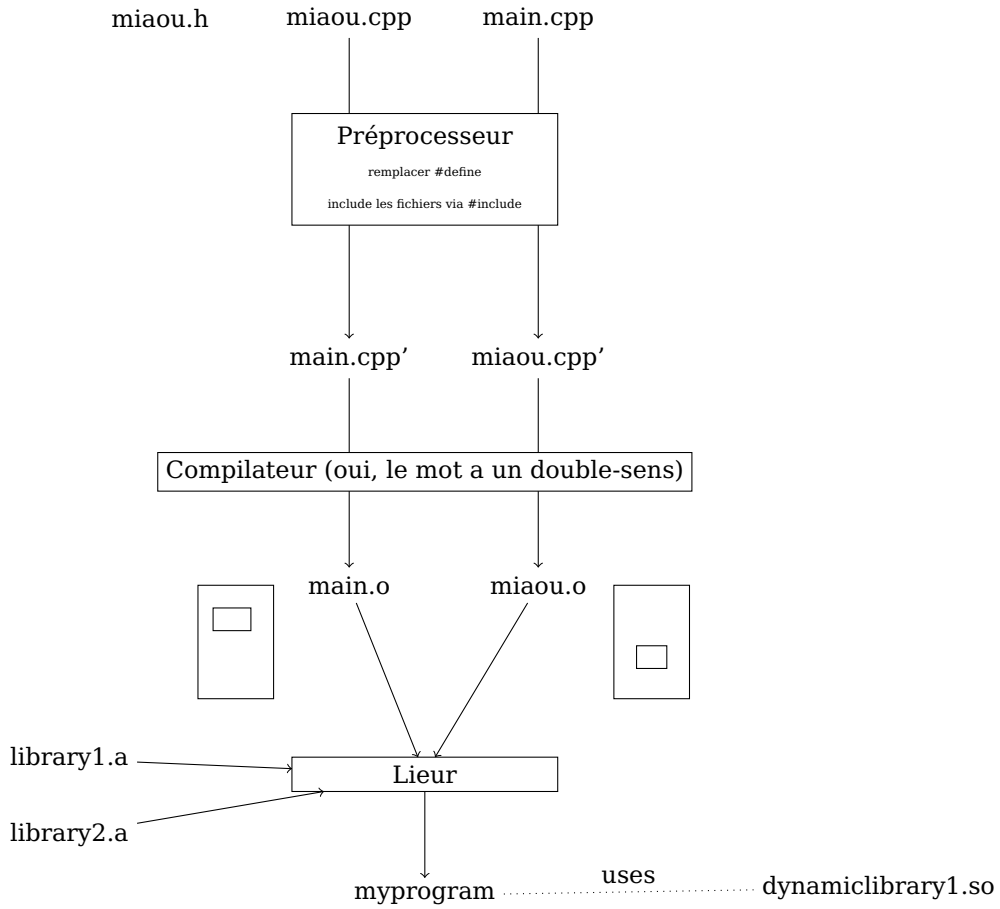
Partie III

Environnement de travail

Chapitre 1
Environnement

1.1 Le compilateur

Le compilateur g++ transforme vos fichiers sources (.cpp) en un programme exécutable.



1.2 Installer une bibliothèque

Généralement, il faut aller dans le répertoire de la bibliothèque et faire :

```
./configure
make
make install
```

Mais il arrive que la procédure est différente. Se renseigner sur le site web de la bibliothèque.

1.3 Code : :Blocks

Code : :Blocks est un environnement de développement téléchargeable ici :

`http://www.codeblocks.org/`

Il permet d'éditer les fichiers et d'appeler le compilateur.

1.3.1 Configurer le compilateur

Il se peut que Code : :Blocks ne reconnaisse pas bien votre compilateur.
Settings > Compiler > Selected compilers (créer un nouveau compilateur dans la liste)
Aller dans 'Toolchain executables' et indiquez où se trouve le compilateur.

1.3.2 Faire reconnaître les .h de votre projet

Vous devez configurer votre projet et indiquer où se trouve les fichiers .h. Les étapes sont :

- Project > Build options > Search Directories (tab)
- Choisir la Policy 'Prepend target options to project options'
- Cliquer sur 'Add' pour chercher et ajouter un répertoire (par exemple 'include')
- À la question "Keep this as a relative path ?", répondez 'Yes' pour ajouter le répertoire.

1.3.3 Utiliser une bibliothèque

Il se peut que vous ayez l'erreur suivante :
`cannot find -lSDLmain`
Bouton droit sur le projet > Building options > Linker settings
et modifier les librairies.

1.4 Documentation

C++ est livré avec multiples bibliothèques notamment pour manipuler les structures de données (ensembles, des piles, etc.). La documentation est disponible ici :

`http://www.cplusplus.com/reference/stl/`

Nous utiliserons également la bibliothèque Allegro (c'est une bibliothèque C) pour dessiner et gérer la souris :

`http://alleg.sourceforge.net/a5docs/5.0.10/`

Vous pouvez produire de la documentation de votre programme à l'aide de l'outil :

`http://sourceforge.net/projects/doxygen/?source=dlp`