Cours n° 3
C++ : allocation mémoire et constructeurs
Module Info3/M3105C

Sebastien.Kramm@univ-rouen.fr

IUT GEII Rouen

2018-2019

Sommaire

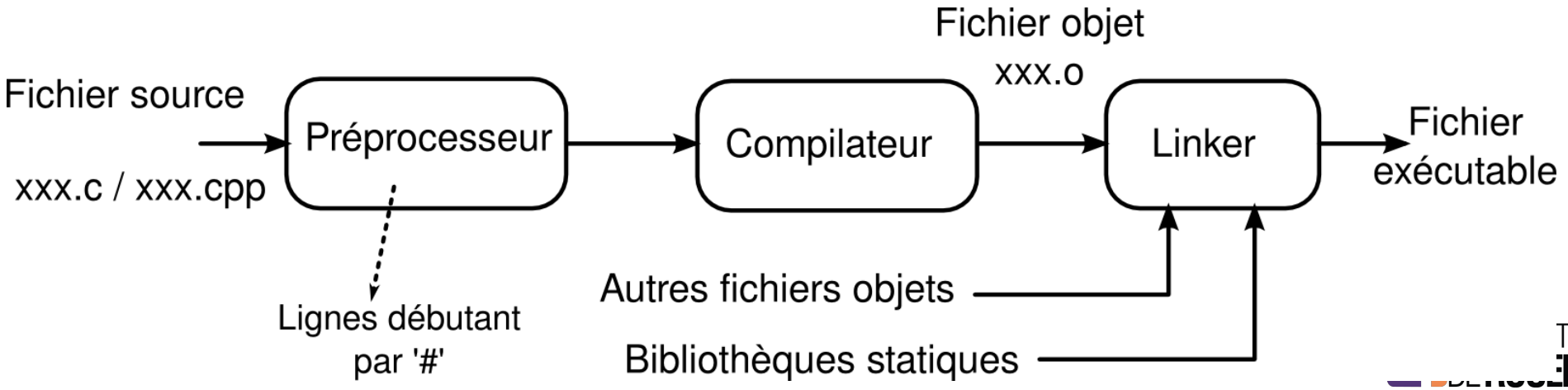
UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Production d'un fichier exécutable en C/C++

- 3 outils logiciels :
  - préprocesseur,
  - compilateur,
  - éditeur de liens (*linker*).
- L'enchainement de ces trois outils est par défaut **automatique** (avec une IDE ou en ligne de commande).
  `$> g++ program.cpp -o program.exe`
- Par extension, l'expression "compiler" désigne l'exécution de ces 3 outils successivement.



Sommaire

UNIVERSITÉ DE ROUEN

Portée d'un identificateur

- Identificateur = nom de variable ou de fonction.
- Portée = zones du source où il est reconnu.
  - **locale** $\Rightarrow$ connu dans le bloc où l'identificateur est déclaré,
  - **globale** $\Rightarrow$ déclaration en dehors d'un bloc, et reconnu dans tout le fichier.

Portée d'un identificateur

- Identificateur = nom de variable ou de fonction.
- Portée = zones du source où il est reconnu.
  - **locale** ⇒ connu dans le bloc où l'identificateur est déclaré,
  - **globale** ⇒ déclaration en dehors d'un bloc, et reconnu dans tout le fichier.

```
int a = 4;
void fonc_2();
//-----------------------------
main()
{
   void fonc_1();
   int b;
}
//-----------------------------
void fonc_1()
{
   int a = 3;
   int b;
   cout << "a = " << a << endl;
}
```

Sommaire

UNIVERSITÉ DE ROUEN

Rappel : Déclaration $\neq$ Définition

- Objets (variables) et fonctions doivent être déclarés,
- Le sens du mot "déclaration" n'est pas le même !
- "Définition" $\Rightarrow$ n'a de sens que pour une fonction.

1 - Déclaration d'une variable

- Réservation mémoire pour son stockage,
- Information au compilateur : *le symbole 'x' dont on parle par la suite est un objet du type 'bidule'*,
- Doit être unique !

1 - Déclaration d'une variable

- Réservation mémoire pour son stockage,
- Information au compilateur : *le symbole 'x' dont on parle par la suite est un objet du type 'bidule'*,
- Doit être unique !

```
int    a;
POINT b;
BIDULE x;
TRUC* pt;
```

(en C++ : "objet" = "variable")
Rem : SAUF pour les pointeurs : pointeur $\neq$ objet

2 - Déclaration d'une fonction

- Pas de réservation mémoire.
- Information au compilateur : *le symbole 'x' dont on parle par la suite est une fonction qui a "tels" arguments, et renvoie une valeur de "tel" type* (**signature** de la fonction).
- Peut être multiple (mais les déclarations doivent être identiques !)
- On peut compiler sans avoir le code de la fonction disponible.
- Mais le *Build* final (édition de lien) en aura besoin :
  - soit sous forme de code source compilé dans le même projet,
  - soit dans une librairie pré-compilée (*static library*).

2 - Déclaration d'une fonction

- Pas de réservation mémoire.
- Information au compilateur : *le symbole 'x' dont on parle par la suite est une fonction qui a "tels" arguments, et renvoie une valeur de "tel" type* (**signature** de la fonction).
- Peut être multiple (mais les déclarations doivent être identiques !)
- On peut compiler sans avoir le code de la fonction disponible.
- Mais le *Build* final (édition de lien) en aura besoin :
  - soit sous forme de code source compilé dans le même projet,
  - soit dans une librairie pré-compilée (*static library*).

```
float sqrt( float val );
TRUC fonc( MACHIN m );
TRUC* fonc2( MACHIN& m );
```

Exercice

Les déclarations suivantes font elles une allocation mémoire ?

`A b;`	Oui - Non
`C d();`	Oui - Non
`E* f();`	Oui - Non
`G* h;`	Oui - Non
`I j( K l );`	Oui - Non
`M n( O* p, Q* r );`	Oui - Non

Sommaire

UNIVERSITÉ DE ROUEN

Allocation mémoire

- A chaque objet correspond un *espace mémoire*, qui est attribué par l'OS au moment de l'exécution.
- La façon dont l'objet est crée conditionne sa *durée de vie*.
- Un objet peut être crée en mémoire de 2 façons :
  - par une *déclaration* : allocation mémoire **automatique** ou **statique**,
  - par l'exécution d'une *fonction* de gestion dynamique de la mémoire : allocation **dynamique**.

UNIVERSITÉ DE ROUEN

Allocation mémoire

- A chaque objet correspond un *espace mémoire*, qui est attribué par l'OS au moment de l'exécution.
- La façon dont l'objet est crée conditionne sa *durée de vie*.
- Un objet peut être crée en mémoire de 2 façons :
  - par une *déclaration* : allocation mémoire **automatique** ou **statique**,
  - par l'exécution d'une *fonction* de gestion dynamique de la mémoire : allocation **dynamique**.
- Allocation :
  - **automatique** : l'objet est crée "automatiquement" lors de l'exécution de la déclaration, et détruit "automatiquement" à la fin du bloc,
  - **statique** : l'objet n'est pas détruit à la fin du bloc dans lequel il est déclaré, sa valeur est conservée (variable globale ou `static`),
  - **dynamique** : c'est le programme qui décide du moment de la création et de la destruction.

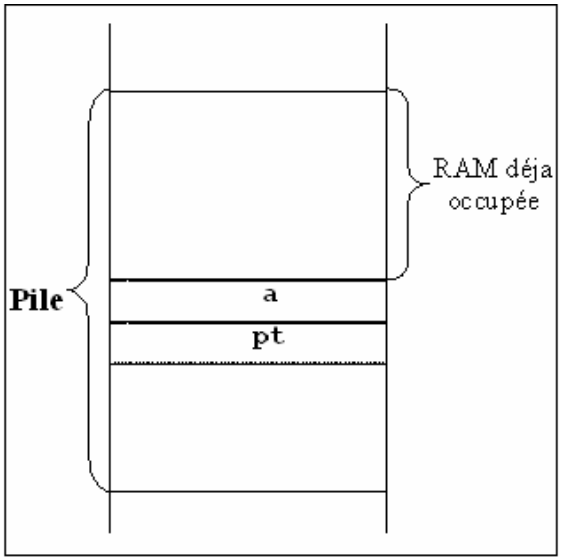
UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

1 - Allocation automatique

```
void ma_fonction()
{
   int a;     // déclaration d'un entier
   POINT pt; // déclaration d'un point
    ...
}
```

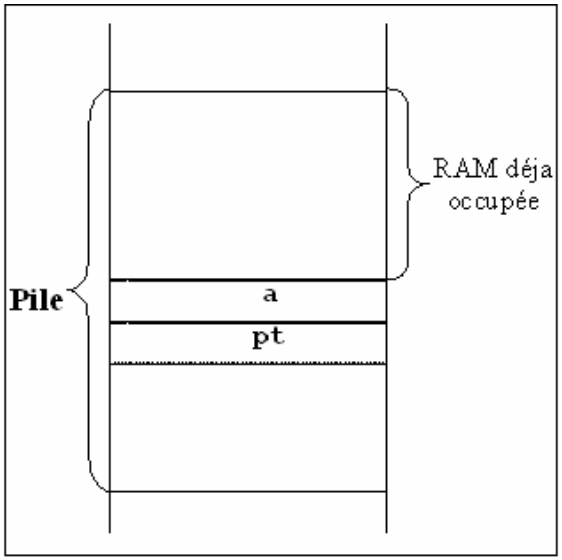


- L'OS réserve un bloc mémoire de la taille de l'objet dans la **pile**.
- C : pas d'initialisation par défaut.
- C++ : on pourra initialiser l'objet, avec des valeurs, ou un autre objet :

```
POINT pt1( 3, 4 );
POINT pt2( pt1 );
```

1 - Allocation automatique

```
void ma_fonction()
{
   int a;     // déclaration d'un entier
   POINT pt; // déclaration d'un point
    ...
}
```



- L'OS réserve un bloc mémoire de la taille de l'objet dans la **pile**.
- C : pas d'initialisation par défaut.
- C++ : on pourra initialiser l'objet, avec des valeurs, ou un autre objet :

```
POINT pt1( 3, 4 );
POINT pt2( pt1 );
```

- Adresse de stockage accessible avec l'opérateur "adresse de" (&) :

```
cout << " adresse de a = " << &a << endl;
```

Sommaire

UNIVERSITÉ DE ROUEN

2 - Allocation statique

- Deux types de variables statiques :
  - Objet déclaré en dehors de toute fonction : portée **globale**.
  - Objet déclaré dans un bloc (fonction) assorti de *static* : portée **locale**.

  $\Rightarrow$ Dans une fonction, la valeur est conservée d'un appel à l'autre.

2 - Allocation statique

- Deux types de variables statiques :
  - Objet déclaré en dehors de toute fonction : portée **globale**.
  - Objet déclaré dans un bloc (fonction) assorti de *static* : portée **locale**.

  $\Rightarrow$ Dans une fonction, la valeur est conservée d'un appel à l'autre.

```
AUTOBUS abus; // var. statique  globale

void ma_fonction()
{
   static SOUSMARIN sm; // var. statique  locale
   static int iter;     // var.  statique  locale
   cout << "appel " << iter++ << endl;
}
```

- Les variables statiques sont détruites **après** la fin de `main()`.

Allocation statique

- Le programme suivant :

```
int main()
{
   cout << "Start\n;"
   ma_fonction();
   ma_fonction();
}
```

- Donnera :

```
Start
appel 0
appel 1
```

Allocation statique

- Le programme suivant :

```
int main()
{
   cout << "Start\n;"
   ma_fonction();
   ma_fonction();
}
```

- Donnera :

```
Start
appel 0
appel 1
```

- Création des objets statiques : on distingue deux situations, en fonction de la portée :
  - globale : la variable est allouée **avant** l'exécution de `main()`.
  - locale : la variable est allouée **au moment** où le contrôle de l'éxécution trouve sa déclaration.

Allocation statique : exemple

- Soit le code suivant :

```
class TRUC
{
   TRUC()
   { cout << "constructeur\n"; }
};

void ma_fonc()
{
   static TRUC t;
}
```

- Avec comme "main()" :

```
int main()
{
   cout << "Start\n;"
   ma_fonc();
}
```

Allocation statique : exemple

- Soit le code suivant :

```
class TRUC
{
   TRUC()
   { cout << "constructeur\n"; }
};

void ma_fonc()
{
   static TRUC t;
}
```

- Avec comme "main()" :

```
int main()
{
   cout << "Start\n;"
   ma_fonc();
}
```

- A l'exécution, on aura :

```
Start
constructeur
```

⇒ La variable n'est crée qu'à la première exécution de la fonction.

Sommaire

UNIVERSITÉ DE ROUEN

3 - Allocation dynamique

- C'est le programme qui effectue une demande à l'OS une allocation au moment de l'exécution ("*Runtime*").
- La mémoire est allouée sur le **tas** (*heap*).
- Implique l'utilisation de **pointeurs**.
- Des pièges : risque de "plantage" en cas :
  - d'utilisation de pointeur non initialisé,
  - d'utilisation d'un pointeur sur une zone mémoire déjà libérée,
  - etc...

UNIVERSITÉ DE ROUEN

Pointeur sur un objet

- L'allocation dynamique implique l'utilisation d'un pointeur.
- Pour éviter qu'il ne pointe sur une zone indéterminée de la mémoire, on l'initialise à 0.
- La déclaration du pointeur alloue de la mémoire **pour le pointeur seulement !**
  (pas pour l'objet qu'on va manipuler via ce pointeur).

```
void ma_fonction( void )
{
// déclaration d'un pointeur sur un point
   POINT* pt = 0;
    ...
}
```

Pointeurs : accès aux méthodes de l'objet

- On accède aux méthodes de l'objet avec l'opérateur $->$

```
VOITURE* voiture;
...    // ici allocation mémoire
voiture->Demarre();
voiture->MetUneVitesse( 1 );
```

- La variable-pointeur contient l'**adresse** de l'objet pointé :

```
cout << "adresse = " << voiture << endl;
```

Allocation dynamique en C++

- On dispose de l'operateur `new` :

```
POINT* pt = 0;
pt = new POINT;
```

Allocation dynamique en C++

- On dispose de l'operateur `new` :

```
POINT* pt = 0;
pt = new POINT;
```

- Ou en une seule ligne :

```
POINT* pt = new POINT;
```

Allocation dynamique en C++

- On dispose de l'operateur `new` :

```
POINT* pt = 0;
pt = new POINT;
```

- Ou en une seule ligne :

```
POINT* pt = new POINT;
```

- En cas d'impossibilité (plus suffisamment de mémoire), le programme s'interrompt (génération d'une **exception**).

Allocation dynamique en C++

- On dispose de l'operateur `new` :

```
POINT* pt = 0;
pt = new POINT;
```

- Ou en une seule ligne :

```
POINT* pt = new POINT;
```

- En cas d'impossibilité (plus suffisamment de mémoire), le programme s'interrompt (génération d'une **exception**).
- La **libération** du bloc mémoire se fera avec l'opérateur delete :

```
delete pt;
```

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Pièges de l'allocation dynamique

- Attention aux risque d'erreur : ne **pas** utiliser l'objet s'il a été détruit !

Pièges de l'allocation dynamique

- Attention aux risque d'erreur : ne **pas** utiliser l'objet s'il a été détruit !
- Exemple :

```
CHIEN* mon_chien = new CHIEN( Epagneul, "Rex" );
mon_chien->AvalerRepas();
delete mon_chien; // je tue le chien
 ...
mon_chien->ReclamerNourriture(); // ???
```

  $\Rightarrow$ comportement indéterminé (plantage dans le meilleur des cas...)
- Pour éviter les erreurs, il est recommandé d'utiliser à la place des pointeurs des objets de type **pointeur intelligent** (*smart pointers*) (non traité dans ce cours).

UNIVERSITÉ DE ROUEN

Exercice

- Allocation automatique d'un objet 't' de type TABLE :

- Allocation dynamique d'un objet 'a' de type ALGO :

Sommaire

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Constructeur

- On peut inclure dans la classe une fonction d'initialisation, avec ou sans arguments, appelée **constructeur**, et exécutée **automatiquement** à la création de la variable.
- Cette fonction va permettre d'**initialiser** les attributs.
- En C : (et aussi valable en C++ !)

```
int   a = 3;
float b = 123.45;
```

Constructeur

- On peut inclure dans la classe une fonction d'initialisation, avec ou sans arguments, appelée **constructeur**, et exécutée **automatiquement** à la création de la variable.
- Cette fonction va permettre d'**initialiser** les attributs.
- En C : (et aussi valable en C++ !)

```
int   a = 3;
float b = 123.45;
```

- En C++ :

```
RDV     rdv("2012/03/04", "08:30");
POINT2D pt(3,4);
string s("abc"); (ou string s="abc";)
```

Constructeur

- On peut inclure dans la classe une fonction d'initialisation, avec ou sans arguments, appelée **constructeur**, et exécutée **automatiquement** à la création de la variable.
- Cette fonction va permettre d'**initialiser** les attributs.
- En C : (et aussi valable en C++ !)

```
int   a = 3;
float b = 123.45;
```

- En C++ :

```
RDV     rdv("2012/03/04", "08:30");
POINT2D pt(3,4);
```

`string s("abc");` (ou `string s="abc";`)

- On pourra aussi utiliser cette syntaxe avec les types natifs :

```
int   a(3);
float b(3.1415);
int   c(a);  (identique à int c = a;)
```

Constructeur sans arguments

- A la différence du C, on pourra avoir une **initialisation par défaut**, en l'absence d'arguments.
- En C :

  `int a;` création sans initialisation ! (valeur indéfinie)

Constructeur sans arguments

- A la différence du C, on pourra avoir une **initialisation par défaut**, en l'absence d'arguments.
- En C :

  `int a;` création sans initialisation ! (valeur indéfinie)
- En C++ :

  `POINT pt;` Création d'un objet de type point et initialisation à des valeurs par défaut

Constructeur **implicite**

En cas d'absence de déclaration d'un constructeur dans une classe, le compilateur en synthétise un, de façon transparente, et qui ne fait rien (aucune initialisation !)

UNIVERSITÉ DE ROUEN

Constructeur : déclaration et définition

Un constructeur (explicite) est une méthode qui :

- porte le nom de la classe,
- ne renvoie rien.

Constructeur : déclaration et définition

Un constructeur (explicite) est une méthode qui :

- porte le nom de la classe,
- ne renvoie rien.

- Déclaration :

```
// point.h

class POINT
{
   private:
      int x, y;
   public:
      POINT( int x0, int y0 );
};
```

Constructeur : déclaration et définition

Un constructeur (explicite) est une méthode qui :

- porte le nom de la classe,
- ne renvoie rien.

- Déclaration :

```
// point.h

class POINT
{
   private:
      int x, y;
   public:
      POINT( int x0, int y0 );
};
```

- Définition :

```
// point.cpp

#include "point.h"

POINT::POINT( int x0, int y0 )
{
  x = x0;
  y = y0;
}
```

Initialisation des attributs par *liste d'initialisation*

- Il existe **deux** façons d'initialiser les attributs de la classe dans un constructeur :
  - par **affectation** (diapo précédente)
  - par **liste d'initialisation** :

```
// point.cpp

#include "point.h"

POINT::POINT( int x0, int y0 ) : x( x0 ), y( y0 )
{
    // plus rien ici ...
}
```

- Cette technique est **indispensable** dans certains cas (initialisation d'attributs de type "référence").

Plusieurs constructeurs

- On pourra avoir dans une classe **plusieurs** constructeurs, avec des **signatures** différentes (nombre et/ou type d'arguments).
- Le compilateur reconnaîtra celui à utiliser selon le contexte.
- Exemple : classe avec 2 constructeurs :

```
// point.h

class POINT
{
  private:
    int x, y;

  public:
    POINT( int x0, int y0 ); // constructeur 1, avec arguments
    POINT();                 // constructeur 2, sans arguments
   ...
};
```

Utilisation du constructeur

Le bon constructeur est appelé automatiquement à la création de l'objet, après l'allocation mémoire.

```
#include "point.h"
int main()
{
   POINT a;          // appel du   ------------------
   POINT b(3,4);     // appel du   ------------------
    ...
}
```

Constructeur implicite

- Attention : s'il existe au moins un constructeur explicite, le compilateur ne génère plus de constructeur implicite !

- Exemple : soit la classe suivante :

```
// point.h
class POINT
{
  private:
    int x, y;
  public:
    POINT( int x0, int y0 );
};
```

Constructeur implicite

- Attention : s'il existe au moins un constructeur explicite, le compilateur ne génère plus de constructeur implicite !

- Exemple : soit la classe suivante :

```
// point.h
class POINT
{
  private:
    int x, y;
  public:
    POINT( int x0, int y0 );
};
```

- Le programme suivant sera rejeté par le compilateur :

```
#include "point.h"
int main()
{
   POINT a( 3,4 );
   POINT b;
    ...
}
```

Constructeur implicite

- Attention : s'il existe au moins un constructeur explicite, le compilateur ne génère plus de constructeur implicite !

- Exemple : soit la classe suivante :

```
// point.h
class POINT
{
  private:
    int x, y;
  public:
    POINT( int x0, int y0 );
};
```

- Le programme suivant sera rejeté par le compilateur :

```
#include "point.h"
int main()
{
   POINT a( 3,4 );
   POINT b;
    ...
}
```

$\Rightarrow$ Il n'existe pas de constructeur sans arguments dans la classe !

Sommaire

UNIVERSITÉ DE ROUEN

Destructeur

- Lors de la destruction de l'objet, une fonction particulière est appelée automatiquement, avant la libération de la mémoire : le **destructeur**.
- **Un seul destructeur** possible par classe, aucun argument.
- Pas de valeur renvoyée.
- Nom : nom de la classe, précédé de ~
- Comme pour le constructeur, en l'absence de déclaration explicite, le compilateur synthétise un **destructeur implicite** (qui ne fait rien).

```
class POINT
{
  private:
    int x, y;
  public:
    POINT( int x0, int y0 );  // constructeur
    ~POINT();                 // destructeur  explicite
};
```

Rôle du destructeur

- Libération des ressources allouées lors de la création de l'objet.
- Les resources peuvent être :
  - de la mémoire (principalement),
  - des resources matérielles (accès réservé à un périphérique),
  - des resources logicielles (accès à un service offert par l'O.S.).
- Remarque : pour les classes triviales, on pourra s'en passer.

UNIVERSITÉ DE ROUEN

Sommaire

UNIVERSITÉ DE ROUEN

Allocation mémoire et constructeur

- Le constructeur doit :
  1. Effectuer l'allocation mémoire pour ses attribut membres.
  2. Les initialiser.
- Deux cas de figures : les attributs sont :
  - de type automatique $\Rightarrow$ l'allocation se fait... automatiquement !
  - des pointeurs $\Rightarrow$ Il faudra explicitement faire l'allocation mémoire pour les objets pointés.

UNIVERSITÉ DE ROUEN

Allocation mémoire et constructeur : exemple 1

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR c;
  public:
    PIXEL();
};
```

Allocation mémoire et constructeur : exemple 1

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR c;
  public:
    PIXEL();
};
```

- Implémentation :

```
// pixel.cpp
#include "pixel.h"

PIXEL::PIXEL()
{
    x = y = 0;
}
```

Allocation mémoire et constructeur : exemple 1

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR c;
  public:
    PIXEL();
};
```

- Implémentation :

```
// pixel.cpp
#include "pixel.h"

PIXEL::PIXEL()
{
    x = y = 0;
}
```

- A la création d'un PIXEL, il y aura automatiquement :
  - Allocation automatique d'une zone mémoire pour mémoriser x,y,c.
  - x et y seront initialisés à 0.
  - Le constructeur sans arguments de COULEUR sera exécuté.

Allocation mémoire et constructeur : exemple 2

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR* c;
  public:
    PIXEL();
};
```

Allocation mémoire et constructeur : exemple 2

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR* c;
  public:
    PIXEL();
};
```

- Implémentation :

```
// pixel.cpp
#include "pixel.h"

PIXEL::PIXEL()
{
    x = y = 0;
    c = new COULEUR;
}
```

Allocation mémoire et constructeur : exemple 2

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR* c;
  public:
    PIXEL();
};
```

- Implémentation :

```
// pixel.cpp
#include "pixel.h"

PIXEL::PIXEL()
{
    x = y = 0;
    c = new COULEUR;
}
```

- A la création d'un `PIXEL`, il y aura automatiquement :
  - Allocation automatique d'une zone mémoire pour mémoriser `x`, `y` et le pointeur `c`.
  - `x` et `y` seront initialisés à 0.
  - Le constructeur **doit** effectuer l'allocation mémoire pour `c`.

Destructeur : objets dynamiques

- Dans ce cas, il sera nécessaire de doter la classe d'un destructeur, pour **libérer** la mémoire allouée.

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR* c;
  public:
    PIXEL();
    ~PIXEL();
};
```

Destructeur : objets dynamiques

- Dans ce cas, il sera nécessaire de doter la classe d'un destructeur, pour **libérer** la mémoire allouée.

- Déclaration :

```
// pixel.h
#include "couleur.h"

class PIXEL
{
  private:
    int x, y;
    COULEUR* c;
  public:
    PIXEL();
    ~PIXEL();
};
```

- Implémentation :

```
// pixel.cpp
#include "pixel.h"

PIXEL::PIXEL()
{
    x = y = 0;
    c = new COULEUR;
}
PIXEL::~PIXEL()
{
    delete c;
}
```

Pour finir : détail de syntaxe C++

- La syntaxe du C++ est parfois ambigue...
- Soit le code suivant : que signifient ces lignes ?

```
a b;
```

```
c d( 1, 2, 3 );
```

```
e = f( 1, 2, 3 );
```

```
g h();
```

Exercice 1 : classe PIXEL

- Reprendre la classe PIXEL (version 1) et ajouter un constructeur permettant d'initialiser le pixel avec une coordonnée et une couleur.

```
class PIXEL
{
   private:

      int x, y;
      COULEUR c;
   public:
      PIXEL( int x, int y, COULEUR c );
    ...
};
```

Exercice 2 : classe RECTANGLE

- Reprendre la classe RECTANGLE du cours n° 2, et lui ajouter :
  - un constructeur sans arguments qui initialise largeur et hauteur de façon à avoir une surface unitaire et un rapport égal à $\sqrt{2}$,
  - un constructeur à 2 arguments (largeur & hauteur).

```
class RECTANGLE
{
   private:
      float haut, larg;
   public:









};
```

Exercice 3 : constructeurs

```
class XYZ {
   public:
      XYZ( int a, string b );    // const. 1
      XYZ( int a );              // const. 2
      XYZ( PLOUF a, string b ); // const. 3
};
```

Parmi les lignes ci-dessous, lesquelles provoqueront des erreurs de compilation ? Pourquoi ? Sinon, donner le n° du constructeur utilisé.

```
int main()
{
   PLOUF p; // ok
   XYZ x1;
   XYZ x2( 3 );
   XYZ x3( "bonjour", 2 );
   XYZ x4( p, "abcd" );
   XYZ x5( p );
   XYZ x6( 2, "aaaaa" );
}
```