Langage C++

"Programmation Orientée objet "

Introduction à C++

- Vous devez juste savoir, éventuellement, qu'il a été conçu en 1982 par **Bjarne Stroustrup**, ce qui aide pour trouver l'excellente bible du C++, écrite par lui-même.
- *Pourquoi du C++: c*'est avant tout une nécessité de répondre à des besoins générés par de gros projets.
  - modulaire: partager l'application en plusieurs module (partie)
  - Extensible: facilité avec laquelle un programme pourra être adapté pour satisfaire à une évolution des besoins
  - Réutilisable: possibilité d'utiliser certaines parties du logiciel dans une autre application.
  - Robuste: aptitude d'un logiciel à fonctionner même dans des conditions anormales d'utilisation « les exceptions ».
  - Portable: adaptation d'un logiciel à différentes conditions matérielles .

Introduction à C++

- Ceci est assez limité lorsqu'on emploie un langage simplement structuré tel que **C** ou **Turbo Pascal**.
- C++ ajoute au langage C des spécificités supplémentaires en lui permettant d’appliquer les concepts de la Programmation Orientée Objets « P.O.O qui se base sur la notion d’objet.».

Historique de la P.O.O

- La programmation orientée objet est apparue très tôt en informatique dans les années 1960 avec Simula67.
- Elle a trouvée son voie dans l'industrie au milieu des années 1980 avec les langages comme Smalltalk et C++.
- les méthodes d'analyses sont arrivées à maturité relativement tardivement vers 1990: UML ...
- Ce type de programmation se base sur la notion d'objet.

- Qu'est ce qu'un **Objet** ?.

Définition d'objet

- Un objet est une entité formée par les données (les champs) et des fonctions (méthodes) qui agissent sur ces données (notion de spécialisation des fonctions).
  - Objet = Méthodes + Champs "avec leur valeur"
  - Objet = Méthodes + Attributs
    - où les attributs sont les données avec leur valeur à un moment de l'application
- La P.O.O permet de définir et de modéliser une application en terme d'objets qui collaborent.
- Exemple d'objets de la vie courante:
  - un téléphone, ville, cafetière, fiches de paie, le bulletin de salaire, l'employé, l'employeur, la date d'émission, etc..

Attribut et État d'un objet

- Un objet est caractérisé par:
  - Les services proposés: se sont les méthodes de l'objet
  - ses champs, son État et son nom.
- les champs
  - Information qui qualifie l'objet qui le contient;
  - Peut être une constante.
- État
  - Valeurs instantanées de tous les attributs d'un objet
  - Évolue au cours du temps
  - Dépend de l'histoire de l'objet

Identité d'objet

champs variable

champs constant

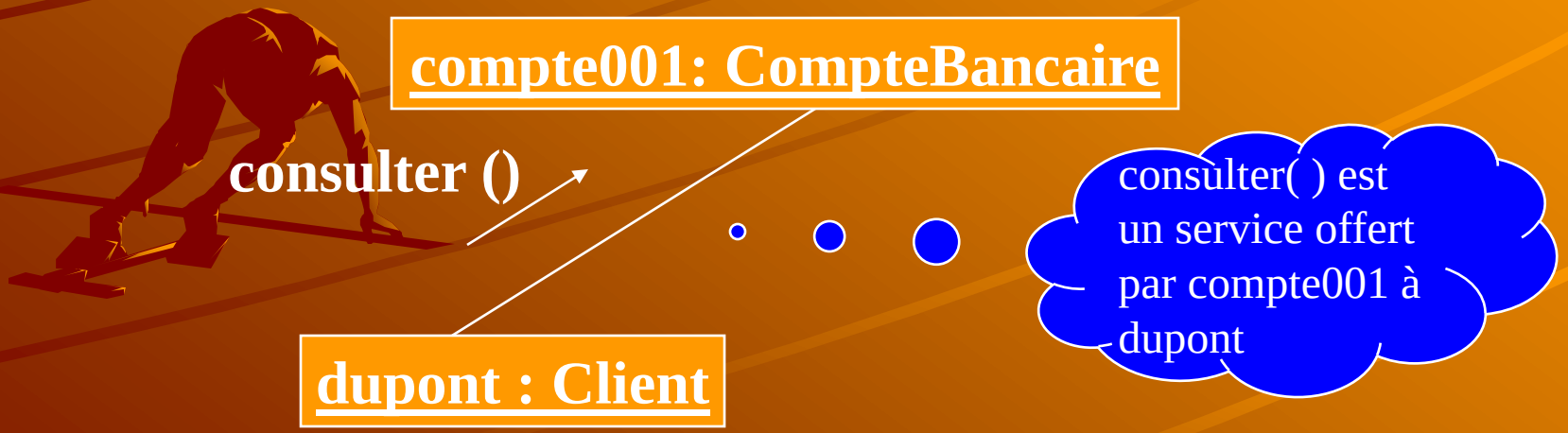
compte001 : CompteBancaire

solde : 10

DEBITAUTORISE : 1

Comportement d'un objet

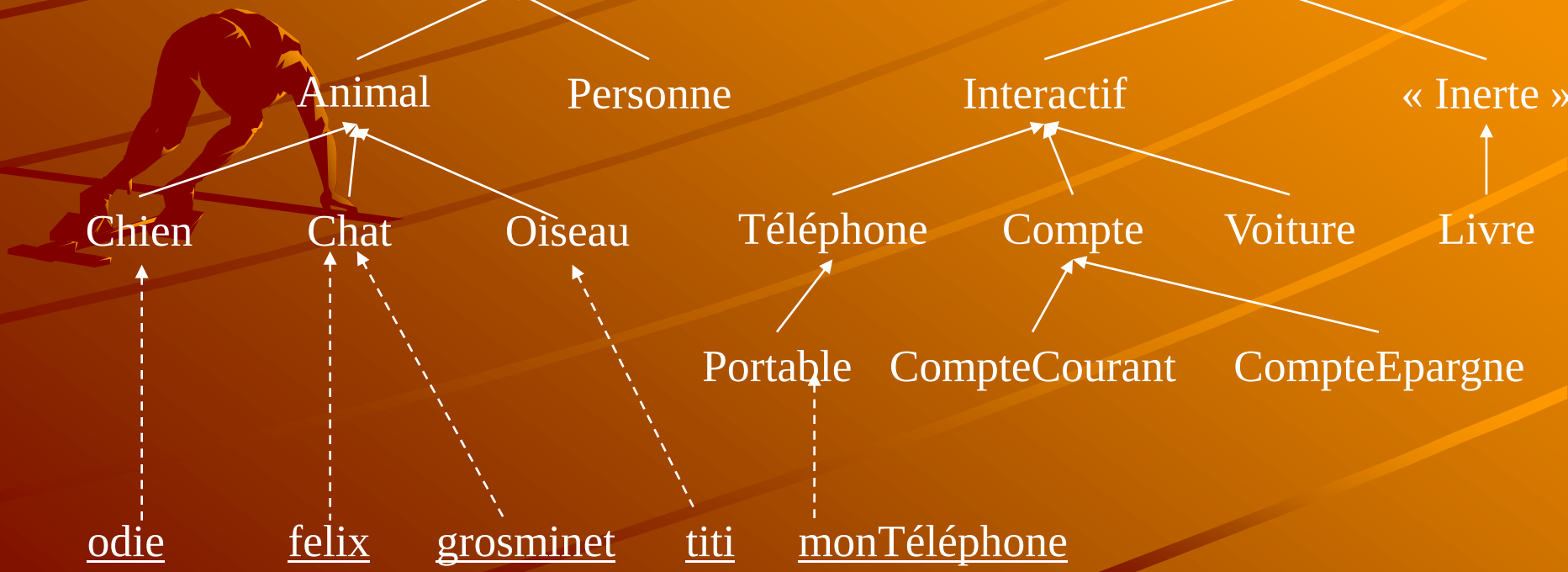
- Le comportement d'un objet décrit les actions et les réactions d'un objet
  - Se sont les services offerts par un objet
- Service = opération = méthode
- Un comportement est déclenché par un message (invocation de service) d'un objet par un autre objet.



Les objets sont rattachés à leur classe

On classe les objets par famille en faisant abstraction de quelques différences. On élabore ainsi une définition théorique de ce que doit être un objet de cette famille ou de cette classe.

les familles d'objets partagent un ensemble d'attributs et de services, en fonction de leur position dans l'arborescence.



Les classes

Une classe est une description d'une famille d'objets, qui ont :

Mêmes attributs

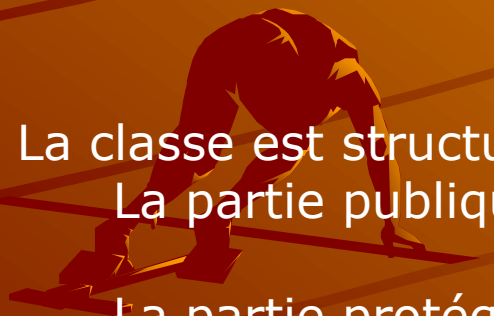
Même méthodes

} Spécification

- Une classe est une moule d'un ensemble d'objets qui partagent une structure commune (les attributs) et un comportement commun (les méthodes).
- Une classe est une collection de champs (*variables membres sans valeurs*), de *fonctions membres (les méthodes), qui leur sont attachées et* qui définissent le comportement d'un objet.
- Une classe est une entité qui permet de regrouper un ensemble de données et les méthodes qui leurs sont attachées.
- Une classe correspond aussi a un type de donnée comme int, double, elle correspond au type des objets qui lui sont liés.

Classe CompteBancaire

Nom de la classe	**CompteBancaire**	
Attributs	**solde**	**private**
	numCompte	**protected**
Opérations	**deposer()**	**public**
	retirer()	



La classe est structurée en 3 partie (les modes d'accès):

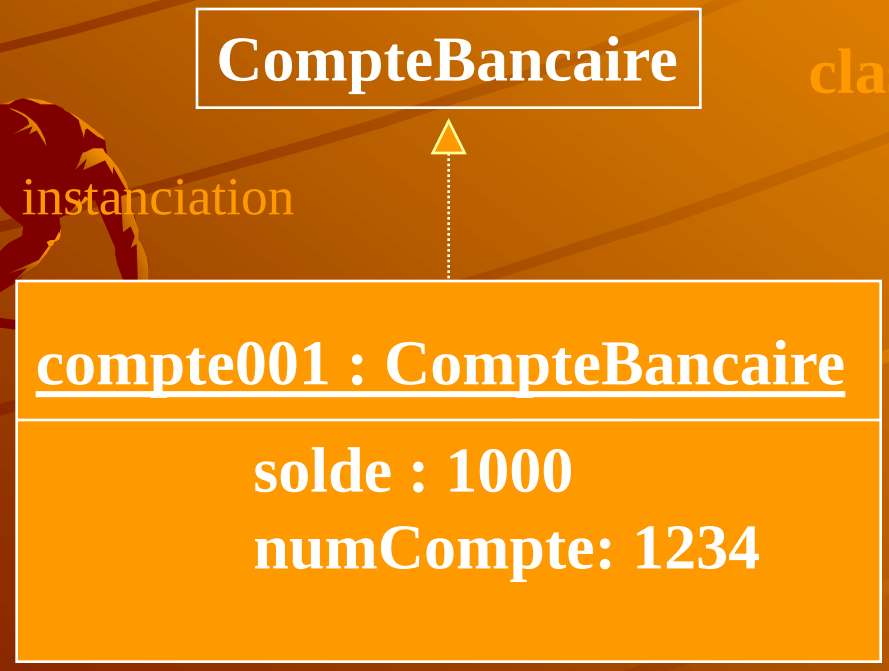
La partie publique: contient les déclarations visibles à tout utilisateur de cette classe.

La partie protégée: contient les déclarations visibles uniquement par les classes **héritant** de la classe de base.

La partie privée: contient les déclarations invisibles à toute autre classe différente de la classe de base.

Instance

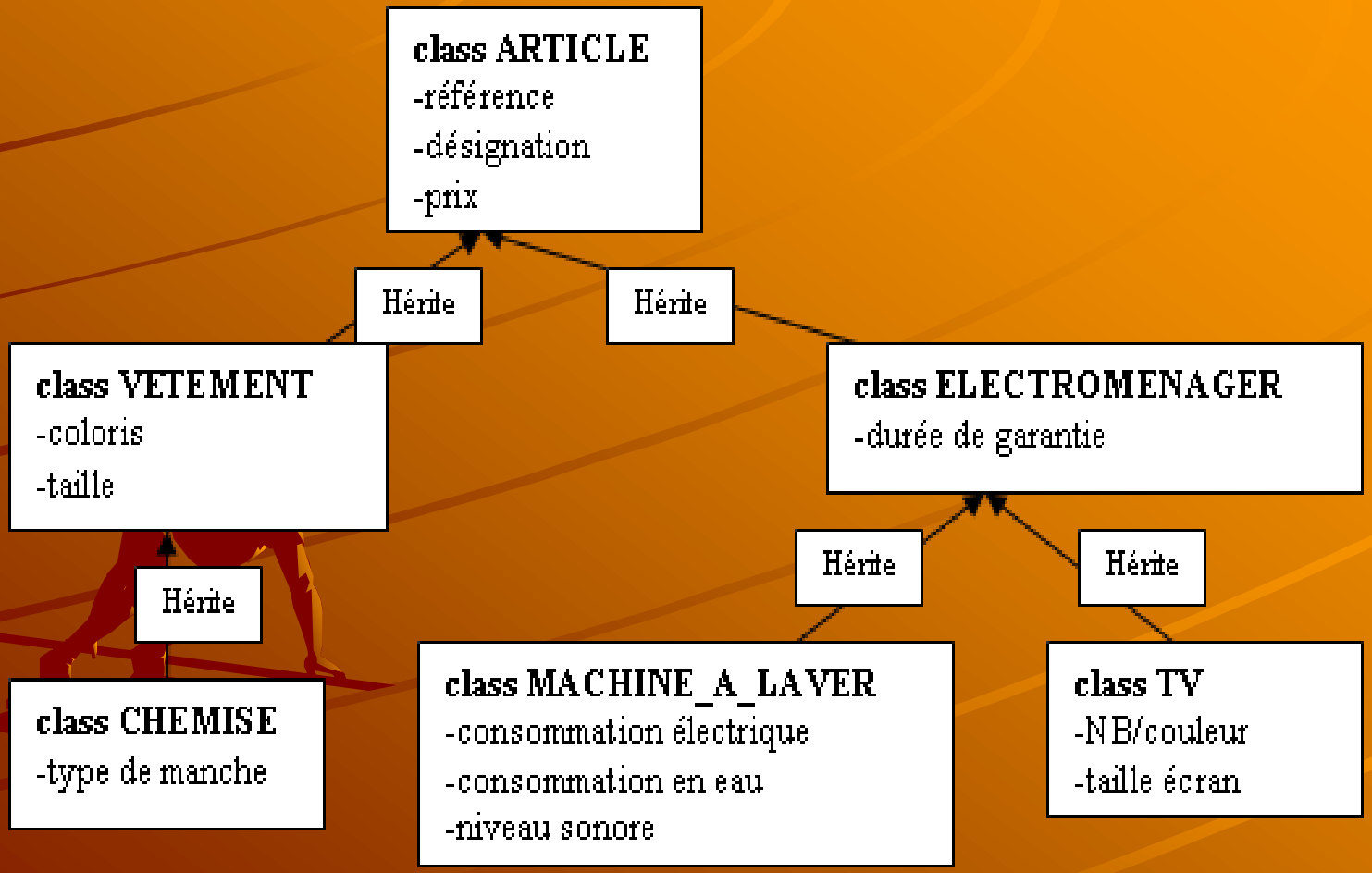
- Chaque objet appartient à une classe
- On dit que:
  - obj1 est un objet de la classe X
  - ou
  - Obj1 est une instance de la classe X



Notions importantes en P.O.O

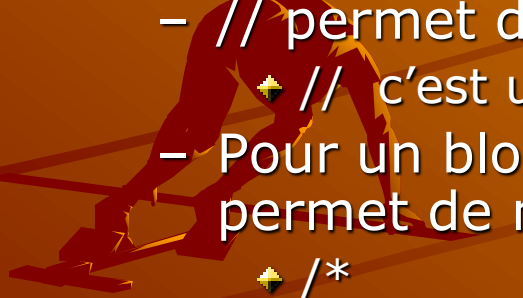
- Encapsulation des données.
  - **Ceci signifie qu'il n'est pas possible pour l'utilisateur d`un objet, d'accéder directement aux données (les modes d'accès protected et private).**
  - **L'utilisateur doit passer par des méthodes spécifiques écrites par le concepteur de l'objet, et qui servent d'*interface* entre l'objet et ses utilisateurs.**
  - **L'intérêt de cette technique est évident. L'utilisateur ne peut pas intervenir directement sur les données d'un objet, ce qui diminue les risques d'erreur, ce dernier devenant une "*boîte noire*".**
  - **En général cela permet de protéger les variables membres contre des modifications incohérentes**
- **Héritage.**
  - **L'héritage permet la définition d'une nouvelle classe à partir d'une classe existante.**
  - **Il est alors possible de lui adjoindre de nouvelles données, de nouvelles fonctions membres (méthodes) pour la spécialiser.**

Exemple d'héritage



Les spécifications C++

- Une instruction élémentaire est une expression terminée par le caractère ; (point virgule)
  - A = B; if (A == B) C = A +B;
- Un bloc est une suite d'instructions élémentaires délimitées par des accolades { et }
- Un bloc peut contenir un ou plusieurs blocs inclus
- Le langage C++ permet d'ajouter des commentaires dans les programmes.
  - // permet d'ignorer tout jusqu'à la fin d'une ligne
    - // c'est une ligne de commentaire
  - Pour un bloc de ligne, on utilise Le symbole /* ... */ qui permet de mettre en commentaire le bloc qui y inclus
    - /*
      ligne1 de commentaire…
      ligne n de commentaire
      */

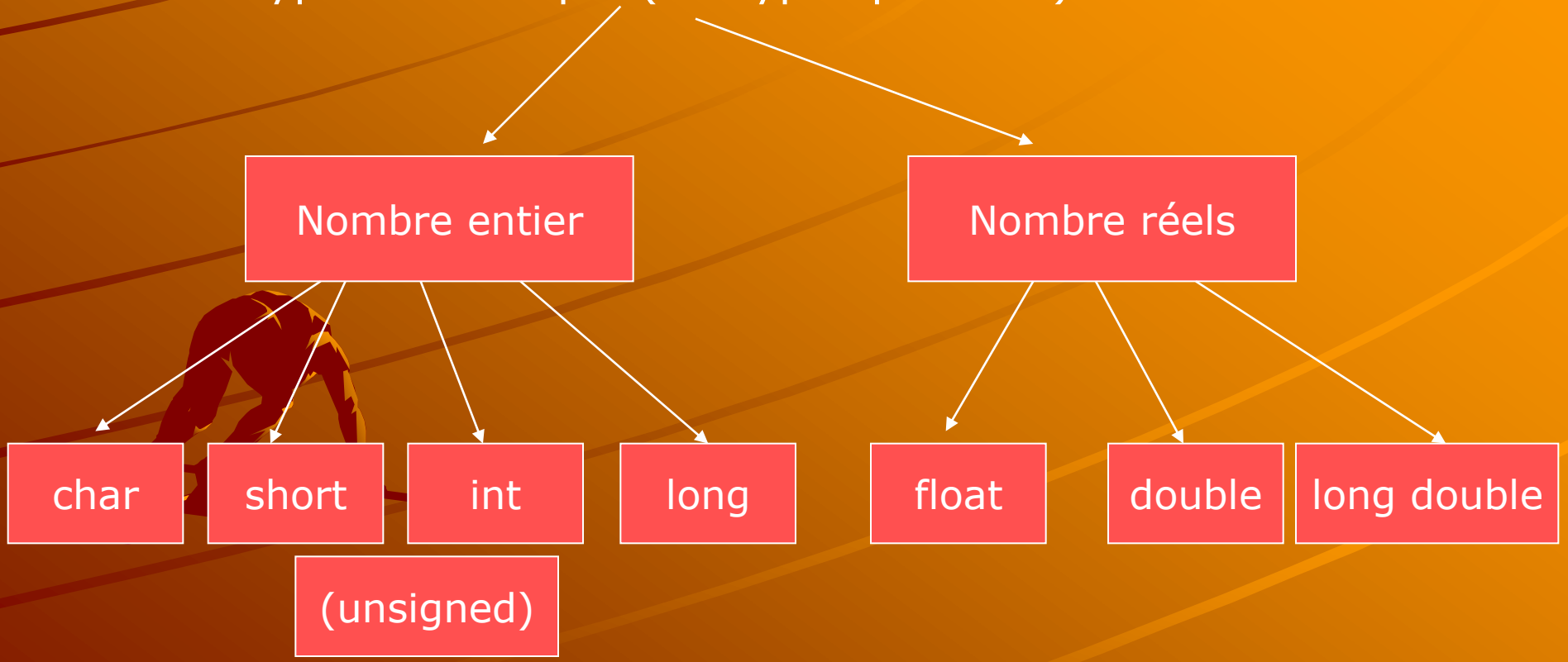


Spécifications: Définitions

- Identificateur:
  - nom formé de lettres et de chiffres, débutant par une lettre ou le _ considéré comme une lettre. (en pratique, on utilise moins de 31 caract.)
- Variable:
  - identificateur qui sert de nom à une région de la mémoire.
- Type:
  - attribut qui détermine les dimensions de l'espace mémoire (d'une variable par exemple) et la façon d'interpréter les 1 et 0.
- Une variable possède un nom, un type, une taille et une valeur
  - **Exemple**: int longueur = 10;
  - Déclaration de la variable Longueur dont le nom est longueur, de taille 4 octet, de type int et de valeur égale à 10.

Les spécifications C++

Les types numérique (les types primitifs)



Spécification C++

- La taille des types primitifs:

  char: 1 octet
  short: 2 octet
  int: 4 octet
  long: 4 octet
  float: 4 octet
  double: 8 octet
  long double: 8 octet

- Opération en C++:

  Addition: +
  Soustraction: –
  Multiplication: *
  Division: /
  Modulo: %

Spécification C++

- Opérateurs relationnels

  >, <, >=, <=

- Opérateurs d'égalité

  == ,(a==b a est égale à b)

  != ,(a != b a est différent de b)

- Opérateur d'affectation

  = ,(c = a+b c reçoit la valeur de la somme de a et b)

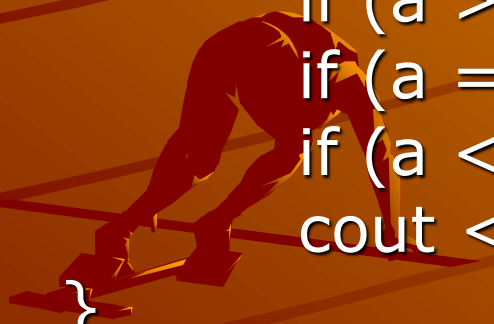
Les spécifications C++

- Pour effectuer les entrées/sorties (flux) dans les programmes, on utilise les fonctions
  - cout (pour afficher à l'écran des caractères)
  - cin (pour lire sur le clavier: entrer des valeurs ou caractère)

- Remarques:
  - il faut inclure le fichier <iostream.h> au début du programme pour pouvoir utiliser les entrées/sorties
  - L'opérateur d'insertion **<<** permet d'envoyer des valeurs dans un flot de sortie
  - L'opérateur d'extraction **>>** permet d'extraire des valeurs d'un flot d'entrée.

Exemple

```
#include<iostream.h>
  void main(){
        int a, b, c;
        cout<< "Entrer la valeur de a"<< endl;
        cin >> a;
        cout<< "Entrer la valeur de b"<< endl;
        cin >>b;
        if (a >b) c = a + b;
        if (a == b) c = a + 1;
        if (a < b)  c = a – b;
        cout <<"la valeur de c est: "  << c<< endl;
}
```

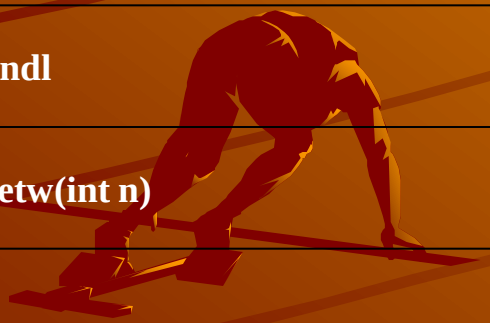


- notez l'absence de l'opérateur & dans la syntaxe du *cin*. Ce dernier n'a pas besoin de connaître l'adresse de la variable à lire.

Les spécifications C++

- Les manipulateurs sont des éléments qui modifient la façon dont les éléments sont lus ou écrits dans le flot.
- Les principaux manipulateurs sont :

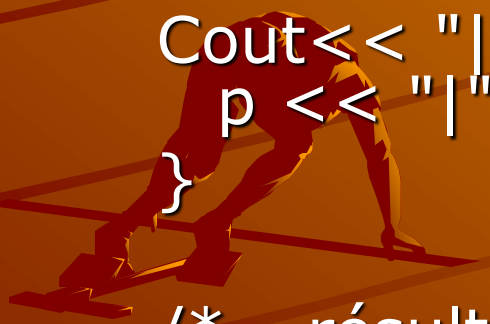
dec	**lecture/écriture d'un entier en décimal**
oct	**lecture/écriture d'un entier en octal**
hex	**lecture/écriture d'un entier en hexadécimal**
endl	**insère un saut de ligne et vide les tampons**
setw(int n)	**affichage de n caractères**
setprecision(int n)	**affichage de la valeur avec n chiffres avec éventuellement un arrondi de la valeur**
setfill(char)	**définit le caractère de remplissage**
flush	**vide les tampons après écriture**



Les spécifications C++

- **Exemple :**

```
#include <iostream.h>
#include <iomanip.h>
void main( ) {
int i=1234;
float p=12.3456;
cout << "|" << setw(8) << setfill('*') << hex
   << i << "|" <<endl;
Cout<< "|"  << setw(6) << setprecision(4) <<
   p << "|" << endl;
}

/*-- résultat de l'exécution ------------------
   |*****4d2|
|*12.35|
-----------------------------------------*/
```

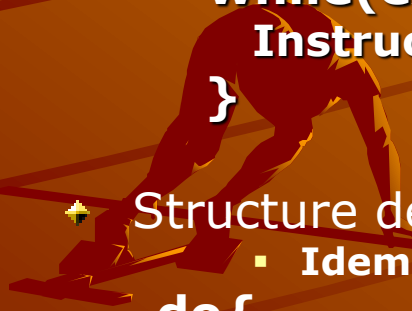


Spécification C++

- Structure de séléction: **if [/else]**
  - **if (expression) instruction1 [else instruction2]**
    - La valeur de l'expression est évaluée et, si elle est vraie, instruction1 est exécutée sinon c'est instruction2 qui est exécutée (si elle existe).
    - La clause **else** peut être omise et se rapporte toujours au dernier if visible.

- Structure de répétition: **while**
  - Permet de spécifier q'une action sera répétée tant que le résultat d'une expression reste vraie ou non nul

```
while(expression){
Instruction ou bloc d'instruction
}
```



- Structure de répétion: **do .. While**
  - **Idem que while**

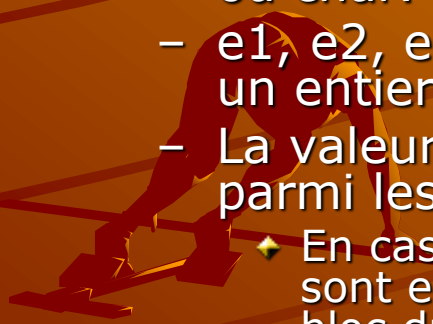
```
do{
instruction ou bloc d'instructions
}
while ( expression);
```

```
#include <iostream.h>

void main() {

      int a, b, compteur, i=0;
      cout << " valeur de a " <<endl;
      cin >> a;
      cout << " valeur de b "<<endl;
      cin >> b;
      if(a > b) compteur = a +1;
      else compteur = b+1;
      while(compteur < (2*b)) compteur = compteur +1;
      while(i != 10) {
       i = i +1;
      cout << " maintenant la valeur de i est égale à « <<i<<endl;
      }
       cout << " la valeur du compteur est : " << compteur<< endl;
      do {
      compteur += i;
      ++i;
      } while (i <= 20);
      cout << " la valeur du compteur est : " << compteur<< endl;
}
```

- La structure de sélection multiple switch:
  - permet un choix multiple (plusieurs if else) en fonction de l'évaluation d'une expression.

```
switch ( expression1 ) {
    case e1 : instruction1 ...
    case e2 : instruction2 ...
     ...
    case en : instructionn ...
    [default : instruction_default ...]
}
```

- **Description :**
  - expression1 doit donner pour résultat une valeur de type int ou char.
  - e1, e2, e3 ... sont des expressions constantes qui doivent être un entier de type int ou char.
  - La valeur de **expression1** est recherchée successivement parmi les valeurs des différentes constantes e1, e2, e3.
    - En cas d'égalité les instructions (facultatives) correspondantes sont exécutées jusqu'à une instruction break ou jusqu'à la fin du bloc du switch (et ceci indépendamment des autres conditions case).
    - S'il n'y a pas de valeur correspondante, on exécute les instructions du cas default (s'il existe).



Spécification C++

```
void main(){
        char car;
        int somme =0;
        cin >>car;
        switch (car) {
             case 'a' :
             case 'A' :
             case 'e' :
             case 'E' : somme++;    break;
             case 'x' :  somme--;      break;
             default  : somme = 10;
            }
        cout << " valeur de somme est égale à" <<somme<<endl;
}
```

- **Remarque**:
  - L'instruction **break** provoque le passage à l'instruction qui suit immédiatement le corps de la boucle *while*, *do-while*, *for* ou *switch*. (on sort complétement de la boucle)

Spécification C++

- L'opérateur conditionnel **?:**
  - Il permet d'écrire des expressions dont le résultat est fonction de certaines conditions.

Syntaxe :

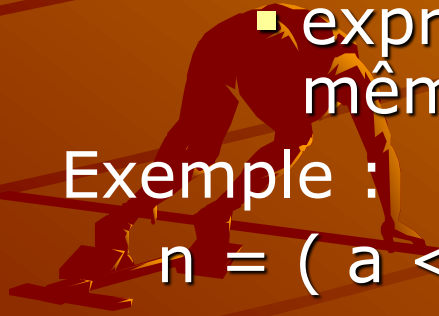
(expression) ? expression1 : expression2 ;

- Si expression est vrai, le résultat est expression1 sinon le résultat est expression2.
- expression1 et expression2 doivent être du même type.

Exemple :

n = (a < 0) ? -a : a ;

- n contient la valeur absolue de a.
- <=> if (a<0) n=-a ; else n =a;



Spécification C++

- Opérateurs d'affectation: ils permettent de simplifier les expression d'affectation

  += ,(a += b <-> a=a+b)

  -= ,(a -= b <-> a=a-b)

  *= ,(a *= b <-> a=a*b)

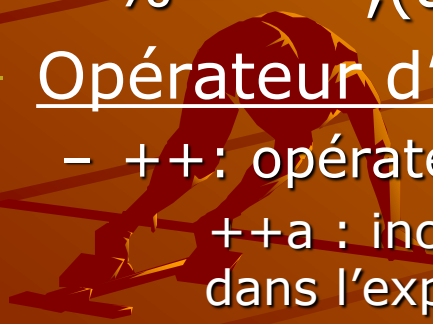
  /= ,(a /= b <-> a=a/b)

  %= ,(a %= b <-> a=a%b)

- Opérateur d'incrémentation et de décrémentation:
  - ++: opérateur pré-incrémentation

    ++a : incrémenter **a** de 1, puis utiliser la nouvelle valeur de **a** dans l'expression où **a** réside
  - ++: opérateur post-incrémentation

    a++ : utiliser la valeur courante de a dans l'expression où a réside, puis incrémenter a de 1

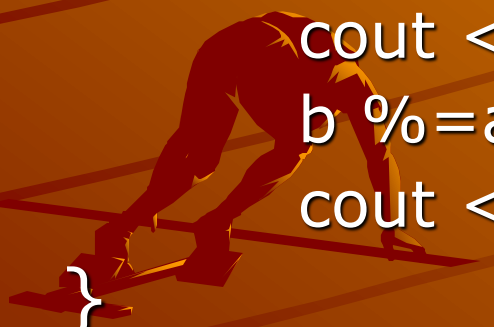


Spécification C++

- --: opérateur pré-décrémentation
  - --a : décrémenter **a** de 1, puis utiliser la nouvelle valeur de **a** dans l'expression où **a** réside
- --: opérateur post-décrémentation
  - a-- : utiliser la valeur courante de **a** dans l'expression où **a** réside, puis décrémenter **a** de 1

Les spécifications C++

```
#include <iostream.h>
void main() {
        int a=5, b=10, c, d;
        a +=b/2;
        ++b;
        c =b+a;
        cout << " valeur de c " << c <<endl;
        b %=a;
        cout << " valeur de b "<< b <<endl;
}
```



Les spécifications C++

- Structure de répétition: for

Syntaxe :

```
for ( [expression1] ; [expression2] ; [expression3] ){
    instruction ou bloc d'instructions
}
```

Description :

- instruction1 est répétée tant que la valeur de expression2 est vraie.
- expression1 sert à initialiser les variables de la boucle.

Exemple:

```
int somme;
for (i=0; i<10; i++)   somme += i*i;
```

Spécification C++

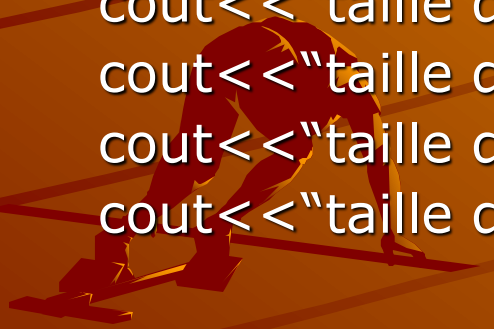
- L'opérateur de taille:
  - L'opérateur **sizeof** est utilisé pour déterminer la taille (en octets) d'une variable ou d'un type.

Syntaxe :

- sizeof(variable) ou sizeof variable ou sizeof(type)

Exemple:

```
int a=12; double b=14.4; char c='M';
cout<<"taille de a est:"<<sizeof(a)<<endl;
cout<<"taille de b est:"<<sizeof(b)<<endl;
cout<<"taille de c est:"<<sizeof(c)<<endl;
cout<<"taille du string hello est:"<<sizeof("hello")<<endl;
```



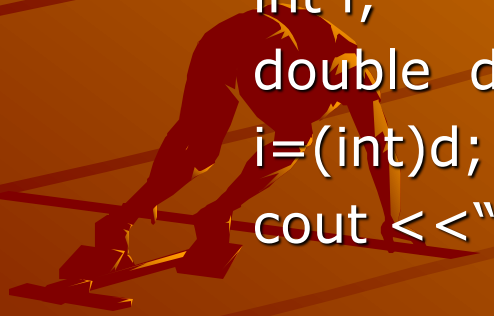
Il faut inclure la bibliothèque (ou le fichier header : iostream.h):

Spécification C++

- C++ fournit aussi le type **bool** dont les valeurs sont **true** ou **false**
  - **bool var;**
    - **Var est une variable de type bool**
    - **var est égale à true ou false**
- Les expressions booléennes
  - Se sont des expressions de type bool
  - **&&** réalise le "ET booléen" entre 2 expressions. Retourne true si les 2 expressions valent true, false sinon.
  - **||** réalise le "OU booléen" entre 2 expressions. Retourne vrai si l'une ou l'autre (ou les deux), des 2 expressions valent true, false sinon.
- exemple:
  - (a >= 5 && a <= 10)
    - retourne true si a est compris entre 5 et 10 sinon false.
  - a > 5 || b == 0
    - retourne true si a est supérieur à 5 ou si b est égal à 0.
    - l'expression b == 0 ne sera pas évaluée si a est supérieur à 5 (évaluation courte).

Les spécifications C++

- En C++, On peut explicitement demander une conversion dans un type désiré.
  - Cette opération s'appelle: casting ou transtypage.
  - Une expression précédée par un nom de type entre **( )** (parenthèses) provoque la conversion de celle ci dans le type désiré.

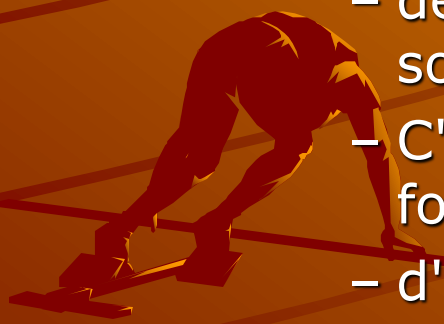


```
int i;
double  d=2.55;
i=(int)d; // conversion du double d en int
cout <<"valeur de i est: "<<i<<endl;
```

Les spécifications C++

DEFINITION DE VARIABLES

- En C++, on peut déclarer les variables ou fonctions n'importe où dans le code.
- La portée de telles variables va de l'endroit de la déclaration jusqu'à la fin du bloc courant.
- Ceci permet :
  - de définir une variable aussi près que possible de son utilisation afin d'améliorer la lisibilité.
  - C'est particulièrement utile pour des grosses fonctions ayant beaucoup de variables locales.
  - d'initialiser, une variable ou une constante, avec une valeur obtenue par calcul ou par saisie, dans n'importe quel endroit du code.



Les spécifications C++

- **Exemple :**

```
#include <iostream.h>
void main() {
    int i=0;       // définition d'une variable
    i++;           // instruction
    int j=i;       // définition d'une autre variable
    j++;           // instruction
    int somme(int n1, int n2); // déclaration d'une fonction
    cout <<i << " "<<j<<  " "<<somme(i, j)<<endl;
    cin >> i;
    const int k = i; // définition d'une constante initialisée
                     // avec la valeur saisie
    Cout << "valeur de k est:  "<< k <<endl;
}
```

- créer la fonction somme(int, int) ; // ???

Les spécifications C++

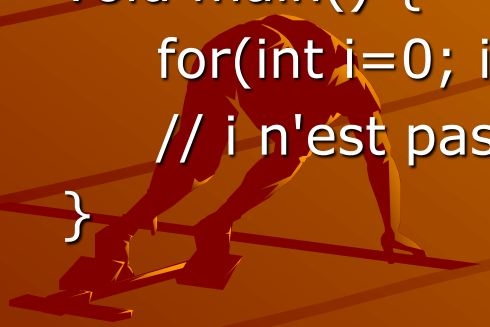
- VARIABLE DE BOUCLE

- On peut déclarer une variable de boucle directement dans l'instruction *for*.

- Ceci permet de n'utiliser cette variable que dans le bloc de la boucle.

Exemple :

```
#include <iostream.h>
void main() {
     for(int i=0; i<10; i++)   cout << i << endl;
     // i n'est pas utilisable à l'extérieur du bloc for
}
```

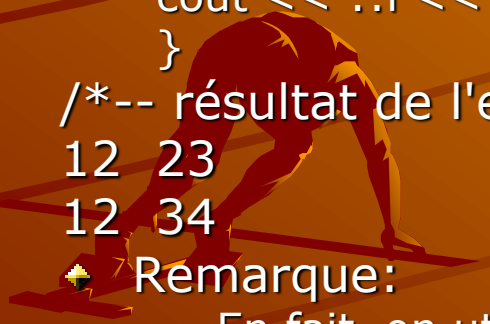


Les spécifications C++

- VISIBILITE DES VARIABLES

  L'opérateur de résolution de portée :: permet d'accéder aux variables globales plutôt qu'aux variables locales.

```
#include <iostream.h>
int i = 11;  // variable globale
void main() {
     int i = 34;
          {  int i = 23;
          ::i = ::i + 1;
          cout << ::i << "  " << i << endl;
          }
     cout << ::i << "  " << i << endl;
     }
/*-- résultat de l'exécution ------------------
12  23
12  34
```



- Remarque:
  - En fait, on utilise beaucoup cet opérateur pour définir hors d'une classe les fonctions membres ou pour accéder à un identificateur dans un espace de noms.
  - L'utilisation abusive de cette technique n'est pas une bonne pratique de programmation (lisibilité). Il est préférable de donner des noms différents plutôt que de réutiliser les mêmes noms.

Les spécifications C++

- **LES TYPES COMPOSES**
  - En C++, comme en langage C, le programmeur peut définir des nouveaux types en définissant par exemple des *struct* , *enum* ou *union*.
  - Mais contrairement au langage C, l'utilisation de ***typedef*** n'est plus obligatoire pour renommer un type.

Exemple :

```
struct FICHE {          // définition du type FICHE
    char *nom, *prenom;  int  age;
    };
```

- En C, il faut ajouter la ligne : typedef struct FICHE; pour construire un synonyme du type Fiche
  - **typedef struct fiche fiche;** permet de construire un synonyme de la structure fiche, en C,
  - par contre, on écrit simplement **fiche**, en C++, pour créer des variables de type fiche.
    - FICHE adherent, *liste; //déclaration de deux variables de type Fiche

variables références

- En plus des variables normales et des pointeurs, le C++ offre **les variables références.**
- Une variable référence permet de créer une variable qui est un "synonyme" d'une autre,
- Une variable et sa référence désignent le même emplacement mémoire
- Donc, une modification de l'une affectera le contenu de l'autre.

```
int i, *ptr;
int &ir = i; // ir est une référence à i ou synonyme de i
i=1;
cout << "i= " << i << " ir= " << ir << “adresse de i: ” <<&i << “adresse de ir: ”  << &ir <<endl;
```

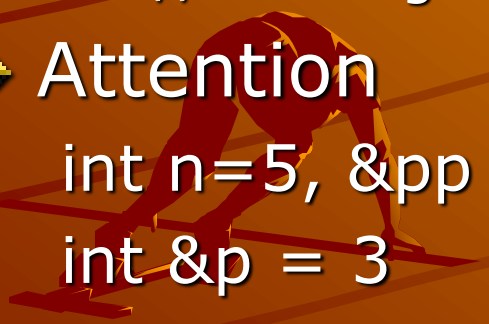
// affichage de : i= 1 ir= 1 adresse de i: 15F adresse de ir 15F

- Une fois une référence est déclarée et initialisée, on ne peut la modifier: ir est une référence sur i et non sur autre variable

variables références

```
ir=2;
cout << "i= " << i << " ir= " << ir << endl;
    // affichage de :   i= 2  ir= 2
ptr = &ir;
*ptr = 3;
cout << "i= " << i << " ir= " << ir << endl;
    // affichage de :   i= 3  ir= 3
```

- Attention

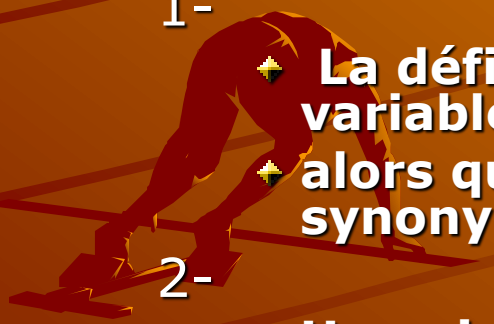


```
int n=5, &pp = n;
int &p = 3
    // initialisation d'une référence par une constante: incorrecte
float x = 5;
pp = x;
    // donc affectation de la valeur de int(x) à pp et non changement de
       référence de n à x pour pp
```

variables références

- Une variable référence doit **obligatoirement** être initialisée et le type de l'objet initial doit être le même que l'objet référence.
- Après l'initialisation, on ne peut pas changer la valeur d'une référence (c.a.d: la relation entre référence et la variable référée et fixée une fois pour toute).
- Différences entre pointeur et une référence:

1-



- **La définition d'un pointeur est une création de variable, avec un emplacement mémoire propre,**
- **alors que la définition d'une référence ne crée qu'un synonyme et donc pas réservation de mémoire.**

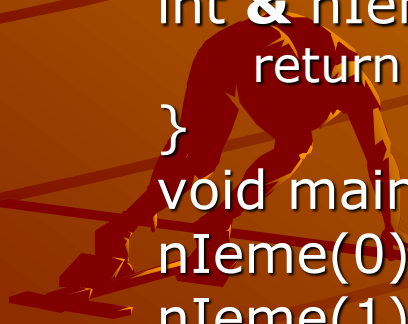
2-

- **Un pointeur étant une variable, il est possible d'en modifier le contenu, et le même pointeur peut permettre d'accéder successivement à des variables différentes.**
- **Par contre, L'association entre une référence et la variable qu'elle désigne est fixée définitivement lors de l'initialisation de celle-ci.**

variables références

- Intérêts:
  - passage des paramètres par référence: dans le cas des fonctions
  - utilisation d'une fonction en lvalue: en utilisant surtout **la surdéfinition d'opérateur (l'opérateur &)**, on récupère le retour de la fonction qui retourne une référence "donc une adresse" du résultat.
- **Exemple :**

```
int t[20];
int & nIeme(int i) {
    return t[i];
}
void main() {
nIeme(0) = 123; // utilisation de la fct nIeme comme une lvalue
nIeme(1) = 456;
cout << t[0] << " " << ++nIeme(1);
}
    // -- résultat de l'exécution ----------
    // 123 457
```



variables références

- En C, on aurait pu écrire le programme suivant :

```
int * nIeme(int i) {
    return &t[i];
}
void main() {
    *nIeme(0) = 123;
    *nIeme(1) = 456;
    cout << t[0] << " " <<++(*nIeme(1)) <<
      endl;
}
```

Allocation dynamique de la mémoire

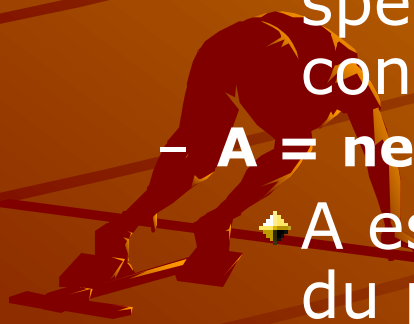
- **Le C++ met à la disposition du programmeur deux opérateurs new et delete (comme malloc et free en C mais en plus simple)**
  - **L'opérateur new permet d'allouer de la mémoire qu'on lui demande et l'initialise (permet de créer des tableaux dynamiques).**
  - **Il retourne l'adresse de début de la zone mémoire allouée.**
- **int *ptr1, *ptr2, *ptr3;**
- **Allocation dynamique d'un entier**
  - **ptr1 = new int; // ptr1 est un pointeur qui pointe vers un int**
- **Allocation dynamique d'un tableau de 10 réels**
  - **ptr2 = new double [10];**
- **Allocation d'un entier avec initialisation**
  - **ptr3 = new int(10);**

Allocation dynamique de la mémoire

- Exemple de code:

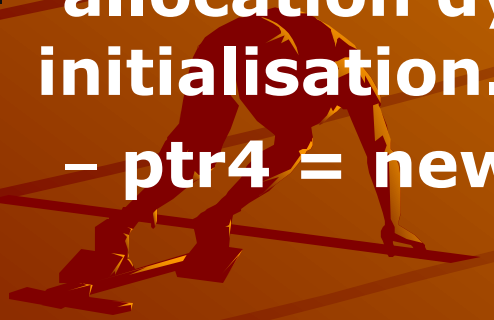
```
int taille;
int *A;
cout << "entrer la taille de votre tableau A:" << endl;
```

  - **cin >> taille;**
    - on construit d'une façon dynamique un tableau avec la donnée taille, on a pas à spécifier que taille est une constante pour construit notre tableau.
  - **A = new int[taille];**
    - A est un pointeur qui pointe vers l'adresse du premier élément du tableau crée dynamiquement par l'opérateur new, dont la dimension est taille.



Allocation dynamique de la mémoire

- **struct date {int jour, mois, an; };**
- **date *ptr4, *ptr5, *ptr6, d = {25, 4, 1952};**
- **allocation dynamique d'une structure**
  - **ptr4 = new date;**
- **allocation dynamique d'un tableau de structure**
  - **ptr5 = new date[10];**
- **allocation dynamique d'une structure avec initialisation.**
  - **ptr4 = new date(d);**



Allocation dynamique de la mémoire

- En cas d'erreur d'allocation par new, un message d'erreur est affiché (exception bad_alloc)
- L'opérateur delete permet de libérer la mémoire qui est allouée par l'opérateur new
- Exemple
  - char *ptr = new char [10];
    - // création d'un tableau dynamique de 10 char
    - Initialisation de la dimension du tableau via ptr peut être n'importe où dans le programme, ce qui n'était pas le cas des tableaux
  - for(int i=0, i < 10; i++) ptr[i] = 'A';
  - delete[] ptr;
    - //la place mémoire pointée par ptr est libérée

Exemple

```
struct date {int jour, mois, an; };
void main(){
    date *ptr4, *ptr5, d = {25, 4, 1952};
    ptr4 = new date;
    ptr5 = new date[10];
    ptr4->jour = 22;
    ptr4->mois = 04;
    ptr4->an = 2005;
    cout << "date de naissance par ptr4 est: "<<ptr4->jour<<"/"<<ptr4-
       >mois<< "/"<<ptr4->an<<endl;
    ptr4 = new date(d);
    cout << "date de naissance par ptr4 est: "<<ptr4->jour<<"/"<<ptr4-
       >mois<< "/"<<ptr4->an<<endl;
    ptr5[0].jour = 01;
    ptr5[0].mois = 02;
    ptr5[0].an = 1999;
    cout << "date de naissance par ptr5[0] est:
       "<<ptr5[0].jour<<"/"<<ptr5[0].mois<< "/"<<ptr5[0].an<<endl;
    delete ptr4;
    delete[] ptr5;
}
```

Les fonctions

- Le langage C++ impose au programmeur de déclarer le nombre et le type des arguments de la fonction.
- Ces déclarations sont identiques aux prototypes de fonctions de la norme C-ANSI.
- La déclaration suivante *int f();* où *f* est déclarée avec une liste d'arguments vide est interprétée en C++ comme la déclaration *int f(void);*
- Une fonction dont le type de la valeur de retour n'est pas void, doit **obligatoirement** retourner une valeur.

Structure d'une fonction

- La forme générale (en ANSI) d'une fonction est :

```
Type_de_valeur_revoyée nom_de_fonction(liste des paramètres )
{
   déclarations et instructions
}
```

  - nom_de_fonction: tout identificateur valide
  - Type_de_valeur_renvoyée: type de données du résultat retourné de la fonction vers la fonction d'appel
  - Le type de la valeur renvoyée **void** indique qu'une fonction ne retourne aucune valeur.
  - Liste de paramètres: est une liste séparée par des virgules qui contient les déclarations des paramètres reçus par la fonction lorsqu'elle est appelée.
  - Si une fonction ne reçoit aucune valeur, cette liste est **void** ou simplement laissée vide
  - On doit explicitement spécifier un type pour chacun des paramètres de la liste de paramètres d'une fonction.
- Le programme principal est une fonction dont le nom doit impérativement être **main**.
- Les fonctions ne peuvent pas être imbriquées.

Exemple de Fonctions

```
#include<iostream.h>
        int somme(int a, int b){
            int z;
            z = a+b;
            return z;
         }
        void affiche(){
             int k = somme(12,4);
            cout <<"la somme de 12 et 4 est "<<k<<endl;
        }
        void main(){
            affiche();
            int r = somme(13, -7);
           cout << "la somme de 13 et -7 est égale: "<<r<<endl;
        }
```

Surdéfinition d'une fonction

- **En général**, une fonction se définit par :
  - son nom,
  - sa liste typée de paramètres formels,
  - le type de la valeur qu'elle retourne.
- Seuls les deux premiers critères sont discriminants. On dit qu'ils constituent la **signature** de la fonction.
- On peut utiliser cette propriété pour donner un même nom à des fonctions qui ont des paramètres différents et de les utiliser dans le même programme: c'est la **surdéfinition ou surcharge de fonction**

Surdéfinition (surcharge)d'une fonction

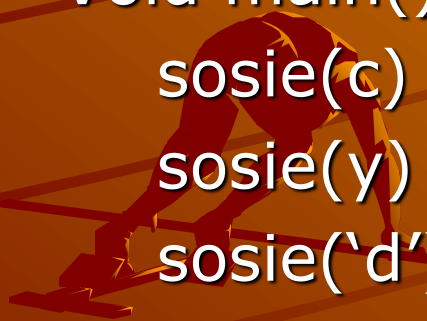
```
int somme( int n1, int n2){
    return n1 + n2;
}
int somme( int n1, int n2, int n3){
    return n1 + n2 + n3;
}
double somme( double n1, double n2){
    cout << "somme de double"<<endl;
    return n1 + n2;
}
void main() {
    cout << "1 + 2 = " << somme(1, 2) << endl;
    cout << "1 + 2 + 3 = " << somme(1, 2, 3) << endl;
    cout << "1.2 + 2.3 = " << somme(1.2, 2.3) << endl;}
```

- **Le compilateur sélectionnera la fonction à appeler en fonction du type et du nombre des arguments qui figurent dans l'appel de la fonction.**
- **Attention**, il y a des conversions implicites, dans le cas où l'argument passé à la fonction n'est pas défini dans le prototype de la fonction
  - char et short → int
  - float -→ double

Surdéfinition (surcharge)d'une fonction

- Autres Exemples:

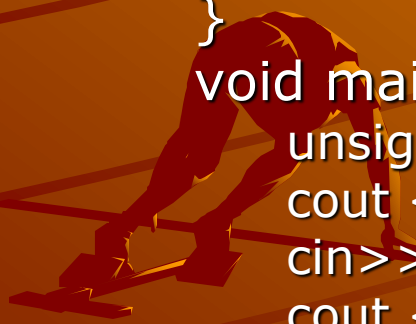
```
void sosie(int) ;   //sosie I
void sosie(double) ; //sosie II
void sosie(int &) ;//conduit à une erreur car une
  //ambiguïté pour le compilateur avec sosieI
char c ; float y ;
Void main(){
    sosie(c) ; // appel de sosie I après conversion de c en int
    sosie(y) ; // appel de sosie II après conversion de y en double
    sosie('d') ;// appel de la fct sosieI après conversion de 'd' en int
}
```



Fonction récursive

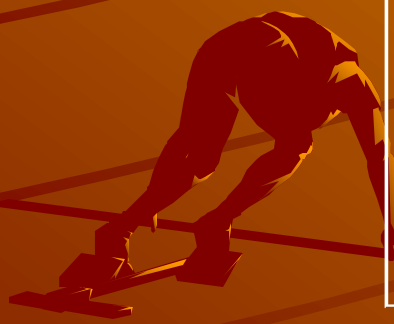
- Une fonction récursive:
  - est une fonction pouvant s'appeler elle-même directement ou indirectement par le biais d'une autre fonction.
- Exemple: factorielle d'un entier

```
include <iostream.h>
unsigned int factorielle(unsigned int nombre){
    if(nombre<=1) return 1;
    else return nombre*factorielle(nombre-1);
}
void main(){
    unsigned int nombre;
    cout <<"donner un nombre"<<endl;
    cin>>nombre;
    cout <<"le nombre factorielle de "<< nombre<<" est égale
     à: "<<factorielle(nombre)<<endl;
}
```



Fonctions usuelles de la bibliothèque mathématique

nom	type param	type retour	rôle
acos	double	double	arc cosinus
asin	double	double	arc sinus
atan	double	double	arc tangente
ceil	double	double	arrondi à la valeur supérieure
cos	double	double	cosinus
exp	double	double	exponentielle
fabs	double	double	valeur absolue d'un double
floor	double	double	arrondi à la valeur inférieure
log	double	double	logarithme néperien
sin	double	double	sinus
sqrt	double	double	racine carrée



*Pour utiliser ses fonctions dans un programme, on doit placer en en-tête la directive : #include <math.h>

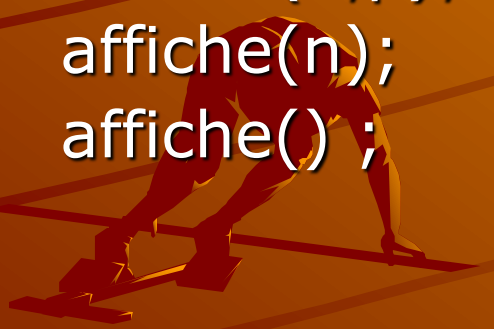
Valeur par défaut des paramètres d'une fonction

- Certains arguments d'une fonction peuvent prendre souvent la même valeur.
- Pour ne pas avoir à spécifier ses valeurs à chaque appel de la fonction,
  - C++ permet de déclarer et d'initialiser par défauts la valeur des arguments lors de la déclaration de la fonction prototype,
  - ce qui autorise l'appel de la fonction avec un nombre d'arguments différents.
- Si la valeur de l'argument n'est pas précisée au moment de l'appel de la fonction, le compilateur lui affecte une valeur par défaut.
- Remarque :
  - en C, l'appel d'une fonction doit contenir autant d'arguments que la fonction en attend.

Valeur par défaut des paramètres d'une fonction

Exemple :

```
# include <iostream.h>
void affiche(int, int = 2, int = 7);
void main() {
int n=10, p=20, t=30 ;
affiche(n,p,t);    // affiche 10 20 30
affiche(n,p);      // affiche 10 20 7
affiche(n);        // affiche 10 2 7
affiche() ;        // rejetée par le compilateur car
                   //le  premier argument n'a pas de
                   // valeur par défaut
}
void affiche(int a, int b, int c){
cout << a << " " << b << " " << c << endl;}
```



Valeur par défaut des paramètres d'une fonction

- **Remarques :**
  - une déclaration de la fonction affiche() de la forme :
    - void affiche (int=2, int, int=5) ; est interdite.
      - En effet, un appel tel que affiche(10, 20) pourrait alors être interprété de deux manières différentes : affiche(2,10,20) ou bien affiche(10,20,5).
  - Exemple concret calcul de TVA pour chaque personne, la fonction
    - calculTVA(char* , double=33.5)
    - Les arguments sont le nom de la personne et la valeur de TVA qui est devrait être constante.
  - Les paramètres par défaut sont obligatoirement les derniers de la liste.
  - Ils ne sont déclarés que dans le prototype de la fonction et pas dans sa définition.

Passage des paramètres à une fonction

- Il y a trois façon de passer les paramètres à une fonction:
  - Par valeur
  - Par adresse
  - Par référence
- Passage des paramètres par valeurs :
  - les paramètres formels sont initialisées par une copie des valeurs des paramètres réels.
  - Modifier la valeur des paramètres formels dans le corps de la fonction ne change pas la valeur des paramètres réels.
  - Remarque:
    - Lorsque un paramètre est de taille très volumineuse, l'opération de copie s'avère coûteuse en durée d'exécution et en place mémoire.
- Dans le cas de passage par adresse (utilisation des pointeurs):
  - Modifier la valeur des paramètres formels dans le corps de la fonction change la valeur des paramètres réels.
  - Remarque:
    - On réserve de la mémoire pour le paramètre passé par adresse, ce qui coûte de la place mémoire.

Exemples

- *void EchangeValeur(int A, int B)*
  { int T; T = A; A = B; B = T;}

- *void EchangePointeur(int *A, int *B)*
  *{ int T; T = *A; *A = *B; *B = T; }*

- *void main()*
  {
  int X=4, Y=128;
  EchangeValeur(X, Y); // passage par valeur
  cout << X <<endl;
  EchangePointeur(&X, &Y);
  / passage des arguments par adresse lors de l'appel d'une fonction, adresses des variables X et Y explicitement passées comme arguments */*
  cout << X <<endl;
  }

Passage des paramètres par référence

- On considère les déclarations suivantes:

  int n=1,

  int &p = n; // initialisation et déclaration de référence
  - p est une référence à la variable n.
  - p est un synonyme ou alias de la variable n.
  - n et p ont même type et même place mémoire.

  p=3;

  cout << n <<endl;
  - // toute modification de n est répercutée sur p et réciproquement
  - Résultat: n=3

  – Une fois une référence est déclarée et initialisée, on ne peut la modifier: p est une référence sur n et non sur autre variable
  - int &p = 3 // initialisation d'une référence par une constante: incorrecte
  - float x = 5; p = x; // interdit car p une référence à un int

Passage des paramètres par référence

- On sait que pour le passage des paramètres par adresse, on doit explicitement spécifier au compilateur l'adresse des paramètres d'une fonction.
- C++, permet de contourner cela, afin que le compilateur prend en charge la transmission des adresses des paramètres lui-même, on parle de transmission d'arguments par références.

- void echange(int& a, int& b){
  int c;
  cout << "debut echange: "<<a << b<<endl;
  c=a; a=b; b=c;
  cout << " fin echange "<<a << " "<<b <<endl;
- }

void main(){
  int n=10, p=20;
  cout << " avant appel: "<< n << " "<< p <<endl;
  echange(n, p) // attention ici pas de &n, &p
  cout << " après appel: "<< n << " " << p << endl;}

- Remarque:
  - a et b sont des références, alias ou synonymes des variables réels n et b
  - Le compilateur, lui-même, transmit les adresses des paramètres réels lors de l'appel de la fonction

Passage des paramètres par référence

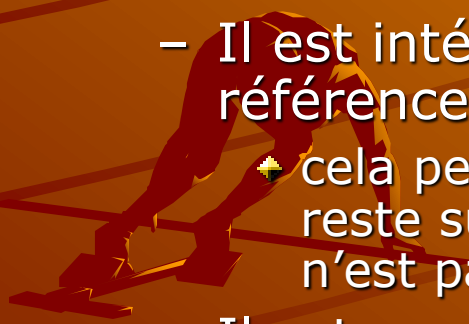
- Remarques:
  - On transmet à la fonction une référence d'une variable effectif,
    - Donc, il y a une transmission par adresse sans avoir à en prendre en charge soi-même la gestion
  - La transmission par référence simplifie l'écriture de la fonction
    - lors de l'appel de la fonction on ne pense plus si passage par valeur ou par adresse
    - On couple les deux propriétés de passage par valeur (la simplicité de l'écriture) et passage par adresse pour modification des valeurs des variables réelles.
  - un argument transmis par référence ne peut être soumis à aucune conversion(le référé et la référence doivent avoir le même type), sauf dans le cas de la référence à une constante.
    - Exemple:
      - float x;
      - const int &n = x;
        - Dans ce cas, il y a création d'une variable temporaire contenant le résultat de la conversion de x en int et placent sa référence dans n
      - Ceci est équivalent à:
        - float x; int temp = x; const int& n = temp;

Passage des paramètres par référence

- le passage par référence est plus efficace du point de vue de la gestion de la mémoire

- Il évite la recopie des arguments (car seule l'adresse mémoire de l'argument est communiqué à la fonction) surtout pour des structures de données de taille importante, par exemple les struct , vector …….

- Le passage par référence est dangereux car des fonctions peuvent modifier les données.
- Il est intéressant de passer des paramètres en tant que référence sur une constante (notation const type & p)
  - cela permet d'éviter le coût de la recopie des arguments et reste sûr car le compilateur vérifiera que le paramètre p n'est pas modifié dans le corps de la fonction.
- Il est possible pour une fonction de renvoyer comme résultat une référence (voir la première partie de variable référence)



Exemple

- Ecrire un programme qui permet de faire :
  - Déclarer un entier
  - Déclarer une référence vers cet entier
  - Déclarer un pointeur vers cet entier
- Dans les deux cas, imprimer la variable, l'adresse de la variable, la valeur pointée

Réponse

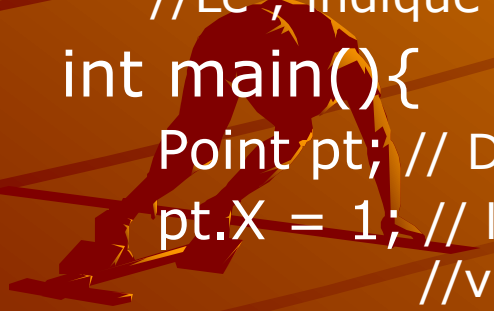
```
void main(){
    int AA=10;
    int &bb = AA; // bb est une référence à la variable AA
    int *cc = &AA; // cc est un pointeur qui pointe vers AA
    cout << "valeur de bb = " << bb << "
      adr de bb = " << &bb << endl;
    cout << "valeur de cc = " << cc << " adr
      de cc = " << &cc << " val de la variable
      pointée par cc = " << *cc << endl;
}
```

Les classes

- **une classe** (mot clé **class**) est une structure, pour laquelle en précise si les membres (champs) et les fonctions membres sont **publics, privés** ou **protégés** (mots clés **public, private** ou **protected): mécanisme de protection**
- Dans les structures tous les membres ou champs sont publics par défaut mais private pour les classes.
- D'après le principe d'encapsulation: on ne peut accéder aux données privées que par l'intermédiaire des méthodes.
- On considèrera donc, en général, les champs comme membres privées et les méthodes comme membres publics.
- une classes , comme les structures, est utilisée aussi comme un type de donnée, comme int., char...

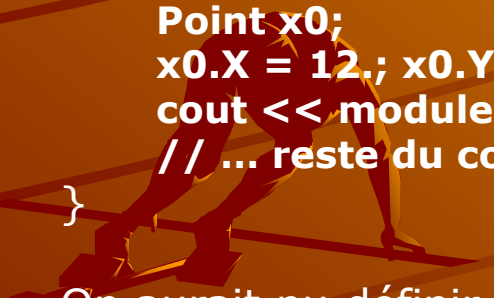
Définition d'une classe Point

```
class Point  //indique au compilateur qu'une classe est
             //définie dont le nom est Point
{
public : // niveau d'accès aux données
         // champs de la classe
    double X; // coordonnées x et y d'un point
    double Y;
}; // fin de la classe.
   //Le ; indique la fin de la définition de la classe
int main(){
    Point pt; // Déclaration d'un objet de type Point
    pt.X = 1; // le . est un opérateur d'accès à une
              //variable membre, ici la variable X
    pt.Y = 0;
    cout << " affichage du point pt(" <<pt.X << " ," << pt.Y
       << " )" <<endl;
}
```



Les classes

```
class Point {
        // définition de la classe comme précédemment // …
};

// définition de la fonction module, à l'extérieur de la classe
double module (double px, double py)
{
        return sqrt(px * px + py * py);
}

int main()
{
        Point x0;
        x0.X = 12.; x0.Y = 10.;
        cout << module(x0.X, x0.Y) << endl;
        // … reste du code
}
```



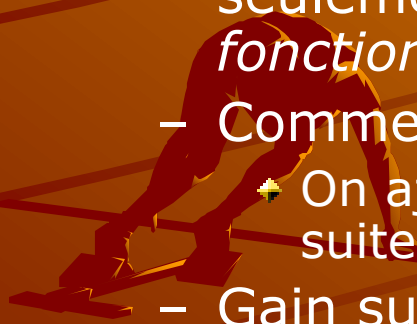
On aurait pu définir la fonction module ainsi :

```
double module (Point pt) {
        return sqrt(pt.X * pt.X + pt.Y * pt.Y);
        }
```

On a transmit un objet de type Point à la fonction module.

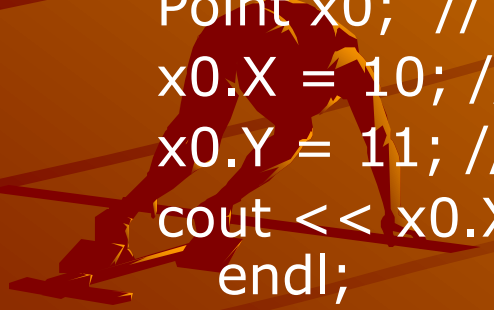
Les classes: Récapitulation

- Nous pouvons regrouper des variables dans une classe (champs)
- Nous pouvons accéder aux membres de la classe
- Nous pouvons transmettre un objet, d'une classe définie, à une fonction.
- Intégration des fonctions membres
  - But: Créer des objets complets qui contiennent non seulement *des variables membres* mais aussi *leurs fonctions* pour les traiter.
  - Comment?
    - On ajoute la définition de la fonction dans la classe à la suite des variables membres.
  - Gain supplémentaire:
    - Plus besoin de spécifier l'objet en argument dans la fonction
    - Plus besoin de l'opérateur d'accès pour l’accès aux champs.



Les classes

```
class Point{
public :
    double X; // coordonnées
    double Y; //
    double module() {// argument implicite:pointeur this
    return sqrt(X * X + Y * Y); // accès direct! }
    };
int main() {
    Point x0;  // création d'un objet X0 de type point
    x0.X = 10; // accès au champs X
    x0.Y = 11; // accès au champs Y
    cout << x0.X << " " << x0.Y << " " << x0.module()<<
       endl;
    return 0;
}
```



Les classes

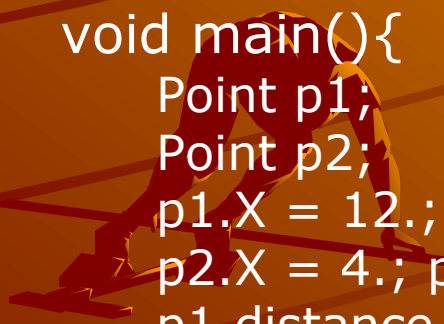
- Remarques importantes
  - Une fonction membre agit:
    - directement sur les variables membres de l'objet courant (via le pointeur this qui pointe vers l'objet courant) et non sur une copie.
  - Une fonction membre peut recevoir des arguments additionnels à l'argument implicite (this) et contenir des variables locales.
- Exemple: **//la fonction distance_point donne distance entre 2 points**

```
class Point {
// ...
double distance_point(double ox, double oy) {
        Point t;
        t.X = ox – X; t.Y = oy – Y;
        return sqrt(t.X*t.X + t.Y*t.Y);
}

void main(){
        double dist;
        Point p1;
        Point p2;
        p1.X = 12.; p1.Y = 3.;
        p2.X = 4.; p2.Y = 5.;
        dist = p1.distance_point(p2.X,p2.Y);}
```

Les classes

- Exemple 2:

```
class Point {
// …Autres déclaration
double distance_point(Point u){
      Point t;
      t.X = u.X – X;
      t.Y = u.Y – Y;
      return sqrt(t.X*t.X + t.Y*t.Y);
};

void main(){
      Point p1;
      Point p2;
      p1.X = 12.; p1.Y = 3.;
      p2.X = 4.; p2.Y = 5.;
      p1.distance_point(p2);
}
```



Les classes

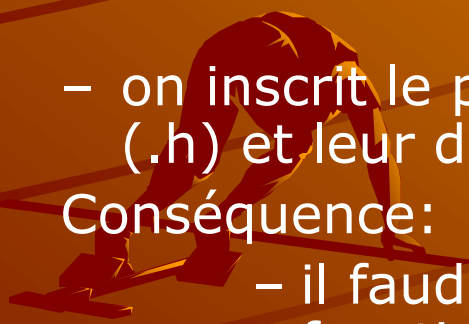
- On peut faire encore mieux:

Problème:

- en intégrant le code des fonctions membres dans les classes, on allonge les fichiers sources et on perd la vue d'ensemble (*principe de visibilité*)

Solution:

- on inscrit le prototype des fonctions membres dans la classe et définition à l'extérieur de la classe.



- on inscrit le prototype des fonctions membres dans un fichier (.h) et leur définition dans un autre fichier source (.cpp)

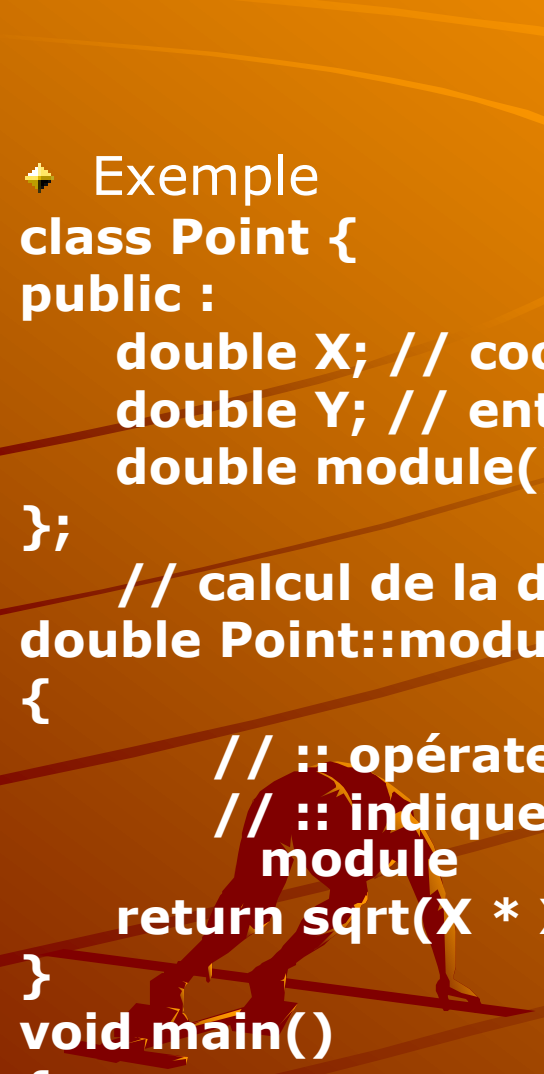
Conséquence:

- il faudra indiquer au compilateur l'appartenance d'une fonction membre à sa classe d'origine car il pourrait y avoir plus d'une fonction membre avec le même nom, tout en appartenant à des classes différentes.

Les classes

- Exemple

```
class Point {
public :
    double X; // coordonnées x et y
    double Y; // entrée 2
    double module();
};
    // calcul de la distance entre un point et l'origine
double Point::module ()
{
        // :: opérateur d'évaluation de portée
        // :: indique à quelle classe se rattache la fonction membre
           module
    return sqrt(X * X + Y * Y);
}
void main()
{
    Point x1;
    x1.X = 2; x1.Y = 3;
    cout << x1.module() << endl;
}
```



Les constructeurs

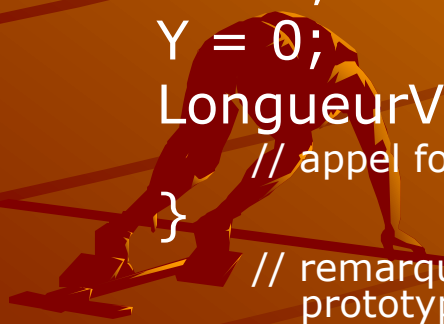
- Les constructeurs se sont des fonctions membres spéciales appelées à la création d'un objet d'une classe
  - Ce sont des fonctions dont le nom est celui de la classe
  - Qui ne retournent rien même pas un void
  - Qui peuvent contenir ou pas des arguments
  - Qui permettent d'initialiser les champs de la classe
    - Attention aux pointeurs, aux tableaux et tout objet qui doit s'allouer dynamiquement de la mémoire, il faut toujours les initialiser dans les constructeurs car sinon
      - cas des pointeur : pas d'initialisation d'un pointeur, le pointeur pointera vers une adresse au hasard, comme une zone de code quelconque, ce qui serait dramatique pour l'application.

- le *constructeur par défaut (constructeur sans arguments)* est appelé automatiquement à la création d'un objet (si pas de constructeur avec arguments)
- Remarque pour les variables statiques: ils sont initialisé par zéro.

Les constructeurs

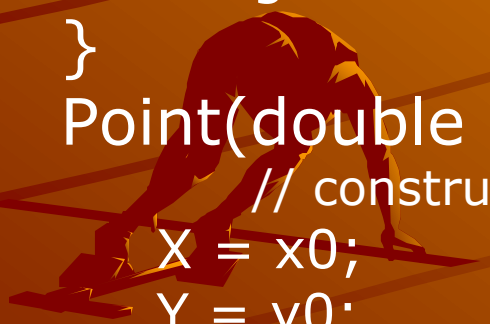
- exemple1

```
class Point {
public :
    double X; // coordonnées x et y
    double Y;
    double LongueurVecteurAssocie;
    Point(){
        // constructeur par défaut
    X = 0;
    Y = 0;
    LongueurVecteurAssocie = module();
        // appel fonction membre pour initialisé un champs
    }
        // remarquez que le constructeur appelle une fonction membre dont le
           prototype est fourni plus bas
    double module();
};
```



Surcharge du constructeur par défaut

```
class Point {
public :
        double X; // coordonnées x et y
        double Y;
        double LongueurVecteurAssocie;
    Point(){ // constructeur par défaut
        X = 0;
        Y = 0;
        LongueurVecteurAssocie = module();
    }
    Point(double x0, double y0) {
            // constructeur avec arguments
        X = x0;
        Y = y0;
        LongueurVecteurAssocie = module();
    }
    double module();
};
```



Constructeur avec valeurs par defaut

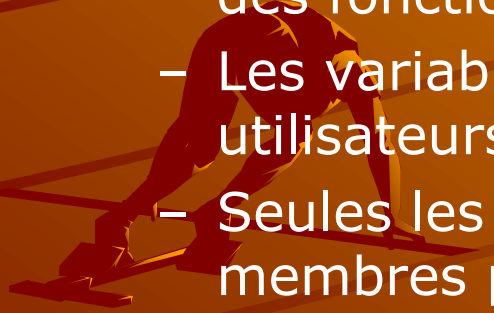
- Définition du constructeur avec valeurs par défaut

```
Point(double x1 = 0, double y1 = 0){
        X = x1;
        Y = y1;
        LongueurVecteurAssocie = module();
}
```

- Avec cette déclaration, on ne peut ajouter dans la classe un constructeur par défaut.
  - Car les valeurs par défaut des champs de la classe sont données par ce constructeur

Les accès

- L'encapsulation: utilisation des modes d’accès
- But:
  - créer une interface avec la classe qui permettra de protéger certains membres internes que l'utilisateur n'a pas à connaître et dont on veut éviter une utilisation hasardeuse
- L'interface se fait par:
  - des fonctions membres *publiques.*
  - Les variables membres privées sont cachées aux utilisateurs externe à la classe.
  - Seules les fonctions membres d'une classe ont accès aux membres privés de cette classe.



Fichier Point.h

```
class Point {
private :
    double X, Y; // coordonnées x et y
    double LongueurVecteurAssocie;
public :
    double getX() {return X;} // les accesseurs
    double getY() {return Y;}
    void setX(double a){X=a;}
    void setY(double a){Y=a} // les accesseurs
Point(double = 0, double = 0);
double module();
double distance_point(Point u);
};
```

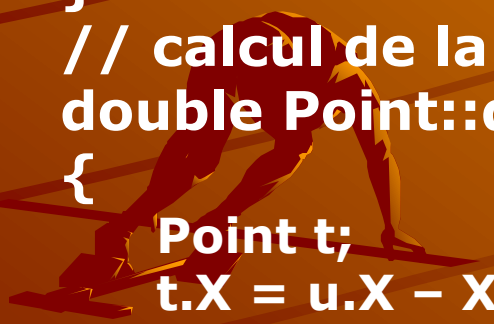
Les accesseurs sont utiles pour accéder aux champs privés hors de la classe Point.

Point.h est l'interface: elle contient les champs de la classe et les prototypes des méthodes

Fichier Point.cpp

Dans le fichier Point.cpp, on a les définitions des fonctions

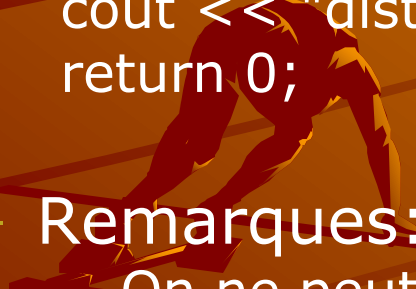
```
// constructeur
Point::Point(double x1 , double y1 ) {
    X = x1; Y = y1;
    LongueurVecteurAssocie = module();
}
// calcul de la distance entre un point et l'origine
double Point::module () {
    return sqrt(X * X + Y * Y);
}
// calcul de la distance entre deux points
double Point::distance_point(Point u)
{
    Point t;
    t.X = u.X – X; // pas besoin des accesseurs get…
    t.Y = u.Y – Y;
    return sqrt(t.X*t.X + t.Y*t.Y);
}
```



Programme principal

Dans le fichier test: main.cpp

```
#include <iostream.h>
#include " Point.h"
int main(){
    Point pp; //création de l'objet pp avec valeur par defaut
    Point p(5.4, -2.3); // création d'un objet, on utilisant un
                        constructeur avec argument!
    cout << "Point pp: " << p.getX() << endl;
    cout << "Point: " << p.getX() << " , " << p.getY() << endl;
    cout << "distance à l'origine" << p.module() << endl;
    return 0;
}
```



- Remarques:
  - On ne peut faire
    - cout << "Point: " << p.X << " , " << p.Y << endl;
    - Car X et Y sont des champs privés, donc pas d'accès directe
  - Une variable membre privée ne peut être accessible via l'opérateur . (le programme ne compilera pas) sauf bien sûr dans les fonctions membres de la classe où l'opérateur . n'est même pas nécessaire.

Tableau d'objets

- Dans le cas d'un tableau d'objet, le constructeur par défaut est appelé à la création de chaque élément du tableau.

```
Rangee tabPoint[3];
void main(){
    Point *tabPoint = new Point[3];
    for(int i = 0; i < 3; i++){
        tabPoint[i].X =4 + i;
        tabPoint[i].Y =8 * i;
    }
    Cout << " coordonnée du point à la deuxième
      position: "<< tabPoint[1].X << " " << tabPoint[1].Y
      <<endl;
    delete[] tabPoint;
    }
```

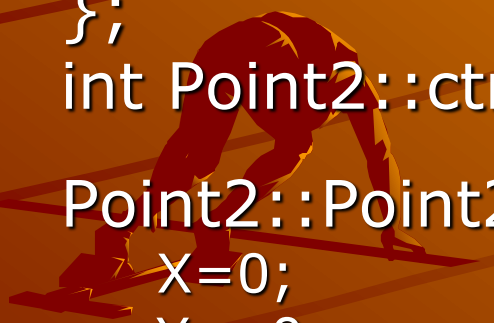
Le destructeur

- **Un destructeur** est une fonction membre qui est automatiquement appelée au moment de la destruction d'un objet, avant la libération de l'espace mémoire associée à l'objet.
  - La destruction des objets se fait de manière identique aux variables ordinaires :
    - à la fin du programme pour les objets statiques,
    - à l'aide de l'instruction delete ou à la fin du bloc ou de la fonction pour les objets dynamiques.
  - La libération de la mémoire allouer dynamiquement étant confié dans ce cas au destructeur.
  - le destructeur est indispensable dès qu'un objet ou champs est amené à allouer dynamiquement de la mémoire.
- Par convention:
  - le destructeur porte le même nom que sa classe, précède du symbole du tilde (~).
- Par définition
  - il ne comporte aucun argument ne renvoi pas de valeur (même pas un void).

- Exemple : ~Point() ;// destructeur de la classe Point

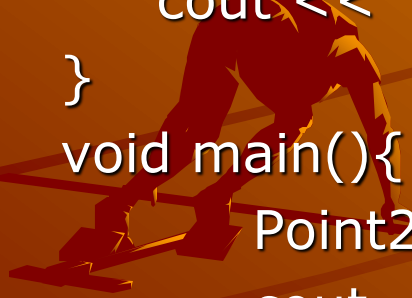
Exemple: utilisation de destructeur

```
#include <iostream.h>
class Point2{
private:
    static int ctr; // compteur du nombre d'objet créés.
    int X, Y;
public :
    Point2() ; // constructeur par défaut
    ~Point2() ;  // destructeur
};
int Point2::ctr=0; // initialisation de la variable statique à
                   l'extérieur du constructeur
Point2::Point2(){
    X=0;
    Y = 0;
    cout << " ++ construction : il y a maintenant " << ++ctr <<
       " objet(s)" << endl ;
}
```



Exemple: utilisation de destructeur

```
Point2::~Point2(){
    cout << ” -- destructeur : il y a maintenant ” << --ctr <<
       ” objet(s)” << endl ;
    cout << ” adresse de l’objet détruit est: ”<<this << endl;
}
// fonction qui crée deux objets (création statique )
void fct(){
    Point2  u, v ;
    cout << ” sortie de la fonction” << endl ;
}
void main(){
        Point2 U ;   // le constructeur crée 1 objet
        cout << ” appel de la fonction création ” << endl ;
        fct() ;          // la fonction crée deux objets
        Point2  V     // le constructeur crée 1 objet
}
```

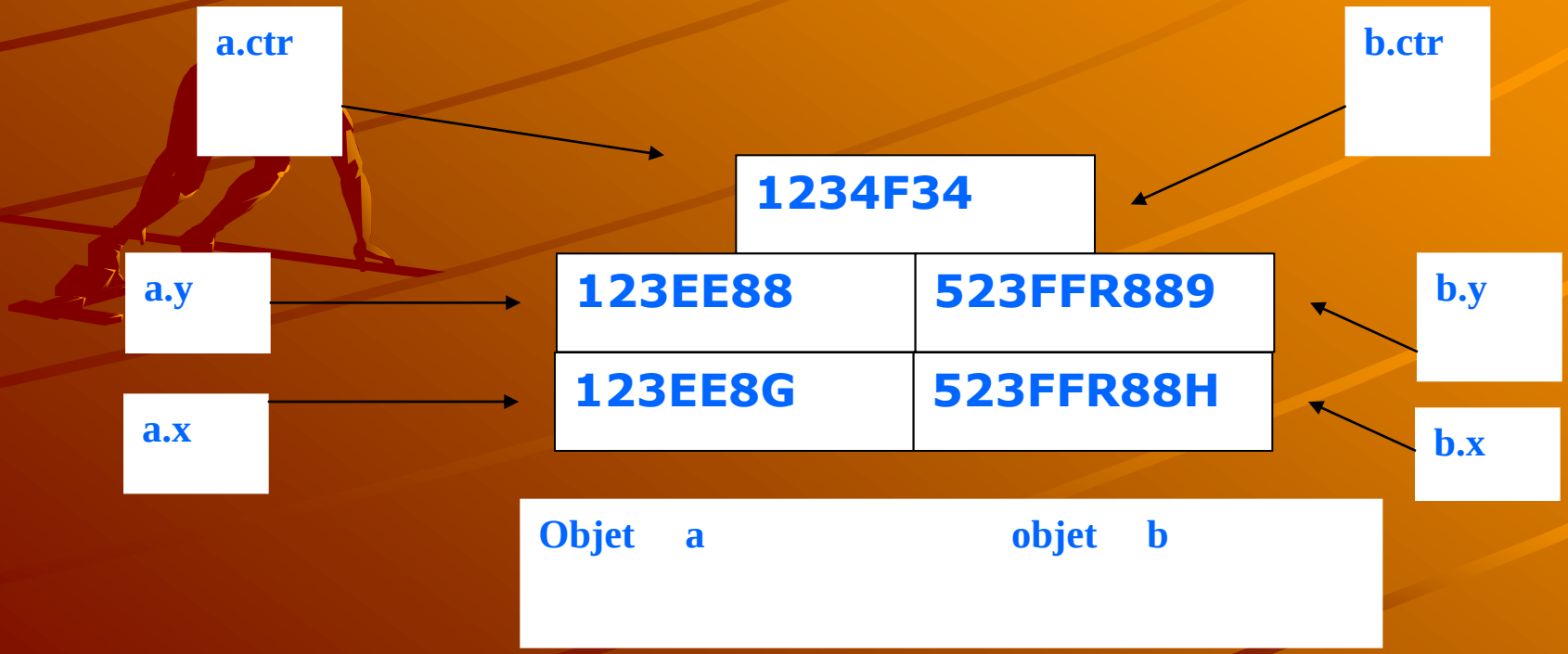


Variable static

- **Lorsqu'un membre donné est déclaré static,**
  - **il est partagé pour tous les objets de la même classe (1 seul emplacement en mémoire pour ce membre donnée pour tous les objets).**
  - **Il est initialisé, par défaut, à zéro (comme toutes les variables statiques).**
- **Exemple**

```
class point{
static double x ;
double y;
} ;
```

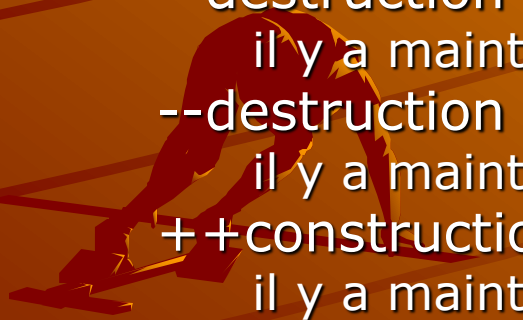
la déclaration : Point a, b ; conduit à la situation suivante :



Exemple: utilisation de destructeur

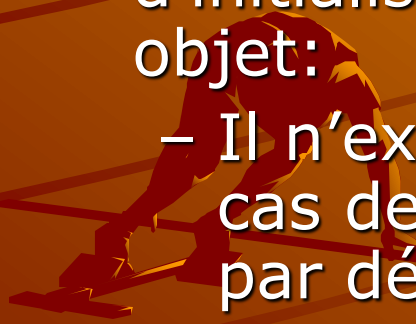
- donne comme affichage :

```
++ construction :
     il y a maintenant 1 objet(s)  // création de l'objet U
appel de la fonction fct()
++construction :
     il y a maintenant 2 objets(s)  // création de l'objet u
++construction :
     il y a maintenant 3 objets(s)  // création de l'objet v
sortie de la fonction fct()
--destruction :
     il y a maintenant 2 objets(s)  // destruction des objets u et v
--destruction :
     il y a maintenant 1 objets(s)  // en sortant de la fonction
++construction :
     il y a maintenant 2 objets(s)  // création de l'objet V
--destruction :
     il y a maintenant 1 objets(s)  // destruction des objets U et V
--destruction :
     il y a maintenant 0 objets(s)  // en sortant du programme.
```



Constructeur par recopie

- Reprenons notre classe point des chapitres précédents.
  - Soit la déclaration d'un objet a : point a(3,5) ;
  - Il est possible de déclarer un nouvel objet b et de l'initialiser aux valeurs de l'objet a par
    - **point B=A ; //initialisation**
- Il y a deux manières différentes qui permettent d'initialiser un objet via les valeurs d'un autre objet:
  - Il n'existe pas de constructeur prévoyant ce cas de déclaration: dans ce cas un traitement par défaut est prévu.
  - Il existe un constructeur prévoyant ce cas de déclaration et il sera alors utilisé.



Constructeur par recopie

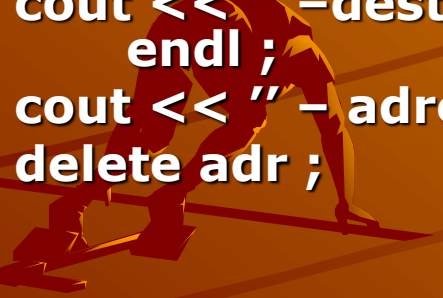
- S'il n'existe pas de constructeur approprié.
  - Cela signifie qu'il n'existe pas, dans la classe point, de constructeur à un seul argument de type point:
    - le compilateur va donc mettre en place une recopie des valeurs de l'objet **A** aux valeurs de l'objet **B,** après création de celui ci (comme l'affectation entre variable de même type).
  - Un problème se pose lorsque les objets contiennent des pointeurs sur des emplacements alloués dynamiquement :
    - les valeurs des pointeurs seront recopiées mais pas les emplacements pointés (c.a.d: les pointeurs pointent vers la même adresse sachant qu'ils ont eux des adresses différents).
  - Dans ce cas, C++ utilise un **constructeur par recopie par défaut** effectuant la simple recopie des membres données.
  - **Remarque**
    - Même s'il existe un constructeur usuel pour la classe, c'est le constructeur par recopie par défaut qui sera utilisé.

Constructeur par recopie

- S'il existe un constructeur approprié
  - Le C++ imposant à ce constructeur(constructeur par recopie):
    - Que son unique argument soit transmis par référence.
    - Il sera donc de la forme: **point(point &).**
  - Ce constructeur sera appelé de façon habituelle (après création de l'objet):
    - mais cette fois ci, aucune recopie n'est mise en place automatiquement :
    - c'est au constructeur de recopie de la prendre en charge.

Exemple: Constructeur par recopie par défaut

```
class tab{
        int nbelem ;     // nombre d'élément dans le tableau
        int  *adr ;
public :
tab(int n) {
        nbelem = n ;
        adr = new int[nbelem] ;    // allocation dynamique du tableau
        cout << " ++ constructeur usuel – adresse de l'objet crée = "
             << this << endl ;
        cout << " –adresse tableau= " << adr << endl ;
}
~tab( ){
    cout << " –destructeur  – adresse objet détruit = " << this <<
         endl ;
    cout << " – adresse tableau = " << adr << endl ;
    delete adr ;
}
} ;
void main( ){
    tab t1(5) ;     // création de t1 par le constructeur usuel
    tab t2 = t1 ;  // création de t2 par appel du constructeur de
         recopie                    par défaut sur l'objet t1
    // ou bien, tab t2(t1) ;
}
```



Exemple: Constructeur par recopie par défaut

- Donne comme affichage :

++constructeur – adresse objet = 0x8F87FFF2 – adresse tableau = 0x8F870D98 //t1

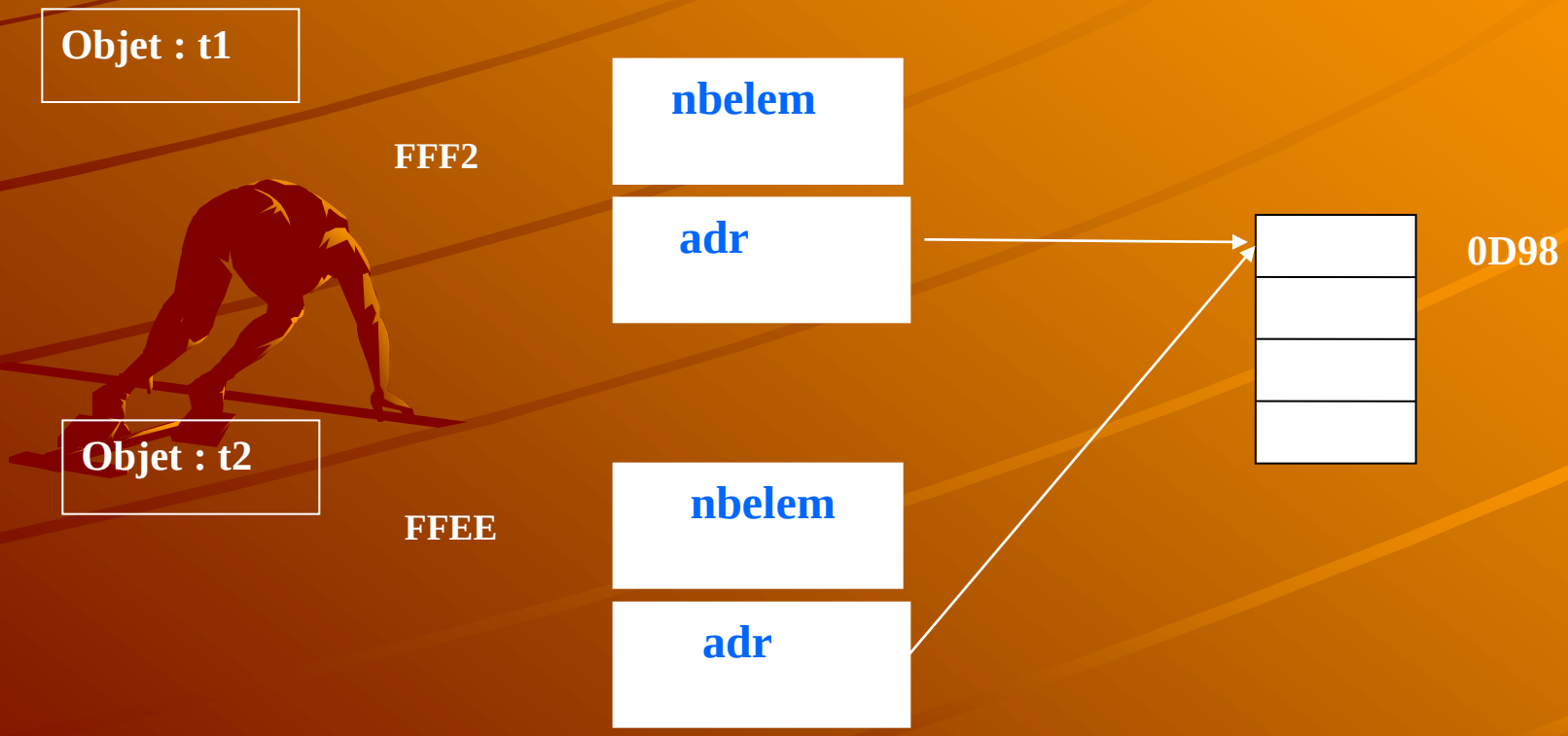
-- destructeur – adresse objet = 0x8F87FFEE – adresse tableau = 0x8F870D98 //dest. t2

-- destructeur – adresse objet = = 0x8F87FFF2 – adresse tableau = 0x8F870D98 //dest. t1

- D'après les résultats de l'exécution du programme, on constate que l'objet t2 a bien été créé, avec recopie des valeurs de l'objet t1.
- A la fin du programme principal, l'appel du destructeur libère l'emplacement pointé par adr pour l'objet t1, puis libère le même emplacement pour l'objet t2 : le même emplacement mémoire est donc libéré deux fois, ce » qui est complètement anormal !!!
- La situation peut se schématiser de la façon suivante :

Exemple: Constructeur par recopie par défaut

- Cette situation montre également
  - toute modification du tableau de l'objet t1 entraînera donc une modification du tableau de l'objet t2 et inversement.
- Pour éviter ce problème, il faut créer un constructeur par recopie.



Constructeur par recopie

- **Définition d'un constructeur par recopie**
  - Le constructeur par recopie devra:
    - créer un nouvel objet avec recopie des membres données, mais surtout avec un nouvel emplacement pour le tableau.
  - La situation deviendra alors la suivante :

Constructeur par recopie

Objet : t1

FFF2

nbelem

adr

0D98

Objet : t2

FFEE

nbelem

adr

0DA6

Exemple: Constructeur par recopie

- On ajoute à l'exemple précèdent la fonction suivante (constructeur par recopie)

```
tab(tab& v)    // constructeur par recopie
{
    nbelem = v.nbelem ;
    adr=new int[nbelem];    // création du nouveau tableau
    for(int i=0 ; nbelem ; i++)  adr[i]=v.adr[i];   // recopie de
      l'encien tableau
    cout << " ++ constructeur par recopie – adresse objet crée
      = " << this << endl ;
    cout << " – adresse tableau= " << adr << endl ;
}
```

- Donne comme affichage :

```
++constructeur – adresse objet = 0x8F87FFF2 – adresse
  tableau = 0x8F870D98  //t1
++constructeur – adresse objet = 0x8F87FFEE – adresse
  tableau = 0x8F870DA6  //t2
--destructeur – adresse objet = 0x8F87FFEE – adresse
  tableau = 0x8F870DA6  //dest.t2
-- destructeur – adresse objet = = 0x8F87FFF2 – adresse
  tableau = 0x8F870D98  //dest. t1
```

Constructeur par recopie

- Remarque
  - Le constructeur par recopie permet ici de régler le problème de l'initialisation d'un objet par un autre objet de même type au moment de sa déclaration.
  - Cela ne résout pas pour autant le problème qui se pose en cas d'affectation entre des objets de même type (exp : t1=t2), où il faut, dans ce cas, surdéfinir l'opérateur = (ce type de surdéfinition sera vu ultérieurement)

Liste d'Initialisation d'un constructeur

- **Soit la classe :**

```
class Y { /* ... */
};
class X {
public:
    X(int a, int b, Y y);
    ~X();    // ....
private:
    const int _x;
    Y _y;
    int _z;
};
X::X(int a, int b, Y y) {
_x = a; // ERREUR: l'affectation à une constante est interdite
_z = b; // OK : affectation};
```

- Questions:
  - **Comment initialiser la donnée membre constante _x**
  - **Comment appeler le constructeur de la classe Y afin d'initialiser l'objet membre _y**
- **Réponse :** la liste d'initialisation.

Liste d'Initialisation d'un constructeur

- La phase d'initialisation d'un objet utilise:
  - une liste d'initialisation qui est spécifiée dans la définition du constructeur.
- Syntaxe :
  - nom_classe::nom_constructeur(args ...) : liste_d_initialisation{ // corps du constructeur}
- Exemple:

```
X::X(int a, int b, Y y) : _x( a ) , _y( y ) , _z( b ) {
// rien d'autre à faire d'autre pour l'initialisation
}
```

  - L'expression *_x(a)*
    - indique au compilateur d'initialiser la donnée membre *_x* avec la valeur du paramètre *a*.
  - L'expression *_y(y)*
    - indique au compilateur d'initialiser la donnée membre *_y* par un appel au constructeur (avec l'argument *y*) de la classe *Y*.
      - Avec y est un objet de la classe Y

      Ou
      - y est un ensemble de valeur des champs de la classe Y

Exemple

```
class Personne{
private:
      char *nom
      double age;
public:
      Personne();
      Personne(char *, double) ;
      void affiche() ;
};
Personne::Personne(){
      cout << "constructeur par défaut: " << endl;
}
Personne::Personne(char *nomPers, double agePers){
      cout << "constructeur avec argument: " << endl;
      Nom = new char[strlen(nomPers)+1].
      strcpy(nom, nomPers) ;
      age = agePers;
      affiche();
}
void Personne::affiche(){
      cout << "nom de la personne est: " << nom << "  " << "son age est: " << age
         << endl;
}
```

Suite exemple

```
class Biblio{
public:
    Biblio(char *, double) ;
    Biblio(Personne) ;
    ~Biblio() ;
private:
    Personne *p1 ;
    Personne p2 ;
    static Personne p3; // variable de classe
                        (partagé par tous les objets)
};
```

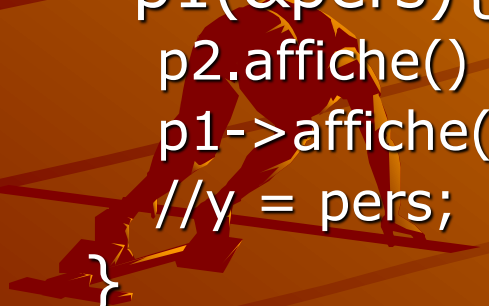
- Question: Comment initialiser p1, p2 et p3 **?**
  - variable de classe: p3

```
Personne Biblio :: p3(”Albertine”,20) ;
    // on ne répète pas le mot clé static et
```

Suite exemple

- variables d'instance P1 et P2:

```
Biblio :: Biblio(char *nomPerso, double d ) :
 p1(new Personne(nomPerso, d)), p2(nomPerso, d){
// p1 est un pointeur, donc on utilise l'opérateur new
//y=(nomPerso, d);
// pas de new : p2 est un l'objet
}
Biblio :: Biblio(Personne pers ) : p2(pers),
   p1(&pers){
    p2.affiche() ;
    p1->affiche() ;
    //y = pers;
 }
Biblio :: ~Biblio( )  // destructeur{
    delete p1 ;  // p1 est détruit automatiquement
}
```



Liste d'initialisation dans un constructeur

- Pour initialiser un objet, on peut utiliser la liste d'initialisation donnée dans le constructeur.
- La liste d'initialisation est utile quand:
  - on est amené à initialiser un attribut d'une classe dont le type est un type d'une autre classe.
  - On a un attribut constant.
  - On utilise l'héritage entre classes:
    - afin de passer des arguments aux constructeurs de classes de base via le constructeur de la classe dérivée.