Programmation en C et Méthodes Numériques

Rappel de C

T. Dietenbeck (thomas.dietenbeck@upmc.fr)

Sorbonne Université

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Pour vous rassurer

[Alan Perlis]

- "Programmer est un acte contre nature."
- "Dans un ordinateur, le langage naturel n'est pas naturel."
- "Il y aura toujours des choses que nous aimerions dire dans nos programmes, mais qui ne peuvent être que mal dites avec tous les langages connus."

- "Vous croyez savoir quand vous apprenez, vous en êtes sûr quand vous écrivez, persuadé quand vous enseignez, mais certain seulement quand vous programmez."

SCIENCES SORBONNE UNIVERSITÉ

Algorithmique et Programmation

Résolution d'un problème

1. Analyse du problème ($\approx 40\%$)
2. **Conception d'une solution** : Algorithmique ($\approx 40\%$)
   - choix de la représentation des données
   - choix de la méthode utilisée
3. **Développement** : Programmation ($\approx 20\%$)
   - choix du langage de programmation
   - choix de la machine utilisée
4. Tests : Selon le résultat, on retournera en 2 ou 3.

[Alan Perlis]

- "Une fois que vous comprenez comment écrire un programme, trouvez quelqu'un d'autre pour l'écrire."
- "Enseigner la programmation va à l'encontre de l'éducation moderne : Quel est le plaisir à planifier, se discipliner à organiser ses pensées, faire attention aux détails et apprendre à être autocritique ?"

Les erreurs

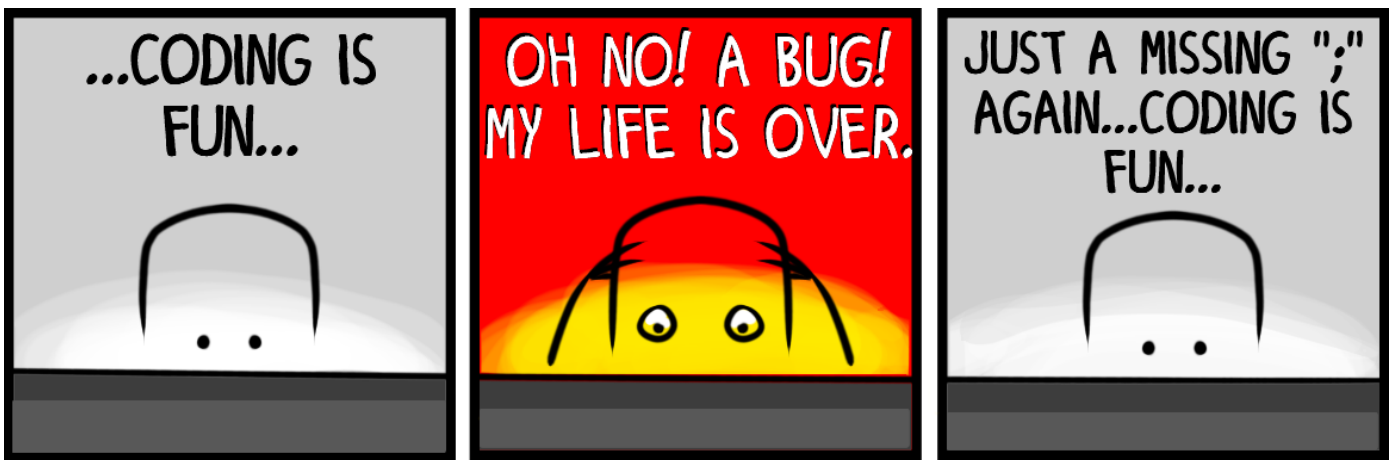
Définition

Il existe 2 types d'erreur

- les erreurs syntaxiques : le progamme ne compile pas
- les erreurs à l'exécution : le programme ne s'exécute pas correctement

[Alan Perlis]

"Il y a deux manières d'écrire des programmes sans erreurs ; seule la troisième marche."



SCIENCES SORBONNE UNIVERSITÉ

Erreurs syntaxiques

Définition

- Erreurs qui surviennent pendant la compilation du programme :
- Non respect des règles de syntaxe (*e.g.* oubli d'un ";", variable non déclarée, ...)
- Faciles à corriger : le compilateur les détecte et les signale dans le compte rendu de compilation

Exemple

```
1  #include <stdio.h>
   int main( ) {
3    double x = 1;
     z = x + 1;
5    printf("z = %f\n", z)
     return 0;
7  }
```

```
ErreursSyntaxiques.c:4: erreur: 'z' undeclared (first use in this
                                function)
ErreursSyntaxiques.c:6: erreur: expected ';' before 'return'
```

SCIENCES SORBONNE UNIVERSITÉ

Erreurs d'exécution

Définition

- Erreurs qui surviennent pendant l'exécution du programme :
  - soit le programme se bloque et affiche un message d'erreur
  - soit le programme produit un résultat faux
- Plus difficiles à corriger : le compilateur ne les détecte pas. Il faut exécuter le programme pas à pas pour comprendre où survient l'erreur
  - soit à l'aide d'affichage (pour savoir où l'on est, le contenu de variables, *etc.*)
  - soit à l'aide du debugger (inclus dans la plupart des logiciels de développement)

Erreurs classiques

- Provoquant l'arrêt du programme
  - Variable non créée (*e.g.* pointeur, tableau)
  - Accès à une case hors du tableau
- Produisant un mauvais résultat
  - Variable non initialisée
  - Oubli des accolades
  - Condition toujours vraie / fausse $\Rightarrow$ pas de passage dans la boucle

SCIENCES SORBONNE UNIVERSITÉ

Commentaires

3 types de commentaires

```
// Commentaire sur une ligne

/* Exemple de commentaires sur
 * plusieurs lignes
 */

/** Documentation de methode
  * Permet de decrire brievement l'utilite de la methode
  */
```

Intérêt

- Permet d'expliquer
  - le rôle d'une variable
  - le fonctionnement d'une méthode
  - l'objectif d'une série d'instructions
- Ne sont pas compilés $\Rightarrow$ n'allourdit pas l'exécutable (n'ayez donc pas peur d'en écrire)
- Une page web ou un fichier pdf de documentation peut être créé automatiquement à partir des commentaires `/** ... */` .

Commentaires

3 types de commentaires

```
// Commentaire sur une ligne

/* Exemple de commentaires sur
 * plusieurs lignes
 */

/** Documentation de methode
  * Permet de decrire brievement l'utilite de la methode
  */
```

[Alan Perlis]

"La documentation est comme une assurance-vie : le bénéficiaire n'est presque jamais celui qui l'a signée."

Guide de Syntaxe

Ce qu'il ne faut surtout pas faire

```
      int main( ){ int pingouin=41,
canard=1337,vache;vache=
(pingouin++ > 40)?142857:canard-pingouin;
printf("%d, %s", pingouin
    ,loutre(vache, canard, pingouin));
      return 0;}
char[] loutre(int c,int b,
int a){if(c<a){printf("%d",b);}
else{printf("%d",a);}return "Facile";}
```

[Martin Golding]

"Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live."

SCIENCES SORBONNE UNIVERSITÉ

Plan

Rappels

Qu'est ce qu'un programme ?

- des variables : pour stocker les résultats, des valeurs intermédiaires, *etc.*
- des instructions : les opérations (somme, lecture, *etc.*) à faire pour parvenir au résultat souhaité.

Définitions

- Variable : zone mémoire définie par
  - Nom : identificateur évocateur
  - Type : entier, caractère, booléen, ...
  - Valeur : son contenu
- Type
  - comparable aux unités de mesure en physique
  - domaine de valeurs, propriétés, opérateurs associés

$\Rightarrow$ On ne mélange pas les types

SCIENCES SORBONNE UNIVERSITÉ

Les types primitifs

Définition

- Types de base (dont la taille exacte est connue) qui permettent de manipuler
  - des nombres entiers,
  - des nombres flottants (réels),
  - des caractères
- En C, il n'existe pas de type booléen

Type	Contient	Taille	Domaine
`char`	caractère	16 bits	
`int`	entier signé	32 bits	$[-2.10^9; 2.10^9]$
`float`	réel	32 bits	$[-3,4.10^{38}; -3,4.10^{-38}]$ et $[3,4.10^{-38}; 3,4.10^{38}]$
`double`	réel	64 bits	$[-1,7.10^{308}; -1,7.10^{-308}]$ et $[1,7.10^{-308}; 1,7.10^{308}]$

SCIENCES SORBONNE UNIVERSITÉ

Conversion de type (cast)

Définitions

- Conversion possible entre entiers, réels et caractères
- Deux types de conversion :
  - **Elargissante** : conversion d'un type vers un type plus large (implicite)
  - **Restrictive** : conversion d'un type vers un type plus petit (doit être explicite)

Exemple de conversions élargissantes

```
int i = 42;
float f = i;  // float > int
f = 13.37;
double d = f;  // double > float
```

SCIENCES SORBONNE UNIVERSITÉ

Conversion de type (cast)

Définitions

- Conversion possible entre entiers, réels et caractères
- Deux types de conversion :
  - **Elargissante** : conversion d'un type vers un type plus large (implicite)
  - **Restrictive** : conversion d'un type vers un type plus petit (doit être explicite)

Exemple de conversions restrictives

```
double d = 13.37;
float f = d;           // Erreur de compilation float < double
float f = (float)d;    // On force la valeur de float a etre
                       // convertie en une valeur float
// Attention a la perte de precision lors d'une conversion!
i = (int) 42.745;        // i = 42 (Valeur tronquee)
```

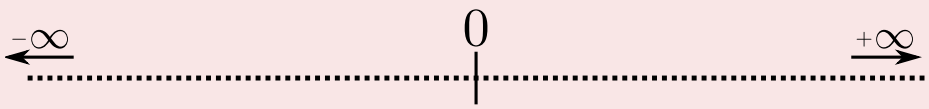
SCIENCES SORBONNE UNIVERSITÉ

Quelques dangers

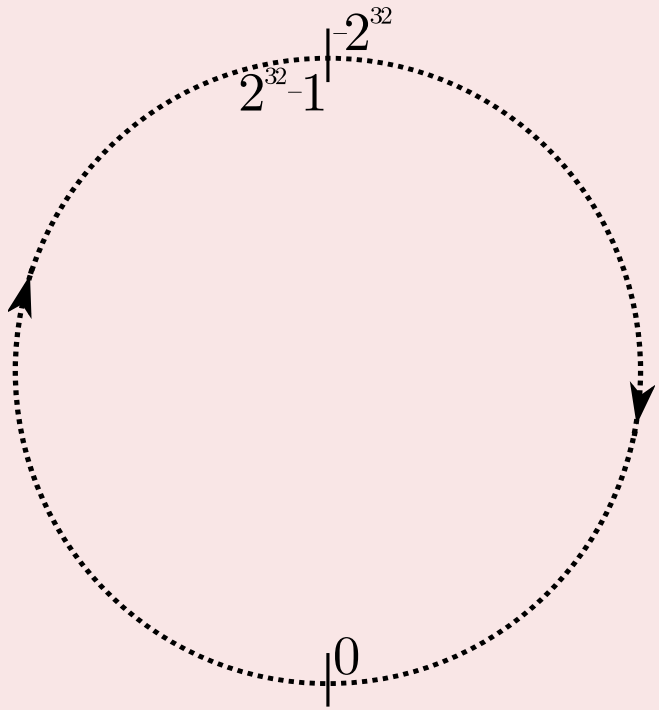
Overflow

En C (et dans d'autres langages), l'arithmétique entière est "circulaire" !

```
int i1 = 2147483647; // plus grande valeur du type int
int i2 = 1;
int sum = i1 + i2; // sum prend la valeur -2147483648 !
```



En mathématiques



En informatique

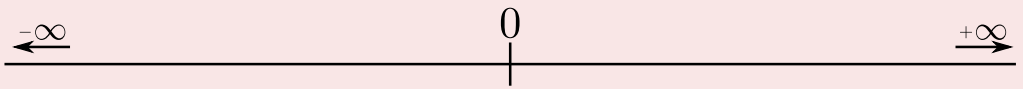
SCIENCES SORBONNE UNIVERSITÉ

Quelques dangers

La précision des réels

```
float f1 = 1e16;
float f2 = 1e16;
float res;

res = f1 - f2 + 1;    // res = 1;
res = f1 + 1 - f2;    // res = 0;
```



En mathématiques

En informatique

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Opérateurs d'affectation et arithmétiques

Opérateur d'affectation

Affectation simple	`a = 0`

Opérateurs arithmétiques

Addition	`a + b`
Soustraction	`a - b`
Multiplication	`a * b`
Division	`a / b`
Modulo	`a % b`

- Le modulo (reste de la division entière) ne fonctionne qu'avec des entiers

Exemples

```
int unEntier = 8 + 3; // unEntier = 11
unEntier = unEntier % 7; // unEntier = 4

double unReel = 3.;    // unReel = 3
unReel = unReel * unEntier;  // unReel = 12
unReel = unReel / 5;  // unReel = 2.4
```

SCIENCES SORBONNE UNIVERSITÉ

Opérateurs d'affectation et arithmétiques

Affectations élargies

Addition	`a += b` ⇒ `a = a + b`
Soustraction	`a -= b` ⇒ `a = a - b`
Multiplication	`a *= b` ⇒ `a = a * b`
Division	`a /= b` ⇒ `a = a / b`
Modulo	`a %= b` ⇒ `a = a % b`

- Ces notations permettent d'alléger / de condenser le code.
- Le modulo (reste de la division entière) ne fonctionne qu'avec des entiers

Exemples

```
int unEntier = 8 + 3; // unEntier = 11
unEntier %= 7;        // unEntier = 4

double unReel = 3.;   // unReel = 3
unReel *= unEntier;   // unReel = 12
unReel /= 5;          // unReel = 2.4
```

SCIENCES SORBONNE UNIVERSITÉ

Quelques dangers

La division

Attention aux types des opérandes lors d'une division

```
int num1 = 8, den1 = 3;
int qI = num1 / den1; // qI = 2

double num2 = 8., den2 = 3.;
double qD = num2 / den2;      // qD = 2.666...
qD = num1 / den1;             // qD =
qD = (double)(num1 / den1); // qD =
qD = (double)(num1) / den1; // qD =
qD = num2 / den1;             // qD =
qD = num1 / 3.;               // qD =
```

Quelques dangers

La puissance

L'opérateur ^ n'est pas le symbole pour la puissance !
Il faut

- soit faire des produits successifs (*e.g.* `x*x*x` pour x^3)
- soit utiliser la fonction `double pow( double x, double y)` de la bibliothèque `math.h`.

```
#include <math.h>
int main( ) {
  int n = 2;
  double x = 8.;
  double xN = pow( x, n );  // xN = 64
}
```

SCIENCES SORBONNE UNIVERSITÉ

Opérateurs relationnels et logiques

Opérateurs relationnels

Inférieur	`a < b` et `a <= b`
Supérieur	`a > b` et `a >= b`
Egalité	`a == b`
Inégalité	`a != b`

Opérateurs logiques

ET logique	`a && b`
OU logique	`a || b`

Exemples

```
int val1 = 1;
int val2 = 2;
if((val1 == 1) && (val2 == 2))
  printf("val1 vaut 1 ET val2 vaut 2 \n");
if((val1 == 1) || (val2 == 1))
  printf("val1 vaut 1 OU val2 vaut 2 \n");
```

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Structures de contrôle

Principe

- En règle générale, les instructions sont exécutées séquentiellement (dans le sens de lecture)
- Les structures de contrôle permettent de modifier cette ordre de lecture
  - soit en choisissant une suite d'instructions selon le résultat d'un test

    si $a > 2$ **alors**
    | $a \leftarrow a + 2$
    sinon
    | $a \leftarrow a * 2$
  - soit en répétant une suite d'instructions

    tant que $a > 2$ **faire**
    ⌊ $a \leftarrow a - 2$

[Alan Perlis]

"Un programme sans boucle et sans structure de donnée ne vaut pas la peine d'être écrit."

SCIENCES SORBONNE UNIVERSITÉ

Sélections : `if/else`

L'instruction `if`

Exécution d'une instruction ou d'un bloc d'instructions si une condition est remplie.

```
if( condition ) {
  /* instruction(s) a executer si la condition est vraie
   */
}
```

Remarque : si le test est faux, on ne fait rien.

SCIENCES SORBONNE UNIVERSITÉ

Sélections : `if/else`

La clause `else`

Définition d'instructions à exécuter si la condition n'est pas remplie.

```
if( condition ) {
  /* instruction(s) a executer si la condition est vraie
   */
} else {
  /* instruction(s) a executer si la condition est fausse
   */
}
```

SCIENCES SORBONNE UNIVERSITÉ

Sélections : if/else

if ... else en cascade

Lorsque plusieurs `if` se suivent, chaque `else` se rattache au `if` le plus proche

```
if( condition1 ) {
  /* instruction(s) a executer si la condition 1 est vraie
   */
} else if( condition 2) {
  /* instruction(s) a executer si la condition 1 est fausse
   * et la condition 2 vraie
   */
} else {
  /* instruction(s) a executer si les conditions 1 et 2
   * sont fausses
   */
}
```

SCIENCES SORBONNE UNIVERSITÉ

Sélections : if/else

Exemple

```
if( unEntier % 2 == 0 ) {
  printf( "%d est pair.\n", unEntier );
} else {
  printf( "%d est impair.\n", unEntier );
}

if( unReel > 0  ) {
  printf( "%f est positif.\n", unReel );
} else if( unReel < 0  ) {  // Si on arrive ici, on sait que
    ↪ unReel est negatif ou nul
  printf( "%f est negatif.\n", unReel );
} else {
  printf( "%f est nul.\n", unReel );
}
```

SCIENCES SORBONNE UNIVERSITÉ

Les boucles

Définitions

- On écrit une seule fois une séquence d'instructions qui pourra être exécutée plusieurs fois
- Trois façons de répéter la séquence
  - **Nombre d'itérations non connu a priori** : boucle avec condition d'arrêt
    `while` : la condition est déterminable avant le traitement, l'instruction itérée peut ne pas être exécutée du tout
    `do ... while` : la condition est calculée par l'instruction itérée, l'itération sera exécutée au moins une fois
  - **Nombre d'itérations connu a priori** : boucle avec compteur
    `for`

Les boucles

La boucle `for`

- **Nombre d'itérations connu a priori**
- Boucle avec compteur : on indique
  - sa valeur de départ
  - la condition d'arrêt
  - la manière dont il est modifié à la fin de chaque itération

```
for( uneVar = debut; uneVar <= fin; incrementation ) {
  /* instruction(s) a repeter
   */
}
```

- `uneVar = debut` (initialisation) : effectuée avant d'entrer pour la 1ère fois dans la boucle
- `uneVar <= fin` (condition) : évaluée au début de chaque tour de boucle
- incrémentation : effectuée après l'exécution des intructions
- `uneVar` doit être déclarée avant la boucle (le programme doit connaître son type)

SCIENCES SORBONNE UNIVERSITÉ

Les boucles : Exemples simples

Parcours classique

```
int i;  // Declaration du compteur de boucle i
for( i = 1; i < 3; i++ ) {
  printf( "%d \n", i );
}
```

Remarque : l'instruction `i++` est équivalente à `i = i + 1`.

Parcours inverse

```
int i;
for( i = 3; i > 1; i-- ) {  // Decrementation
  printf( "%d \n", i );
}
```

Remarque : l'instruction `i--` est équivalente à `i = i - 1`.

SCIENCES SORBONNE UNIVERSITÉ

Les boucles : Exemples simples

Incrémentation quelconque

```
int i;
for( i = 0; i < 10; i += 2 ) {  // On incremente i de 2
  printf( "%d \n", i );  // Affiche les entiers pairs
}
```

Avec des réels

```
double x;
for( x = 13.37 ; x < 142.857 ; x += 4.2 ) {
  printf( "%f \n", x );
}
```

SCIENCES SORBONNE UNIVERSITÉ

Les boucles `while` et `do ... while`

Principe

On ne connaît pas toujours le nombre d'opérations nécessaires pour obtenir un résultat

- données saisies par un utilisateur (numéro de téléphone, *etc.*) avec risque d'erreur de saisie (pas que des chiffres, pas la bonne longueur, *etc.*)
- calcul itératif de la limite d'une suite mathématique, du zéro d'une fonction avec une borne sur l'erreur
- recherche d'une valeur particulière dans un ensemble de données

La condition de continuation ne porte plus (uniquement) sur un compteur

SCIENCES SORBONNE UNIVERSITÉ

La boucle `while`

Définition

La condition de continuation est évaluée **au début** de la boucle

- elle porte sur une ou plusieurs variables initialisées avant la boucle
- si la condition est fausse initialement, le programme n'exécute aucune instruction de la boucle
- si la condition de continuation ne devient jamais fausse, le programme ne sort jamais de la boucle (on parle de boucle infinie)
  $\Rightarrow$ dans la boucle, une ou plusieurs instructions agissent sur les variables de la condition de continuation pour la faire évoluer vers la condition d'arrêt

Syntaxe

```
while( condition ) {
  // instruction(s) a repeter
}
```

SCIENCES SORBONNE UNIVERSITÉ

La boucle `do ... while`

Définition

Similaire à la boucle `while`, **mais**

- la condition de continuation est évaluée **à la fin** de la boucle
- même si la condition est fausse initialement, le programme passe au moins une fois dans la boucle
- les variables de la condition peuvent être initialisées dans la boucle

Syntaxe

```
do {
  // instruction(s) a repeter
} while( condition );
```

Attention au “;” après la condition

SCIENCES SORBONNE UNIVERSITÉ

Les boucles : Exemples

Affichage de tous les entiers positifs inférieurs à 10

On connaît

- la valeur de départ (1)
- la valeur de fin (10)
- l'incrémentation (+1)

⇒ on peut utiliser une boucle `for`

```
int i;
for( i = 1; i <= 10; i++ ) {
  printf( "%d \n", i );
}
```

Les boucles : Exemples

Simulation de la division entière

On calculera le quotient et le reste de la division entière de a par b (entrés par l'utilisateur) sans utiliser les opérateurs / et %.

- On connaît les valeurs de départ ($q = 0$, $r = a$)
- On ne connaît pas le nombre d'itérations à faire (valeur du quotient)
- On peut ne pas passer dans la boucle (si $b > a$)

$\Rightarrow$ on peut utiliser une boucle `while`

```
int a, b;
scanf( "%d \n", &a );  // On demande a et b a l'utilisateur
scanf( "%d \n", &b );
int q = 0, r = a;
while( r >= b ) {
  q++;       // On incremente le quotient
  r -= b;    // On retranche b au reste => modification d'une
}            // variable dans la condition d'arret
printf( "%d / %d = %d et il reste %d \n", a, b, q, r );
```

SCIENCES SORBONNE UNIVERSITÉ

Les boucles : Exemples

Somme d'entiers

On demandera à l'utilisateur d'entrer des entiers dont on calculera la somme. On s'arrêtera si la valeur entrée est négative.

- On ne connaît pas le nombre de valeurs que l'utilisateur va donner
- On demandera au moins une valeur à l'utilisateur

$\Rightarrow$ on peut utiliser une boucle `do ... while`

```
int i;  // Initialisation pas necessaire
int somme = 0;
do {
  scanf( "%d \n", &i );  // On modifie la variable de la
  if( i > 0 )            // condition de continuation =>
    somme += i;          // Pas de boucle infinie
} while( i > 0 );
printf( "%d \n", somme );
```

Les boucles : Exemples

Conversion boucle for vers while ou do/while

```c
// Initialisation, test et incrementation dans le for
int i;
for( i = 0; i < 4; i++ ) {
   printf( "%d \n", i );
}
```

```c
int i = 0;              // Initialisation
while( i < 4 ) {        // Test
   printf( "%d \n", i );
   i++;                 // Incrementation
}
```

```c
int i = 0;              // Initialisation
do {
   printf( "%d \n", i );
   i++;                 // Incrementation
} while( i < 4 );       // Test
```

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Exemple introductif

Signe d'un entier

```
int a;
scanf( "%d\n", &a );
if( a > 0 ) printf( "a = %d est positif\n", a );
else        printf( "a = %d est negatif ou nul\n", a );

...   // Des instructions modifiant a

if( a > 0 ) printf( "a = %d est positif\n", a );
else        printf( "a = %d est negatif ou nul\n", a );

int b;  // Une nouvelle variable
scanf( "%d\n", &b );
if( b > 0 ) printf( "b = %d est positif\n", b );
else        printf( "b = %d est negatif ou nul\n", b );
```

Problème

Beaucoup de répétitions $\Rightarrow$

- Code plus long (donc moins lisible)
- Nombreuses modifications si on souhaite différencier le cas négatif et nul (et risque d'oubli)

SCIENCES SORBONNE UNIVERSITÉ

Exemple introductif

Signe d'un entier

```
int a;
scanf( "%d\n", &a );
if( a > 0 ) printf( "a = %d est positif\n", a );
else        printf( "a = %d est negatif ou nul\n", a );

...   // Des instructions modifiant a

if( a > 0 ) printf( "a = %d est positif\n", a );
else        printf( "a = %d est negatif ou nul\n", a );

int b;  // Une nouvelle variable
scanf( "%d\n", &b );
if( b > 0 ) printf( "b = %d est positif\n", b );
else        printf( "b = %d est negatif ou nul\n", b );
```

Solution

On voudrait pouvoir écrire

```
afficher le signe de a;
```

SCIENCES SORBONNE UNIVERSITÉ

Fonctions

Qu'est ce qu'une fonction ?

- Une fonction définit le programme permettant de résoudre un problème.
- Ce problème peut lui-même être décomposé (appel à d'autres fonctions).

$\Rightarrow$ un des mécanismes de base en programmation

Pourquoi écrire des fonctions ?

- Factorisation de code répétitif $\Rightarrow$ moins de travail pour le programmeur
- Amélioration de la lisibilité $\Rightarrow$ donne un nom à du code
- Découpage fonctionnel du programme $\Rightarrow$ décompose un problème en sous-problèmes
- Correction d'erreurs plus facile $\Rightarrow$ on peut tester chaque fonction séparément, 1 seul endroit à modifier
- Ré-utilisabilité / Mutualisation de code $\Rightarrow$ on peut réutiliser des fonctions dans d'autres projets ou les partager avec d'autres programmeurs.

SCIENCES SORBONNE UNIVERSITÉ

Déclaration

Syntaxe

```
typeRetour nomMethode( parametres ) {
  // Corps de la methode
}
```

- `typeRetour` :
  - si la fonction retourne une valeur : type de la valeur retournée (`int`, `double`, *etc.*)
  - si la fonction ne retourne rien : `void`
- `parametres` : liste des variables (ou paramètres) nécessaires à la fonction. S'il y a plusieurs paramètres, ils sont séparés par une virgule

Exemples

```
void afficher( int a ) {
  printf( "a = %d", a );
}
int somme( int a, int b ) {
  return a + b;
}
int main( ) {
  ...
  return 0;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Paramètre(s) d'une fonction

Syntaxe

```
typeRetour nomMethode( parametres ) {
  // Corps de la methode
}
```

Principe

On donne dans la parenthèse après le nom de la fonction :

- la liste des paramètres (type et nom) de la fonction
- le type de chaque paramètre doit être précisé
- les paramètres sont séparés par une virgule

Remarque : si la fonction n'a aucun paramètre $\Rightarrow$ parenthèse vide

SCIENCES SORBONNE UNIVERSITÉ

Paramètre(s) d'une fonction

Exemples

```
// Fonction sans parametre
void afficheLoutre( ){
  printf("Vive les loutres!\n");
}

// Fonction avec un parametre
void afficheSigne( int n ){
  if( n >= 0 )      printf( "n est positif\n" );
  else if( n < 0 ) printf( "n est negatif\n" );
}

// Fonction avec 2 parametres de types differents
void afficheSigne( int n, char* nom ){
  if( n >= 0 )
    printf( "%s est positif\n", nom );
  else if( n < 0 )
    printf( "%s est negatif\n", nom );
}
```

SCIENCES SORBONNE UNIVERSITÉ

Paramètre(s) d'une fonction

Exemples

```
// Fonction avec 2 parametres de meme type
void afficheMax( int a, int b ){
  if( a >= b )
    printf( "Le max est %d\n", a );
  else
    printf( "Le max est %d\n", b );
}

// Fonction avec 3 parametres
void afficheMax( int a, int b, double c ){
  double m = a;
  if( m < b )     m = b;
  if( m < c )     m = c;
  printf( "Le max est %d\n", m );
}
```

Valeur de retour

Syntaxe

```
typeRetour nomMethode( parametres ) {
  // Corps de la methode
}
```

Fonction avec valeur de retour

- Valeur de retour
  - résultat renvoyé à la fin de l'exécution de la méthode
  - on précise le type du résultat (`int`, `double`, *etc.*) dans la déclaration de la fonction
- Mot clé `return`
  - arrête l'exécution de la fonction
  - renvoie la valeur qui suit
- Attention : la valeur qui suit `return` doit être du même type que celui indiqué dans la déclaration de la méthode

Valeur de retour

Syntaxe

```
typeRetour nomMethode( parametres ) {
  // Corps de la methode
}
```

Fonction sans valeur de retour

Pas de valeur de retour ⇒

- l'exécution de la fonction s'arrête à la fin du bloc
- `void` comme type de retour
- Pas de `return`

Valeur de retour

Exemples

```
int somme( int a, int b ) {
  int res = a + b;
  return res; // Ok: res est un int
}

double somme( int a, int b ) {
  return a + b; // Ok: cast implicite de int vers double
}

int max( int a, int b ) {
  if( a >= b ) return a;  // Ok: valeur de retour
  else return b;          // connue pour les 2 cas
}

void affiche( int x ) {
  printf( "x = %d\n", x );
  // Ok: valeur de retour = void => pas de return
}
```

Valeur de retour

Quelques erreurs (syntaxiques)

```
int somme( int a, int b ) {
  int res = a + b;          // oubli du return
}
int somme( int a, int b ) {
  return a + b;
  // la ligne suivante ne sera jamais executee
  printf( "Le resultat est %d", (a+b) );
}
int somme( int a, double b ) {
  return a + b;             // a + b est un double
}
int max( int a, int b ) {
  if( a >= b )  return a;  // oubli du return si a < b
}
void affiche( int x ) {
  printf( "x = %d\n", x );
  return 1;  // pas de return car void
}
```

SCIENCES SORBONNE UNIVERSITÉ

Écriture de fonctions

Conseils

- Commencer par se demander de quoi la fonction a besoin (paramètres) et si elle renvoie un résultat (de quel type) ou pas
- Différencier les fonctions qui réalisent des affichages et les fonctions qui retournent des résultats
  $\Rightarrow$ une fonction qui produit un résultat n'affiche généralement rien !

[Alan Perlis]

"Si vous avez une fonction avec 10 paramètres, vous en avez probablement oublié."

SCIENCES SORBONNE UNIVERSITÉ

Écriture de fonctions

Bonnes pratiques

- Regarder s'il n'existe pas déjà une fonction qui résout le problème
- Écrire le plus de fonctions possible et les plus petites possible
- Mettre un commentaire au dessus pour expliquer ce que fait la fonction et comment elle s'utilise.

SCIENCES SORBONNE UNIVERSITÉ

Appel d'une fonction

Principe

- La définition d'une fonction n'exécute pas le programme (suite d'instructions) qui la compose. Il faut l'appeler depuis une autre fonction.
- Paramètres formels, paramètres effectifs :
  - Les paramètres formels sont utilisés lors de la définition de la fonction.
  - Lors de l'appel, des valeurs doivent être données à ces paramètres → paramètres effectifs.
  - Un paramètre effectif est soit une variable (sa valeur), soit une valeur d'expression, soit une valeur de retour d'un appel de fonction.

Syntaxe

- Fonction sans valeur de retour :

```
nomMethode( paramètres_effectifs ) ;
```

- Fonction avec valeur de retour :

```
variable = nomMethode( paramètres_effectifs ) ;
```

SCIENCES SORBONNE UNIVERSITÉ

Appel d'une fonction

Remarques

- L'appel d'une fonction peut se faire uniquement à l'intérieur d'une autre fonction.
- La seule fonction qui n'a pas besoin d'être appelée explicitement est la fonction `main`.
- La méthode doit toujours être déclarée avant d'être appelée (sinon le programme ne la connait pas !)
- Le nombre et l'ordre des paramètres formels et des paramètres effectifs d'une fonction doivent être identiques.
- Les variables utilisées au sein d'une fonction doivent être déclarées comme variables locales en tête du corps de la fonction ou doivent correspondre à des paramètres de la fonction.

SCIENCES SORBONNE UNIVERSITÉ

Appel d'une fonction

Exemple

```
int somme( int a, int b ) { return a + b; }

void afficheSomme( int a, int b ) {
  int s = somme( b, a );  // Appel avec 2 variables
  printf( "La somme est %d\n", s );
}

int main( ) {
  int x = 4;
  int res = somme( x, 38 );  // Appel avec 1 variable et
  printf( "res = %d\n", res );     // 1 expression
  afficherSomme( 13, somme( x, 38 ) );  // Appel avec 1 expression
  return 0;                    // et 1 valeur de retour de fonction
}
```

- `??` : paramètres formels de la méthode
- `??` : paramètres effectifs de la méthode

Appel d'une fonction

Quelques erreurs (syntaxiques)

```
double somme( int a, double b ) {
  return a + b;
}

// Appel de somme en dehors d'une fonction
int res = somme( x, 38 );

int main( ) {
  int x = 4;
  double d = 8., res1, res2;

  res1 = somme( x );  // mauvais nombre de parametres
  res2 = somme( d, x );  // mauvais ordre de parametres
  int resI = somme( x, d );  // mauvais type de retour
  return 0;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Portée des variables

Variables locales

- Une variable déclarée à l'intérieur d'une fonction est une variable locale.

Portée des variables

- La portée d'une variable désigne la partie du programme dans laquelle on peut l'utiliser.
- Une variable locale n'est utilisable qu'à l'intérieur de la fonction dans laquelle elle est déclarée.
- Les paramètres formels d'une fonction correspondent à des variables locales initialisées lors de l'appel de la fonction.

Portée des variables

Exemple

```c
int somme( int a, int b ) {
  int res = a + b;
  return res;
}
int main( ) {
  int a = 4;
  printf( "%d\n", somme( 38, a ) );
  return 0;
}
```

- a, b : portée des paramètres de la fonction `somme`
- res : portée de la variable locale de la fonction `somme`
- a : portée de la variable locale de la fonction `main`

Remarques

- Le nom d'une variable passée à une fonction n'a pas d'importance, seule sa valeur est transmise (*e.g.* dans `main`, $a = 4$ et dans `somme`, $a = 38$).

Portée des variables

Quelques erreurs (syntaxiques)

```
int somme( int a, int b ) {
  int n = 2;
  x = a + n;  // x n'est pas defini dans somme
  return a + b;
}

int main( ) {
  int n = 4;
  int x = somme( n, 38 );
  printf( "%d, %d\n", a, b );  // a et b pas definis dans main
}
```

SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par valeur / copie

- La valeur de la variable est copiée à un autre endroit de la mémoire (que la fonction appelante ne connaît pas)
- À la fin de la fonction, la copie est détruite
- Les types primitifs (`int`, `double`, `char`, ...) sont gérés par copie

Conséquences

- Les modifications apportées à une variable passée par copie ne sont donc pas connues de la fonction appelante

SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par valeur sans modifications

```
void modifX( int x ) {
  x = 9;
}

void main( ) {
  int x = 5;
  modifX( x );
  printf( "%d\n", x );
  return 0;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par valeur sans modifications

```
void modifX( int x ) {
  x = 9;
}

void main( ) {
  int x = 5;
  modifX( x );
  printf( "%d\n", x );
  return 0;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par valeur sans modifications

```
void modifX( int x ) {
   x = 9;
}

void main( ) {
   int x = 5;
   modifX( x );
   printf( "%d\n", x );
   return 0;
}
```

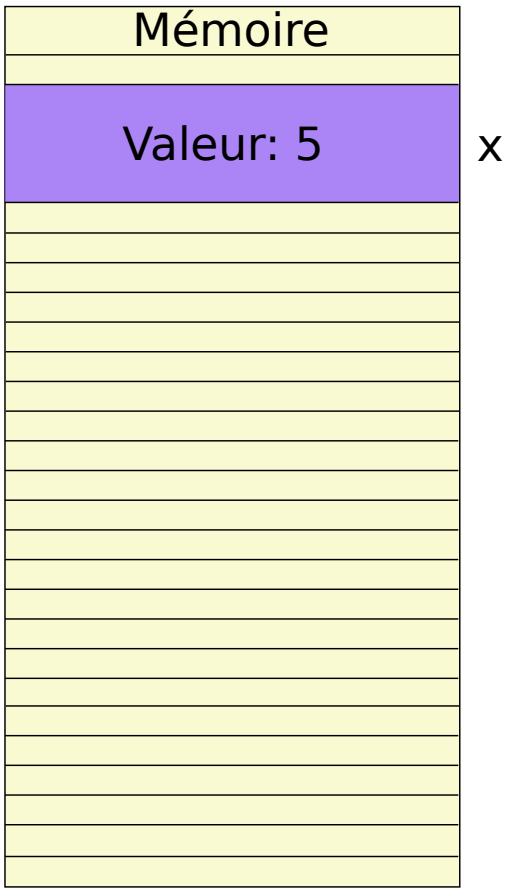
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par valeur sans modifications

```
void modifX( int x ) {
  x = 9;
}

void main( ) {
  int x = 5;
  modifX( x );
  printf( "%d\n", x );
  return 0;
}
```



SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse

- L'adresse de la variable est copiée à un autre endroit de la mémoire.
- À la fin de la fonction, la copie de l'adresse est détruite (mais pas le contenu pointé).

Conséquences

- Les modifications apportées à une variable passée par adresse sont donc connues de la fonction appelante

Rappel

- Une variable contenant l'adresse d'une zone mémoire est appelée pointeur
- Pour accéder à l'adresse d'une variable, on la fait précéder du symbole & (*e.g.* &x)
- Pour accéder au contenu d'un pointeur, on le fait précéder du symbole * (*e.g.* *x)

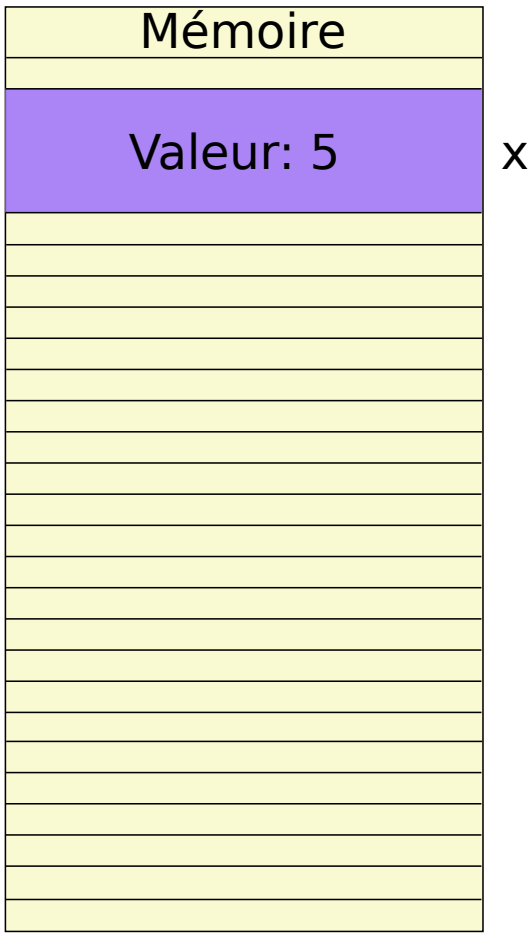
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse avec modifications

```
void modifX( int *pX ) {
  *pX = 9;          // Contenu de pX
}

void main( ) {
  int x = 5;
  int* pX = &x;   // Adresse de x
  modifX( pX );
  printf( "%d\n", x );
  return 0;
}
```



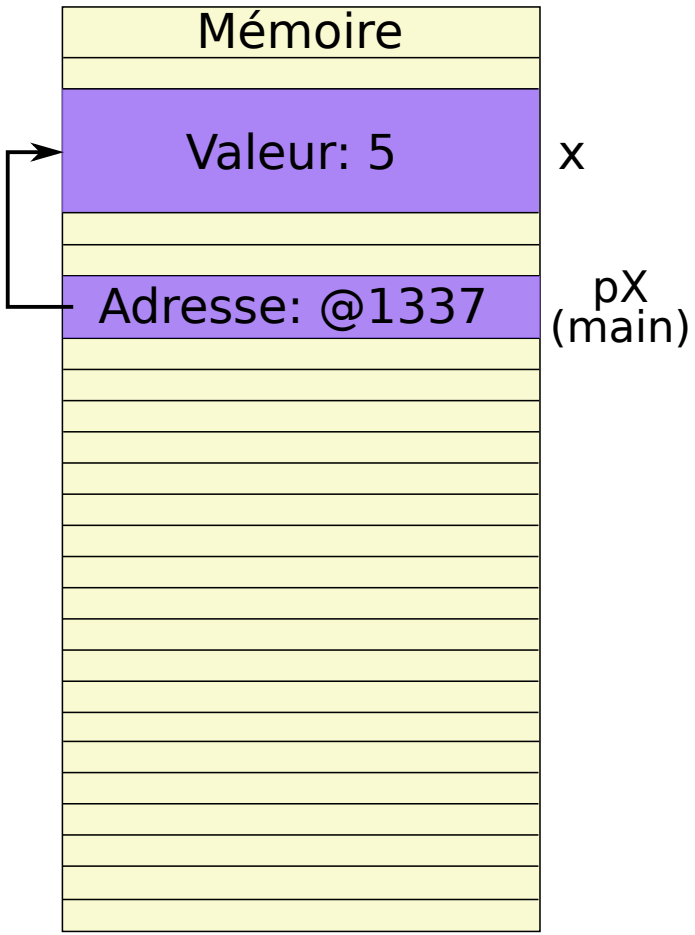
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse avec modifications

```
void modifX( int *pX ) {
  *pX = 9;          // Contenu de x
}

void main( ) {
  int x = 5;
  int* pX = &x ;  // Adresse de x
  modifX( pX );
  printf( "%d\n", x );
  return 0;
}
```



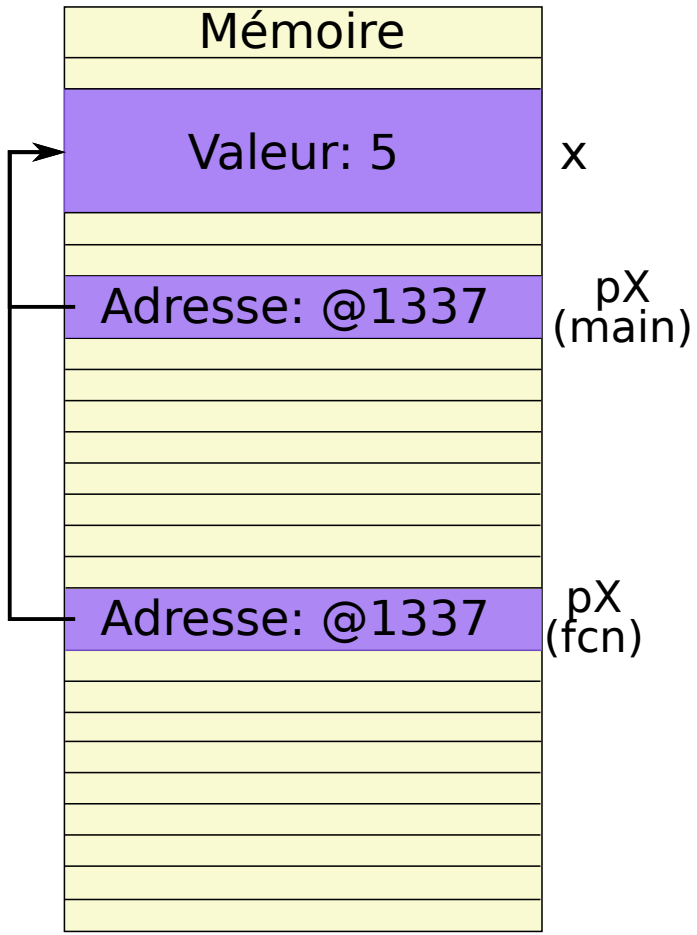
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse avec modifications

```
void modifX( int *pX ) {
  *pX = 9;          // Contenu de x
}

void main( ) {
  int x = 5;
  int* pX = &x;   // Adresse de x
  modifX( pX );
  printf( "%d\n", x );
  return 0;
}
```



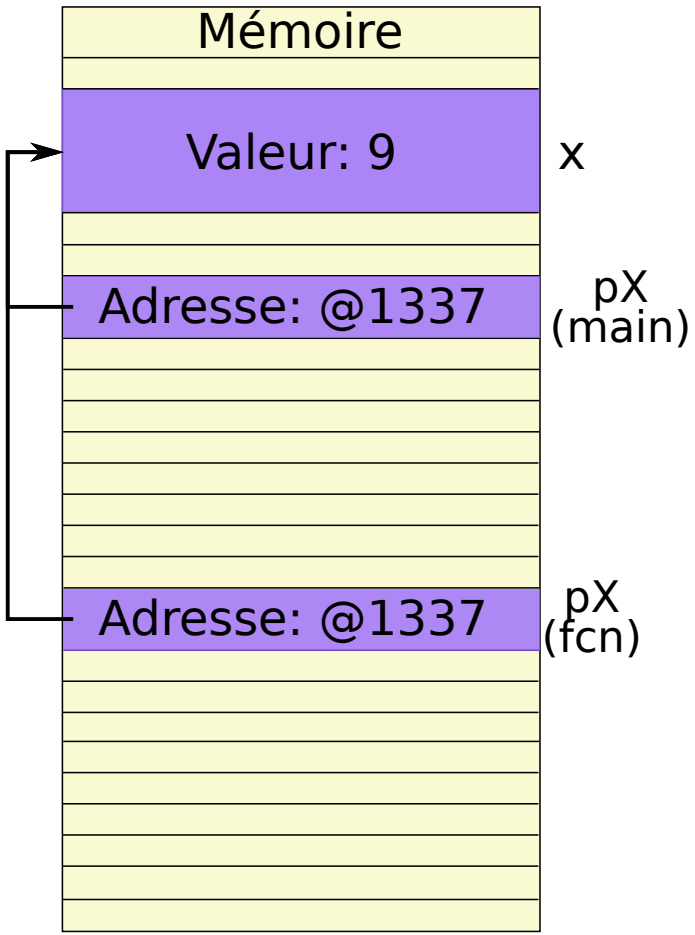
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse avec modifications

```
void modifX( int *pX ) {
   *pX = 9;          // Contenu de x
}

void main( ) {
   int x = 5;
   int* pX = &x;   // Adresse de x
   modifX( pX );   // Adresse de x
   printf( "%d\n", x );
   return 0;
}
```



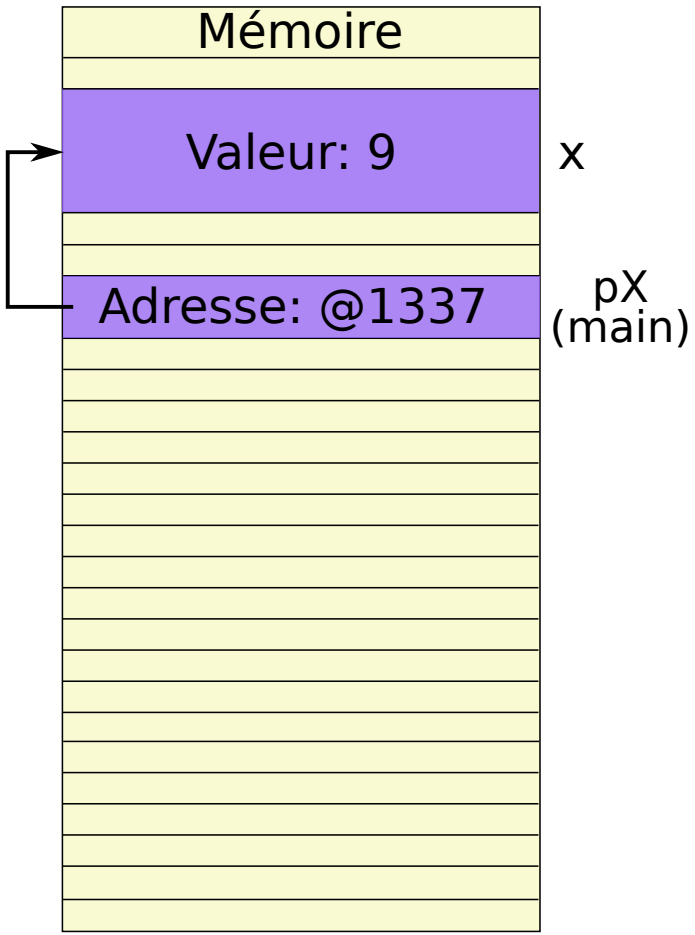
SCIENCES SORBONNE UNIVERSITÉ

Passage de paramètres

Passage par adresse avec modifications

```
void modifX( int *pX ) {
  *pX = 9;          // Contenu de x
}

void main( ) {
  int x = 5;
  int* pX = &x;   // Adresse de x
  modifX( PX );
  printf( "%d\n", x );
  return 0;
}
```



Passage de paramètres

Fonction avec plusieurs résultats

On peut profiter du passage par adresse pour créer des fonctions produisant plusieurs résultats

Exemple : Fonction qui renvoie le maximum et le minimum de 3 entiers

```
void minMax3( int a, int b, int c, int *min, int *max ) {
  if( a > b ) {
    *max = a;  *min = b;
  } else {
    *max = b;  *min = a;
  }
  if( c > *max )  *max = c;
  if( c < *min )  *min = c;
}

void main( ) {
  int x = 5, y = 7, z = 12;
  int min, max;
  minMax3( x, y, z, &min, &max );
  printf("minMax3(%d, %d, %d) = (%d, %d)\n", x, y, z, min, max);
  return 0;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Exemple

Calcul de la puissance d'un réel

```c
float powV1(float x, int n) {
  float pX = 1, X = x;
  int N = n;
  while( N > 0 ) {
    if( N % 2 == 0 ) {
      X *= X;
      N /= 2;
    } else {
      pX *= X;
      N--;
    }
  }
  return pX;
}
```

```c
float powV2(float x, int n) {
  int i;
  float pX = 1;
  for( i = 1; i <= n; i++ )
    pX *= x;
  return pX;
}

float powV3(float x, int n) {
  if( n == 0 )
    return 1;
  else
    return powV3(x, n-1) * x;
}
```

Lequel des 3 algorithmes est le plus efficace pour calculer x^n ?

SCIENCES SORBONNE UNIVERSITÉ

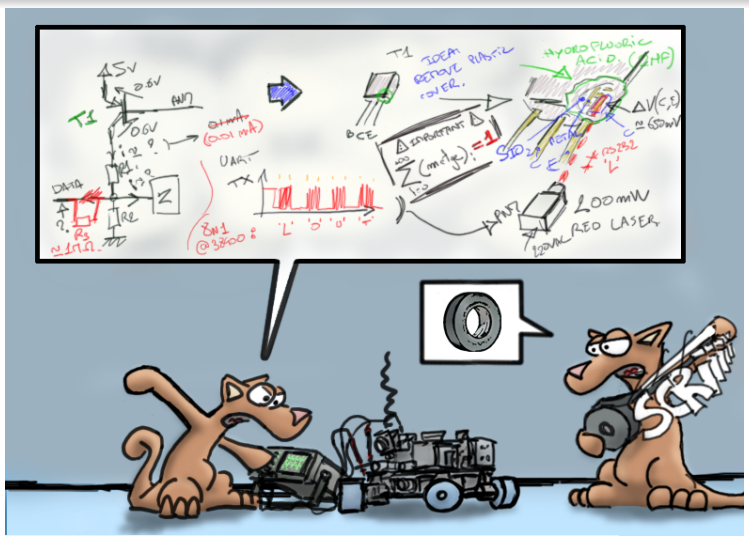
Complexité

Pourquoi parler de compléxité ?

- Estimation des ressources nécessaires pour le fonctionnement de l'algorithme (temps, mémoire, . . .)
- Estimation du coût d'un passage à l'échelle (de 10 échantillons à 100, 1000, . . .)
- Comparaison d'algorithmes
- Optimisation

[Alan Perlis]

"Vous ne pouvez pas communiquer la complexité, juste en faire prendre conscience."



SCIENCES SORBONNE UNIVERSITÉ

Complexité

Qu'est ce que la compléxité d'un algorithme ?

- **Nombre d'opérations à effectuer**
- Mémoire nécessaire pour stocker les données et variables
- Temps de calcul
- ...

Comment estimer la compléxité d'un algorithme ?

- Moyenne
- Pire des cas
- **Estimation à un facteur près**
- ...

On ne s'intéressera qu'à l'estimation du nombre d'opérations à effectuer à un facteur près

SCIENCES SORBONNE UNIVERSITÉ

Complexité

Notation

La complexité d'un algorithme est noté $\mathcal{O}(x)$

Définitions

- une instruction élémentaire est de complexité $\mathcal{O}(1)$
- $n * \mathcal{O}(1) = \mathcal{O}(n)$
- $\mathcal{O}(n) + \mathcal{O}(n) \sim \mathcal{O}(n)$
- $\mathcal{O}(2n) \sim \mathcal{O}(n)$
- $\mathcal{O}(n^p) + \mathcal{O}(n^q) \sim \mathcal{O}(n^p)$ avec $p \geq q$
- $n * \mathcal{O}(n^p) = \mathcal{O}(n^{p+1})$

SCIENCES SORBONNE UNIVERSITÉ

Complexité

Estimation

De manière grossière, on peut dire que répéter n fois

- des instructions élémentaires $\Rightarrow \mathcal{O}(n)$
- des instructions de complexité $\mathcal{O}(n^p) \Rightarrow \mathcal{O}(n^{p+1})$
- une opération en divisant une taille par 2 (ou plus) $\Rightarrow \mathcal{O}(log(n))$

SCIENCES SORBONNE UNIVERSITÉ

Complexité

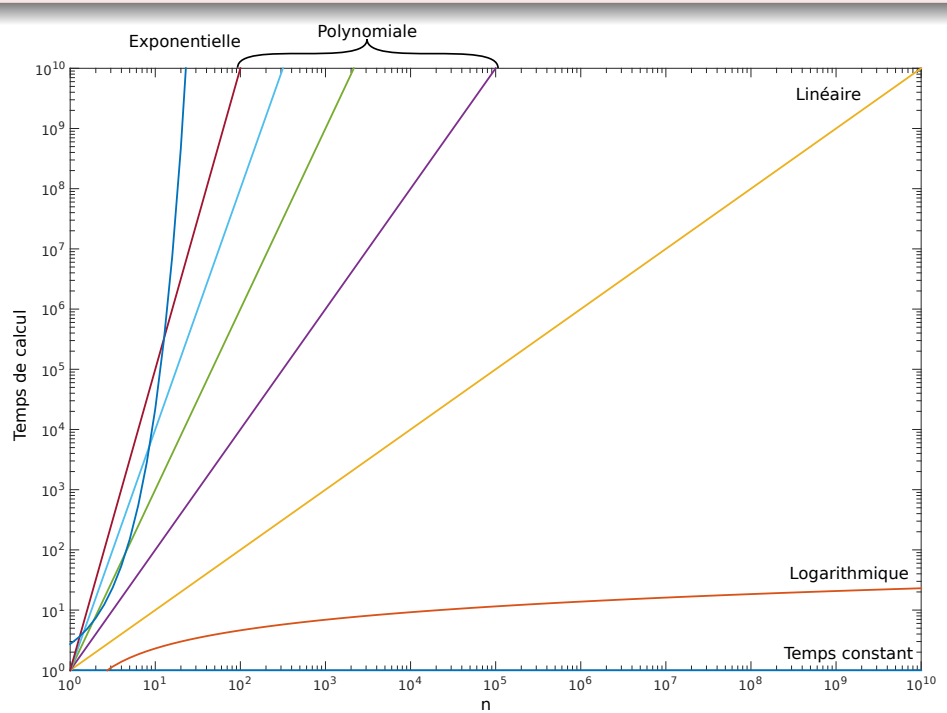
Comparaison

On préfèrera une compléxité

- $\mathcal{O}(\log(n))$ à $\mathcal{O}(n)$
- $\mathcal{O}(n^q)$ à $\mathcal{O}(n^p)$ avec $p \geq q$

Remarque

Un algorithme de complexité plus faible qu'un autre n'est pas forcément plus simple à implémenter !



SCIENCES SORBONNE UNIVERSITÉ

Exemple : Calcul de la puissance d'un réel

Algorithmes à comparer

```
float powV1(float x, int n) {
  float pX = 1, X = x;
  int N = n;
  while( N > 0 ) {
    if( N % 2 == 0 ) {
      X *= X;
      N /= 2;
    } else {
      pX *= X;
      N--;
    }
  }
  return pX;
}
```

```
float powV2(float x, int n) {
  int i;
  float pX = 1;
  for( i = 1; i <= n; i++ )
    pX *= x;
  return pX;
}

float powV3(float x, int n) {
  if( n == 0 )
    return 1;
  else
    return powV3(x, n-1) * x;
}
```

SCIENCES SORBONNE UNIVERSITÉ

Exemple : Calcul de la puissance d'un réel

Complexité

```
float powV2( float x, int n ) {
  int i;  // Declaration sans affectation => pas de complexite
  float pX = 1;                  // O(1) +
  for( i = 1; i <= n; i++ )      // n *
    pX *= x;                     // O(1)
  return pX;
}
```

Conclusion : $\mathcal{C} = \mathcal{O}(1) + n\mathcal{O}(1) = \mathcal{O}(1) + \mathcal{O}(n) = \mathcal{O}(n)$

SCIENCES SORBONNE UNIVERSITÉ

Exemple : Calcul de la puissance d'un réel

Complexité

```
float powV3( float x, int n ) {
  if( n == 0 ) {
    return 1;                    // O(1)
  } else {
    return powV3(x,n-1)*x;       // O(1)
  }
}
```

Conclusion : $\mathcal{C} = \mathcal{O}(1) \times n = \mathcal{O}(n)$

Exemple d'appel : powV3$(1.5, 22)$

0. powV3$(1.5, 22)$
1. powV3$(1.5, 21)$
2. powV3$(1.5, 20)$
3. ...
20. powV3$(1.5, 2)$
21. powV3$(1.5, 1)$
22. powV3$(1.5, 0)$

SCIENCES SORBONNE UNIVERSITÉ

Exemple : Calcul de la puissance d'un réel

Complexité

```
float powV1(float x, int n){
  float pX = 1, X = x;
  int N = n;
  while( N > 0 ) {
    if( N % 2 == 0 ) {
      X *= X;
      N /= 2;
    } else {
      pX *= X;
      N--;
    }
  }
  return pX;
}
```

Meilleur des cas :

$n = 2^k \Rightarrow$ on passe k fois dans la boucle ("si n pair" est toujours vrai)

$\mathcal{C} = \mathcal{O}(k)$

Pire des cas :

$n = \underbrace{(((2+1)2+1)2+\ldots)2+1}_{k \text{ fois}}$

$n = \sum_{i=0}^{k-1} 2^i = 2^k - 1 \Rightarrow$ on passe $2k$ fois dans la boucle ("si n pair" est vrai 1 fois sur 2)

$\mathcal{C} = \mathcal{O}(2k) \sim \mathcal{O}(k)$

Et $k = \log(n)/\log(2)$,
d'où $\mathcal{C} = \mathcal{O}(k) = \mathcal{O}(\log(n)/\log(2))$

$\mathcal{C} = \mathcal{O}(\log(n))$

SCIENCES SORBONNE UNIVERSITÉ

Exemple : Calcul de la puissance d'un réel

Comparaison

- powV1 : la puissance n est divisée par 2 chaque fois que n est pair $\Rightarrow \mathcal{O}(\log(n))$
- powV2 : une boucle (on répète une instruction élémentaire n fois) $\Rightarrow \mathcal{O}(n)$
- powV3 : la fonction powV3 s'appelle n fois (récursivité) $\Rightarrow \mathcal{O}(n)$

On préfèrera donc l'algorithme powV1.

Remarque

Les algorithmes powV2 et powV3 ont une complexité (en nombre d'opérations) équivalente mais powV3 nécessite beaucoup plus d'espace mémoire. On préfèrera donc powV2 à powV3.

SCIENCES SORBONNE UNIVERSITÉ

Plan

SCIENCES SORBONNE UNIVERSITÉ

Exercice

Calcul de $\sin(x)$

On souhaite calculer les valeurs de $\sin(x)$ avec une précision ϵ. On rappelle que

$$\sin(x) = \sum_{n=0}^{+\infty} \left(\frac{(-1)^n x^{2n+1}}{(2n+1)!} \right)$$

SCIENCES SORBONNE UNIVERSITÉ

Exercice

Calcul de $\sin(x)$

On souhaite calculer les valeurs de $\sin(x)$ avec une précision ϵ. On rappelle que

$$\sin(x) = \sum_{n=0}^{+\infty} \left(\frac{(-1)^n x^{2n+1}}{(2n+1)!} \right)$$

Algorithme de $\sin(x)$

Entrées : x, ϵ : réel
Variables : n : entier
uN, x2, sinX : réel
sinX $\leftarrow$ x, n $\leftarrow$ 0
uN $\leftarrow$ x, x2 $\leftarrow$ x * x
tant que $|uN| \geq \epsilon$ **faire**
    uN $\leftarrow$ -uN * x2 / ((2n+2)(2n+3))
    sinX $\leftarrow$ sinX + uN
    n $\leftarrow$ n + 1
écrire sinX

SCIENCES SORBONNE UNIVERSITÉ

Exercices

Le Rubik's cube

Un Rubik's Cube est un cube composé de plusieurs mini-cubes qui pivotent autour du centre selon 3 axes.

Question

- Écrire un programme qui affiche le nombre de mini-cubes visibles sur un cube de taille N donnée.

SCIENCES SORBONNE UNIVERSITÉ

Exercice

Entiers particuliers

- Un entier n est divisible par p si le reste de la division de n par p est nul
- Un entier n est premier si il n'est divisible que par 1 et lui même
- Un entier n est parfait si la somme de ses diviseurs est égale à n

Questions

- Écrire une fonction qui dit si un entier est premier
- Écrire une fonction qui dit si un entier est parfait
- Écrire une fonction qui calcule la somme de la somme des diviseurs de tous les nombres inférieurs à N
  Exemple : si $d(n)$ est l'ensemble des diviseurs de n, alors :
  - $d(1) = \{1\}$, $d(2) = \{1, 2\}$, $d(3) = \{1, 3\}$, $d(4) = \{1, 2, 4\}$
  - leur somme $S(4) = 1 + 1 + 2 + 1 + 3 + 1 + 2 + 4 = 15$.
- Écrire un programme qui affiche tous les entiers premiers inférieurs à N puis tous les entiers parfaits inférieurs à N

Exercice

Calcul de $\sqrt{x}$

Calculer $\sqrt{x}$ à l'aide de la suite : $u_{n+1} = \frac{1}{2}\left(u_n + \frac{x}{u_n}\right)$ et $u_0 = x$

Algorithme de $\sqrt{x}$

Entrées : x, ε : réel
Variables : uN, uN1 : réel
uN $\leftarrow$ x
répéter
    uN1 $\leftarrow$ uN
    uN $\leftarrow$ (uN1 + x/uN1)/2
tant que $|uN - uN1| \leq \varepsilon$
écrire uN