Polycopié de Programmation Structurée IMA3

— Version 2011/2012 —

Polytech'Lille, Villeneuve d'Ascq

Laure GONNORD

Premature optimization is the root of all evil (or at least most of it) in programming.

Donald Knuth, *Décembre 1974, Conférence du Prix Turing 1974, Communications of the ACM.*

Afin d'améliorer ce (déjà magnifique) poly n'hésitez pas à me soumettre toute critique, suggestion, remarque ou correction, dans mon casier ou, électroniquement, à l'adresse

`Laure.Gonnord@polytech-lille.fr`

Table des matières

1 Motivations et premiers pas en C **4**

2 Algorithmique/programmation C de base **10**
2.1 Concepts de base 10
2.2 Programmes en C 21

3 Fonctions/Actions **29**
3.1 Actions/fonctions : notions de base 29
3.2 Notions de complexité et de correction 36
3.3 Actions/fonctions récursives 40

4 Vecteurs/Tableaux **44**

5 Algorithmique du Tri **53**

6 Variables modifiables en C : les pointeurs **60**
6.1 Notions de base sur les pointeurs 60
6.2 Pointeurs et tableaux et chaînes de caractères 74

Chapitre 1

Motivations et premiers pas en C

Dans ce cours nous abordons le concept de système informatique et nous motivons l'apprentissage de l'algorithmique et de la programmation.
Un premier programme C est étudié. Une démo illustre l'édition du programme, sa compilation, son exécution.

Savoir répondre aux questions

- Qu'est-ce qu'un système informatique ?
- Qu'est-ce qu'un fichier source ?
- Que fait la compilation ?
- Comment compiler avec `gcc` le fichier `toto.c` ?

Algorithmique et Programmation, IMA

Cours 1 : Introduction et Hello World

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Introduction
- Pourquoi ?

2 Premier programme en C

1 Introduction
- Pourquoi ?

2 Premier programme en C

Systèmes informatiques

Un système informatique :

- est conçu pour automatiser le traitement d'une tâche
- est divisé en matériel (stockage, periphériques, unité centrale, . . .) et logiciel.
- logiciels/applications : gestion, jeux, bureautique, traitement de données, . . .
- un système d'exploitation fait le lien et gère les ressources

Systèmes informatiques

Différents types :

- temps réel (contraintes de temps) : contrôle d'une chaîne de production, ou le contrôle commande d'un avion.
- transactionnels : contrôles de bases de données, ...
- embarqués (pda), distribués (réservation de train), ...

Développement logiciel

Création, développement de logiciels suivant les besoins et les offres matérielles :

- Systèmes complexes
- Gros logiciels

Programmation structurée

Objectifs de l'enseignement :

- **Conception** de bout en bout d'un logiciel : du cahier des charges à l'implémentation et la documentation.
- **Analyse** du problème initial et hiérarchisation des priorités : notion de sous problème.
- Réflexion sur la correction d'une solution et de son implémentation : **Preuve d'invariants**.
- Réflexions sur la pertinence des solutions et leur coût d'exécution : **Analyses de complexité**.
- Travail contraint en équipe et en temps : **Réalisation de projet**.

Placement dans les enseignements IMA

Cours **prérequis** des enseignements de : réseaux, systèmes, programmation avancée, programmation objet, ...

mais aussi analyse numérique, électronique, automatique !

1 Introduction

2 Premier programme en C

Premier programme

```
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return 0;
}
```

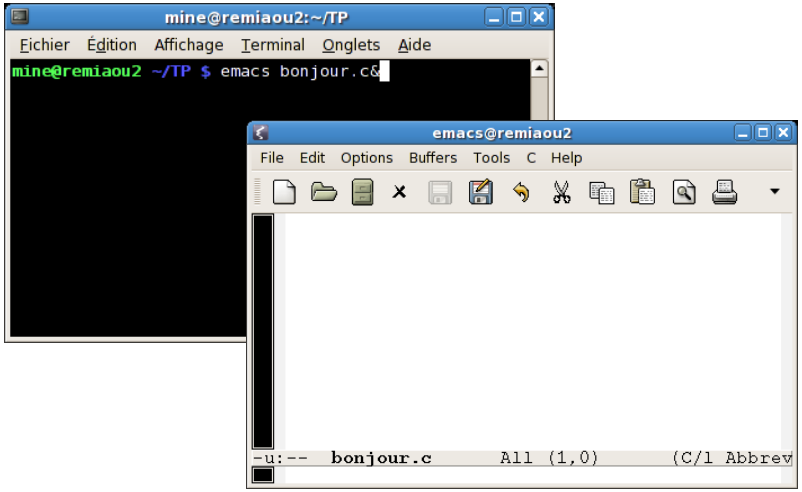
Effet :

- affiche `Bonjour tout le monde!`,
- retourne le code 0 (tout s'est bien passé).

Édition, compilation et exécution

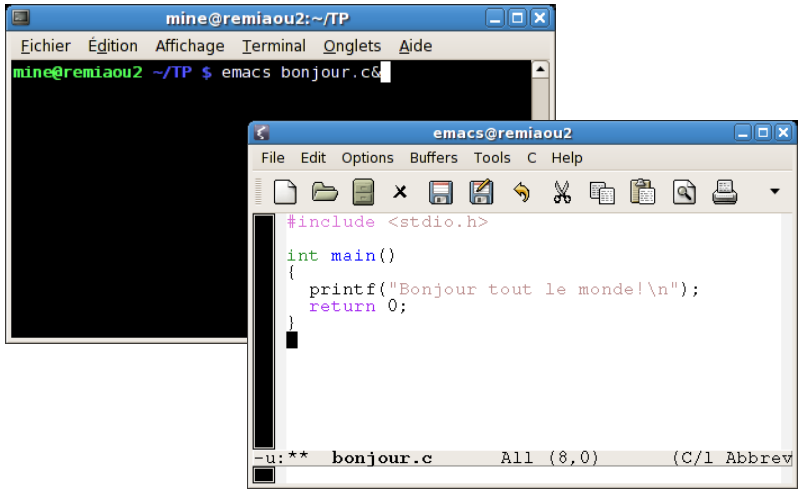
Lancement de l'éditeur en tâche de fond (&).

Édition, compilation et exécution



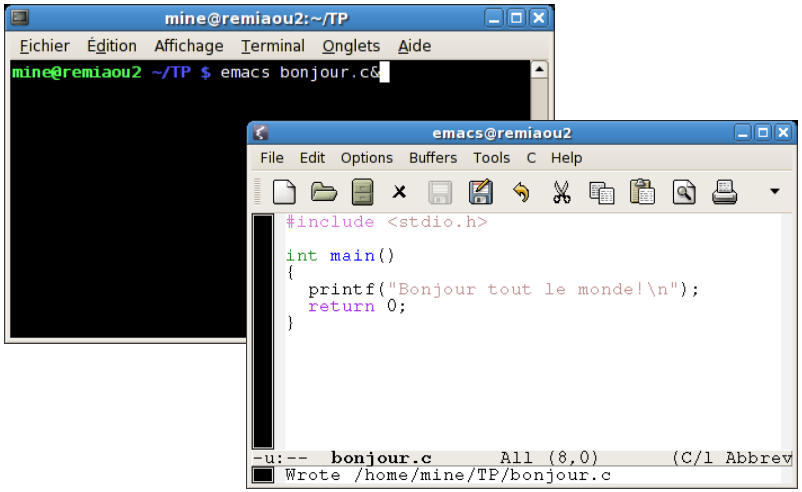
Lancement de l'éditeur en tâche de fond (&).

Édition, compilation et exécution



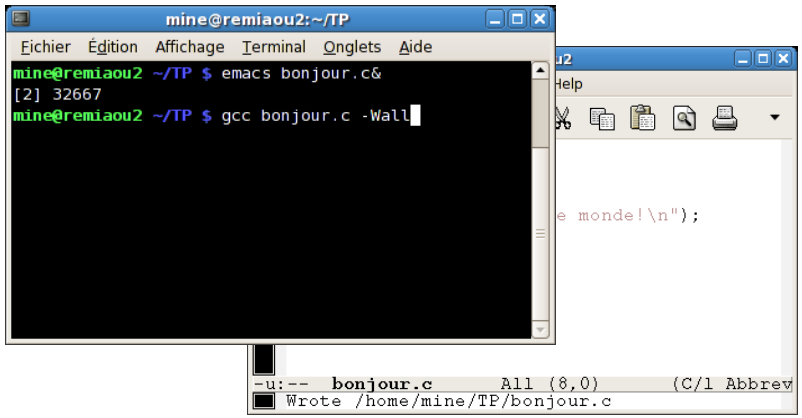
On tape le texte du programme : **édition.**

Édition, compilation et exécution



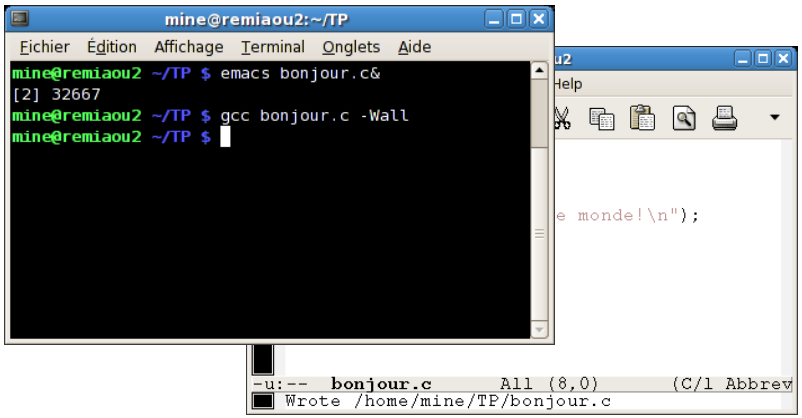
Il ne faut pas oublier de sauvegarder.

Édition, compilation et exécution



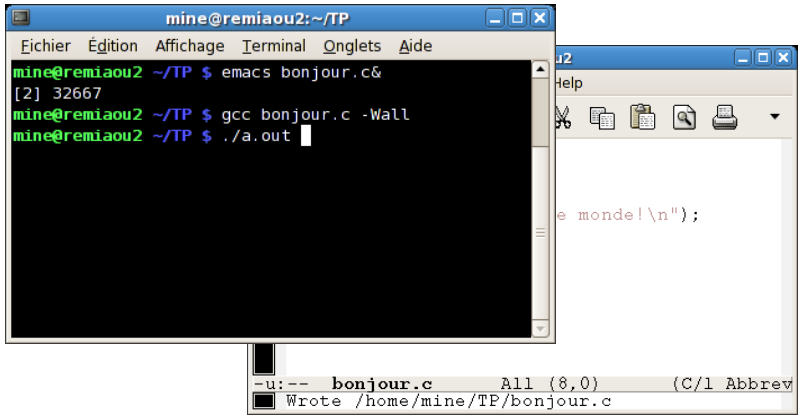
Lancement de la **compilation** avec `gcc`.

Édition, compilation et exécution



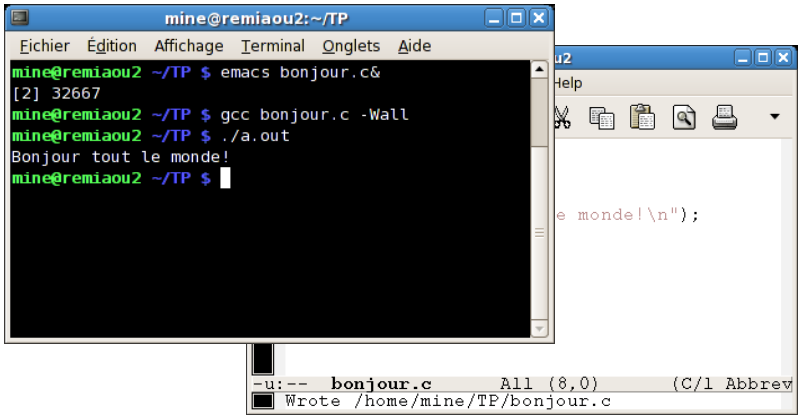
Si le compilateur ne dit rien, tout s'est bien passé.
Un fichier `a.out` "binaire" a été créé.

Édition, compilation et exécution



Lancement de l'exécutable.

Édition, compilation et exécution

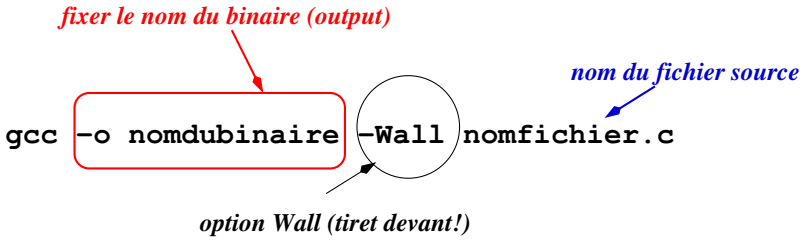


Le programme s'**exécute** et rend la main.

Binaire généré

Le fichier `a.out` généré est un fichier **binaire** (compréhensible par l'ordinateur. On peut donner n'importe quel nom à ce binaire : `gcc hello.c -o hello` et ensuite l'exécuter avec :`./hello`

Ligne de compilation à connaître



ou encore (l'ordre est indifférent) :

- `gcc -Wall nomfichier.c -o nombinaire`
- `gcc nomfichier.c -o nombinaire -Wall`

Chapitre 2

Algorithmique/programmation C de base

2.1 Concepts de base

Dans ce cours nous abordons les concepts fondamentaux en algorithmique que sont la notion de constante, de variable, de type, d'expression, de boucle. Un **pseudo-code** *sera utilisé pour décrire les programmes. L'équivalent en langage C est aussi donné.*

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Qu'est-ce qu'une variable ?
- Qu'est-ce que le type d'une variable ? Connaître les types de base.
- Qu'est-ce qu'une constante ? Savoir déclarer une constante symbolique en C.
- Donner un exemple d'expression numérique / d'expression booléenne.
- Qu'est-ce qu'une affectation ?
- Soit l'instruction $x \leftarrow 42 + 23$;. Expliquer ce que fait le programme lorsqu'il rencontre cette instruction.
- Connaître les tests, la boucle pour, la boucle while (ce que ça fait, et la syntaxe).

Algorithmique et Programmation, IMA

Cours 2 : C Premier Niveau / Algorithmique

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1. Notations, identificateurs
2. Variables et Types de base
3. Expressions
4. Constantes
5. Instructions
   - Instruction simple, instruction composée
   - Structures de contrôle
   - Itérations

1. Notations, identificateurs
2. Variables et Types de base
3. Expressions
4. Constantes
5. Instructions

Notations

Notations **algorithmiques** :

Faire
| action
Tantque *condition* ;

Exemples en C :

```
char c;
```

Identificateurs

Mot désignant des variables, fonctions, types.

- Suite de charactères, chiffres et '_' (underscore) ;
- Commence par une lettre
- Distinction majuscule/minuscule

Convention : les **variables** sont en minuscule.

```
toto , a23_plouf , a89zZ_10
```


1 Notations, identificateurs
2 Variables et Types de base
3 Expressions
4 Constantes
5 Instructions

Variables

Une **variable** est une place en mémoire qui a un **nom** (convention : en minuscules) :

- Une variable a un **type** qui définit quelles opérations sont valides (entier, booléen, réel, caractère, . . .)
- Elle doit être déclarée AVANT d'être utilisée.

Une **déclaration de variable** est la donnée d'un type et d'un nom (identificateur).

▶ Important ! Déclarer une variable d'un certain type interdit de l'utiliser pour stocker des informations d'un autre type !

Type entier

Caractéristiques :

- Codé sur 2 (ou 4 octets, ou 8) : range = $[-2^{15}, 2^{15} - 1]$
- `sizeof(int)` rend 2 ou 4 ou 8
- Opérateurs : +, *, /, %(reste modulo), << (shift)
- Comparaison : !=, ==, <=

Déclaration pseudo-code

x : Entier

Déclaration en C

```
int x;      // declaration simple
int z=10; // declaration avec valeur initiale
```

Type booléen

Caractéristiques :

- N'existe pas en C : `int`,
- Représentation : deux valeurs entières, 0 pour faux, 1 pour vrai (en fait toute valeur différente de 0) : `stdbool`
- Opérateurs et (`&&`), ou (`||`) : paresseux de gauche à droite

Déclaration pseudo-code

b : Booléen

Déclaration en C

```
#include <stdbool.h>
bool a;
bool b=false; //avec initialisation
```


Type réel

Caractéristiques :

- Float 4 octets et double 8.
- Notation décimale ou exponentielle (12.3, -.38, .5e-11)
- Opérateurs : mêmes que int sauf %. / est la division réelle.

Déclaration pseudo-code

r : Réel

Déclaration en C

```
float x;        // declaration simple
float r=0.34; // declaration avec valeur initiale
```


Type caractère - 1/2

Caractéristiques :

- 1 octet : 256 valeurs de l'ASCII étendu
- Notation `'a'`
- Caractères spéciaux `\n` saut de ligne, `\t` tabulation, ...

Déclaration pseudo-code

c : Caractère

Déclaration en C

```
char c;       // declaration simple
char c='a'; // declaration avec valeur initiale
```


Type caractère - 2/2

En C : un caractère est un entier (les valeurs de l'ascii), donc :

```
int i='a';     // fonctionne aussi !
c = 80;        // ascii code 80 == P
char d;
d= c+1;        // d vaut ? Q!
```

▶ Le savoir, mais en général, éviter l'utilisation de la conversion implicite !

Autres types

Les types chaînes de caractères, tableaux, et les types composés seront vus plus tard.

1 Notations, identificateurs
2 Variables et Types de base
3 Expressions
4 Constantes
5 Instructions

Expression numérique, expression booléenne

Expression **numérique** (C/pseudo-code) :

```
1+x+y+41
```

Expression **booléenne en pseudo-code** :
(x<7 **et** y=2) **ou** b

Expression **booléenne en C** :

```
(x<7 && y==2) || b
```

▶ Une expression est constituée d'opérateurs, de sous-expressions, de sous-expressions de base (variable ou constante).

Syntaxe générale des expressions en C

Une **expression C** peut être (entre autres) :

- un identificateur : toto
- une constante : 42
- une chaîne littérale : ''hop''
- une expression numérique
- une expression booléenne
- une expression-affectation (à venir)

en C, l'affectation est une expression !

Expression-affectation

En **C** :

```
x = 7
t[2] = 23
```

En **pseudo-code** :

x ← 7
t[2] ← 23

À gauche de l'affectation : une expression qui doit délivrer une variable (par opp. à constante) : une variable simple, ou un élément de tableau.

Sémantique :

- Effet de bord : la valeur de droite est calculée et affectée à la variable de gauche.
- (en C) La valeur de l'expression entière est cette valeur calculée : `x = (y=8) +1` est une expression dont la valeur vaut .....

1 Notations, identificateurs
2 Variables et Types de base
3 Expressions
4 Constantes
5 Instructions

Qu'est-ce qu'une constante ?

Une **constante** est une valeur qui ne change pas tout au long d'un programme.

Cas d'utilisation : écrire du **code paramétrique** :

- Nombre d'itérations d'un algo ;
- Tailles de tableaux...

Définition de constantes symboliques

En pseudo-code :

```
Entier X : Constante(2)
```

En C :

```
#define CST valeur
```

- `CST` : identificateur, par convention en majuscules,
- `valeur` : texte arbitraire,
- doit occuper une ligne complète,
- pas de point-virgule ; final.

Effet : dans la suite du programme, `CST` est remplacé par `valeur` (preprocessing C)

Danger des constantes symboliques

Définition de constante symbolique $\neq$ affectation de variable !

- affectation : évaluation,
- constante symbolique : substitution littérale.

$\Rightarrow$ danger de “capture” syntaxique.

Exemple d’erreur :

```
#define N x+y
z = 3*N; /* signifie z = 3*x+y, pas z = 3*(x+y) */
         /* x et y peuvent aussi etre symboliques! */
```

Solution :

```
#define N ((x)+(y))      /* plus su^r */
```


1 Notations, identificateurs
2 Variables et Types de base
3 Expressions
4 Constantes
5 Instructions
- Instruction simple, instruction composée
- Structures de contrôle
- Itérations

Notion d’instruction

Une **instruction** est une ligne de pseudo-code/C qui effectue un calcul, qui a un effet sur les variables du programme, . . .

Dans la suite, nous allons voir différentes formes d’instructions :

- les instructions simples
- les instructions composées
- les instructions conditionnelles
- les instructions itération.

Instruction simple en pseudo-code / en C

Instruction **simple en C** : expression suivie d’un ; (point-virgule)

```
x=4 ;                  // affectation
z=42+x;
printf("Hello!") ;     //impression
scanf(%d,&x);          //demande d'un entier
toto(x);               // appel de procedure, (cours 3)
w=f(z,x);              // appel de fonction (cours 3)
```

Attention `2+4;` est donc bien une instruction simple !

En **pseudo-code** c’est pareil :

x $\leftarrow$ 7 ;

t[2] $\leftarrow$ f(z,x) ;

Instruction composée ou bloc - en C

Un bloc (C uniquement) (entre accolades !) permet

- de grouper l'ensemble d'instructions en lui donnant la forme syntaxique d'une seule instruction (voir le IF)
- de déclarer des variables accessibles uniquement à l'intérieur du bloc.

```
{
  int x; // declaration
  x=4 ;
  z=42+x;
}
{
  x=2; //erreur, x non declare dans le bloc
}
```


Conditionnelle - 1

Le test/la **conditionnelle** en pseudo code :

Si *condition* **alors**
| action_alors
Fsi

Si *condition* **alors**
| action_alors
Sinon
| action_sinon
Fsi

condition est une expression booléenne. Si son évaluation donne "true" alors la première action est exécutée, sinon c'est la deuxième.

Conditionnelle - 2

En C cela donne :

```
if (2x+5<=b) printf(''blabla'');

if (a>b) max=a; else max=b;

if (a>b) if c<d u=v; else i=j;
// le else est associe au if le plus proche

if (a)    // teste si a!=0
{
...       // groupement d'instructions (bloc)
}
```


Conditionnelle - 3

Important ! : l'instruction

```
if (x==4) t=3;
```

est différente de :

```
if (x=4) t=3;
```

Cette dernière est fortement déconseillée !

Exercices

Écrire les suites d'**instructions** pour

- Afficher le maximum de deux entiers x et y
- Afficher la valeur absolue de l'entier z
- Afficher `pair` ou `impair` selon la parité de l'entier x.
- Afficher le maximum de 3 entiers

Instruction d'itération : POUR - 1 (version simple)

Utilisation classique avec **compteur**

Pour i **de** *inf* **à** *sup* **Faire**
| corps
Fpour

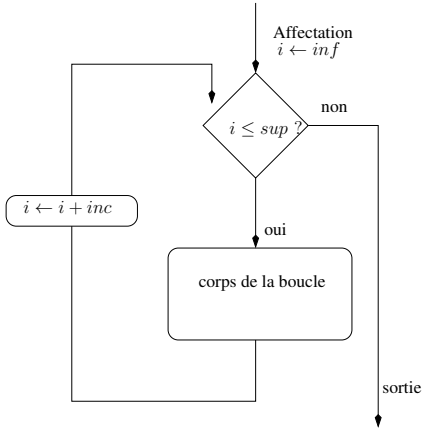
(augmentation implicite de 1 à chaque tour).

En C cela donne :

```
for (i=inf;i<=sup;i=i+inc)
    corps
```

À utiliser en priorité lorsqu'on connaît le nombre d'itérations

Instruction d'itération : POUR - 2



▶ La boucle "pour" est en fait plus générale/complexe en C.
Nous verrons quelques utilisations en TP.

Exercices Boucle POUR

Écrire les suites d'**instructions** pour

- Afficher les entiers de 1 à 10 séparés par des espaces.
- Afficher les entiers de 10 à 1 séparés par des espaces.
- Ajouter les entiers de 1 à 100, puis afficher le résultat.
- Ajouter les entiers pairs de 6 à 2048, puis afficher le résultat.
- Afficher la liste des multiples de 3 et des multiples de 5 (dans l'ordre croissant) inférieurs à 60 ; puis un point.

Instruction d'itération : TANTQUE - 1

La boucle **tant que** en pseudo code :

Tq *condition* **faire**
| action
Ftq

En C cela donne :

```
while (x>0) x=x-1;

while (x>0) {
  x=x-1;
  z=z+x;
}
```


Instruction d'itération : TANTQUE - 2

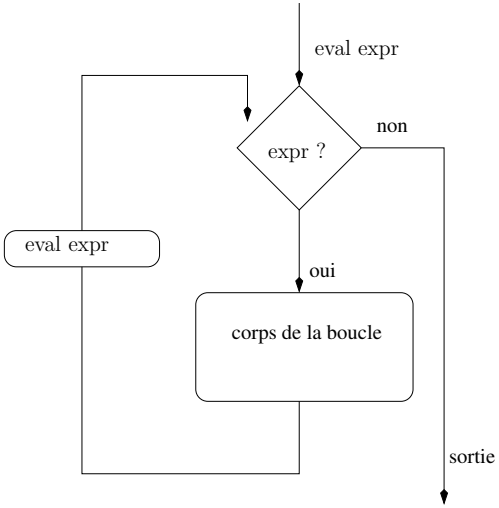
En C :

```
while(expression)
  {
      instructions
  }
```

Sémantique (effet) ▶ Tant que l'expression est vraie, le bloc est exécuté.

- Si la condition est initialement fausse, le bloc n'est jamais exécuté.
- La condition est retestée après chaque tour de boucle.
- Les parenthèses autour de la condition sont **obligatoires**.
- Si une seule instruction : `{` et `}` facultatifs.

Déroulement d'une boucle Tant que



Instruction d'itération TANTQUE - exemple C

Longueur d'une ligne :

```
l=0;c=getchar();
while(c != '\n')
  {
    l=l+1;        //augmentation du compteur
    c=getchar()   //on avance !
  }
```

Instruction d'itération : DO WHILE

Faire
| action
Tantque *condition* ;

En C cela donne :

```
do
    c = getchar();
while (c != '\n');
```

2.2 Programmes en C

Dans ce cours est exposée la syntaxe de programmes C simples. Un programme C ayant une syntaxe particulière, tous les fichiers texte ne sont pas "acceptés" lors de la compilation, nous verrons quels messages d'erreur nous obtenons alors. Nous verrons aussi comment un programme C peut interagir avec son environnement (entrées/sorties au terminal). Le cours se termine par des exercices simples.

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Syntaxe d'un programme C simple.
- Usage et syntaxe de `printf` et `scanf`.
- Qu'est-ce une erreur de compilation ? Comment avoir le plus de messages d'erreurs de compilation possible ?

Algorithmique et Programmation, IMA

Cours 2b : C/Algo : Programmes

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

Syntaxe générale d'un programme

Un programme comprend :

- Une liste de déclarations (de variables globales, de types, de structures, ...) : **optionnelle** ;
- Une liste de définitions de fonctions (cf cours 3) : **optionnelle** aussi ;
- Une fonction main, unique et obligatoire, qui est le point d'entrée du programme

En **pseudo-code**

```
...
Fonction main()
    Imprimer("bonjour")
    ...
    Retourner 0
FFonction
```

Syntaxe générale d'un programme - C

```c
#include <stdio.h> // liste de defs de fonctions (lib)

//autres defs de fonctions (internes)
...


int main()
{
  printf("Hello world!\n");
  return 0; // convention obligatoire dans ce cours
}
```


Syntaxe générale du main - C

(cf poly p 22, le main est un cas particulier de **fonction**)

```c
int main()
{
  //declarations
  //instructions
  return 0;
}
```


1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

Exemple : Anatomie de `bonjour.c`

```c
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return 0 ;
}
```

Tout programme C doit contenir une fonction appelée `main`.
L'exécution commence au début de `main`.

Exemple : Anatomie de `bonjour.c`

```c
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return 0;
}
```

Par convention, la fonction `main` renvoie un code de retour :

- il est de type `int` (entier),
- la convention est de retourner 0 si tout se passe bien,
- les parenthèses de `return` sont facultatives,
- le code de retour est exploitable depuis le shell.

Exemple : Anatomie de `bonjour.c`

```c
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return 0 ;
}
```

La fonction `printf` permet d'écrire sur l'écran.

- elle fait partie de la bibliothèque C standard,
- elle doit être importée depuis l'en-tête `stdio.h`.

Exemple : Anatomie de `bonjour.c`

```c
#include <stdio.h>

int main()
{
    printf("Bonjour tout le monde!\n");
    return(0);
}
```

`printf` prend en argument une chaîne de caractères :

- tapée entre guillemets ",
- \ sert à entrer des caractères spéciaux :
  `\n` signifie "retour à la ligne".

1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

Lire une information au clavier : scanf

La procédure **scanf** est bien utile pour demander des informations à l'utilisateur.

```
int x;
printf(``donnez un entier svp !\n'');
scanf(''%d'',&x); // on passe une adresse (voir+tard)
```

Le premier argument de scanf est une **chaîne de formattage** : "%d" si on demande un entier, "%f" si on demande un flottant,...

```
int x,y;
printf(``donnez deux entiers svp !\n'');
scanf(''%d %d'',&x,&y);
```


Écrire quelque chose sur le terminal : printf

La procédure **printf** est bien utile pour imprimer des informations au clavier.

```
int x;
printf(``donnez un entier svp !\n'');
scanf(''%d %d'',&x,&y);
printf(``maintenant x=%d et y=%d",x,y);
```


1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

Exemple d'erreur

```
1 #include <stdio.h>
2
3 int main()
4 {
5    printf("Bonjour tout le monde!\n")
6    return(0);
7 }
```

Compilation : `gcc bonjour.c -Wall -o bonjour`

```
bonjour.c: In function 'main':
\alert{bonjour.c:6: error: expected ';' before 'return'}
bonjour.c:7: warning : control reaches end of ...
```

- Il manque un ;. Aucun **binaire** n'est généré.

Exemple d'avertissement

```
1  int main()
2  {
3      printf("Bonjour tout le monde!\n");
4      return(0);
5  }
```

Compilation : `gcc bonjour.c -Wall -o bonjour`

```
bonjour.c: In function 'main':
bonjour.c:3: warning: implicit declaration of function
'printf'
```

▶ Il manque un `#include <stdio.h>`.
C'est un avertissement non fatal généré par `-Wall`.

Les options `-Wall` et `-Wextra`

L'option `-Wall` attire l'attention, entres autres, sur :

- les oublis d'imports `#include`,
- les ambiguïtés syntaxiques courantes,
- les incohérences de types.

La norme est très laxiste ne considère pas ces points comme des erreurs !

`-Wextra` ajoute des avertissements supplémentaires.

▶ Toujours compiler avec `-Wall` au moins.

Espacement

L'espacement et les sauts de lignes sont libres.

```
     # include <stdio.h>
int main                        (
      ){
printf
  ("toto\n"
 );return(0)    ;}
```

Exceptions :

- `#include <stdio.h>` doit être sur une seule ligne,
- les sauts de ligne comptent dans les chaînes de caractères.

Commentaires

Commentaires : tout ce qui est entre `/*` et `*/` est ignoré.

```
#include <stdio.h> /* pour avoir printf */

/* la fonction principale
*/
int main(/* rien ici */)
{
  printf("toto\n");
  return(0); /* OK */
}
```

Conseils : - indentez votre code (TAB sous Emacs),
- commentez votre code.

1 Structure générale d'un programme
2 Exemple
3 Printf et Scanf
4 Les erreurs de compilation
5 Exercices

Exercice : programme et boucle while

Écrire un **programme** qui :

- Lit (au clavier) une suite de caractères qui finit par # et qui affiche le nombre de caractères lus différents de #
- Lit au clavier une suite de notes entre 0 et 20 et qui s'arrête lorsque l'utilisateur tape -1, puis affiche la moyenne des notes.

Exercice : Programme

Écrire un **programme** qui :

- lit 50 entiers rentrés au clavier ;
- calcule la somme de tous ces entiers en affichant la somme partielle à chaque nouveau nombre lu ;
- affiche à la fin la somme et la moyenne de ces entiers ;
- modifier le programme pour qu'il affiche la moyenne des entiers strictements positifs
- modifier ... entiers pairs

▶ On a besoin d'une **fonction** de sélection

Chapitre 3

Fonctions/Actions

3.1 Actions/fonctions : notions de base

Dans ce cours la notion-clef de fonction, utile au découpage d'un algorithme/programme, est introduite. La distinction entre **action**, *qui ne retourne pas de résultat, et* **fonction**, *qui retourne un unique résultat, est effectuée. La* **déclaration** *d'une fonction/d'une action ; ainsi que son utilisation (***appel***) sont illustrés dans les deux syntaxes introduites précédemment (pseudo-code/C). La principale difficulté du cours est la notion de* **paramètre**, *et les différentes variantes de passage de ces paramètres.*

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Quand utilise-t-on les fonctions et les actions ?
- Fonctions : usage, syntaxe de la définition d'une fonction, de l'appel.
- Actions : idem.
- Quelle est la différence entre fonction et action ?
- Paramètres données, résultats, données résultat.
- Quelle est la différence entre valeur de retour et paramètre résultat ?
- Savoir écrire une fonction ou une action simple en pseudo-code en C.
- Savoir simuler à la main l'exécution d'une fonction ou d'une action.

Algorithmique et Programmation, IMA

Cours 3 : Actions, Procédures

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Conception Structurée Descendante
2 Les Fonctions
3 Les Actions / les Procédures
4 Résumé

1 Conception Structurée Descendante
2 Les Fonctions
3 Les Actions / les Procédures
4 Résumé

Conception Structurée descendante - 1

Découper l'algorithme (action) en sous-algorithmes (sous-actions) plus simples, jusqu'à des opérations considérées primitives. Buts :

- Simplification
- Abstraction (ignorer les détails)
- Structuration
- Réutilisation

Conception Structurée descendante - 2

Exemple : sélectionner les entiers selon un certain critère

- une fonction de sélection qui dit "oui" ou "non" et qui peut être plus ou moins compliquée ;
- un appel dans le "main".

Outil : actions et fonctions paramétrées.

1 Conception Structurée Descendante
2 Les Fonctions
3 Les Actions / les Procédures
4 Résumé

Fonctions - Définition

Une fonction est un sous-programme qui à partir de **données** produit un (et un SEUL) **résultat**.

Syntaxe Algo (exemple)

Fonction *max(a,b) : entier*
| **D**: a,b : entiers {Données}
| **L**: m : entier {Variable locale}
| **Si** *a<b* **alors**
| | m ← b
| **Sinon**
| | m ← a
| **Fsi**
| **Retourner** *m* {Obligatoire}
FFonction

Fonctions - Appel de fonction

Un **appel** de fonction est une expression du **type de retour** de la fonction.

Exemple :
x ← **max(3,43)**

Que se passe-t-il lors de l'appel ?

- Les données sont remplacées par des valeurs (ou des expressions)
- Le code de la fonction est exécuté jusqu'au premier return.
- Le résultat **retourné** par la fonction est la valeur de l'expression du return.
- Ce résultat (valeur) est récupéré dans la variable x ici.

Fonctions en C - Syntaxe

Définition

```
type_de_retour nom_fonction(liste-params) {
    liste-declarations (optionnelle)
    liste_instructions
}
```

La liste d'instructions comprend **au moins** une instruction `return` (du type `type_de_retour`).

Appel (poly p 189)

```
nom_fonction(liste-expressions)
```


Fonctions en C - Exemple

Définition d'une fonction

```
int max (a:int,b:int)
{
  int m;
  if (a<b) m=a; else m=b;

  return(m);
}
```

attention au type de retour !

Appel

```
toto = max(3,45); //int declare avant !
```


Fonctions en C - Exercices

Bons entiers

- Modifier le programme de sélection des entiers inférieurs à 100 pour utiliser la fonction d'entête :

  ```
  bool bon_entier(int n)
  ```

- Écrire la fonction `bon_entier` de façon à sélectionner les entiers multiples de 3.

Fonctions en C - Erreurs courantes

- Oubli du ; dans le if :
  `error: expected ';' before 'else'`
- Oubli du return :
  `warning: control reaches end of non-void function`
- Si appel : `max(3,4)` : aucun warning ni erreur
- appel avec des arguments du mauvais type : par exemple
  `max(``toto'',4)` :
  `[...]note: expected 'int' but argument is of type 'char*`

1 Conception Structurée Descendante

2 Les Fonctions

3 Les Actions / les Procédures

4 Résumé

Les actions - définition

Une action ne retourne pas de résultat.

Action *maxproc(a,b,maxi)*
- **D**: a,b : entiers {Données}
- **R**: maxi : entier {Résultat}
- **Si** *a<b* **alors**
  - maxi ← b
- **Sinon**
  - maxi ← a
- **Fsi**

FAction

Les variables résultats servent à propager les informations produites à l'extérieur de la définition.

Les actions - Utilisation (1)

L'**appel** d'une action est une **instruction**. Les paramètres formels sont TOUS remplacés par des paramètres effectifs **de même type**

- Une donnée par une valeur (ou une expression qui a une valeur)
- Un résultat par une variable dans laquelle la procédure doit ranger le résultat
- Une Donnée/Résultat par une variable valuée.

Les actions - Utilisation (2)

Exemple
resu : entier ;
maxproc(3,43,resu);

▶ Après l'appel, la variable `resu` contient le max des deux entiers.

Les actions/procédures sont surtout utiles pour :

- **imprimer** des valeurs, des structures, des messages . . .
- **modifier** des paramètres qui ne peuvent être retournés (tableaux, paires, structures compliquées) : ceux-ci sont alors appelés *Données/Résultats* ou *entrées/sorties*.

Les actions en C - procédures

En C les actions/procédures sont des fonctions qui ne retournent rien (mot clef **void**).

```
void printmaxproc(int a, int b)
{ // impression du max
  int maxi;
  if(a<b) maxi=b; else maxi=a;
  printf("Le max est %d \n",maxi);
}
```

Paramètres R ? voir le chapitre « pointeurs »

Procédures en C - Syntaxe

Définition

```
void nom_action(liste-params) {
    liste-declarations (optionnelle)
    liste_instructions
}
```

Appel (poly p 189)

```
nom_action(liste-expressions)
```

▶ on ne récupère pas le résultat d'une procédure, il n'y en a PAS.

Procédures en C - Exemples

Écrire les procédures suivantes :

- Impression du maximum de 3 entiers en paramètres
- Impression des 100 premiers termes de la suite suivante :
  - $u_0 = 32$
  - $u_n = 3 * u_{n-1} + 19$

1 Conception Structurée Descendante
2 Les Fonctions
3 Les Actions / les Procédures
4 Résumé

Les fonctions

Une fonction retourne un **et un seul** résultat.

Fonction *ajoute_un(a) : Entier*
| **D**: a : Entier
| **Retourner** *a+1*
FFonction

Appel :

Programme *Main*
| **L**: x,y :Entiers
| x ← 12
| y ← ajoute_un(x)
| Imprimer(y)
| **Retourner** *0*
FProgramme

Les actions

Une action ne retourne pas de résultat.

Action *imprime_succ(a)*
| **D**: a : Entier
| Imprime(a+1)
FAction

Appel :

Programme *Main*
| **L**: x :Entier
| x ← 12
| imprime_succ(x)
| **Retourner** *0*
FProgramme

Et si ?

- Et si je veux modifier un paramètre d'entrée ?

Action *inc(a)*
| **D/R**: a : Entier
| a ← a+1
FAction

Programme *Main*
| **L**: x :Entier
| x ← 12
| inc(x)
| Imprime(x)
| **Retourner** *0*
FProgramme

Et si ?

- Et si je veux "retourner" deux résultats ?

Action *quotient_et_reste(a,b,q,r)*
| **D**: a,b : Entiers
| **R**: q,r : Entiers
| .....
| (calcul de q et r)
FAction

Programme *Main*
| **L**: x,y,vq,vr :Entiers
| x ← 37
| y ← 7
| quotient_et_reste(x,y,vq,vr)
| Imprime(vq,vr)
| **Retourner** *0*
FProgramme

3.2 Notions de complexité et de correction

*Pour évaluer la performance d'un programme, ou de la solution à un problème, on utilise la notion de complexité d'un programme, qui est une fonction des variables d'entrée et des constantes du programme ou de la fonction/action considérée. Dans ce mini-cours, nous abordons également une notion-clef pour "prouver" qu'un programme fait bien ce que l'on veut : la notion d'***invariant** *de boucle.*

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Définition des complexités en temps et en mémoire.
- Calcul de cette complexité sur des programmes simples, asymptotiquement.
- Définition de complexité linéaire, quadratique, exponentielle, ...
- Qu'est-ce qu'un invariant ? Donner un invariant pour une boucle donnée d'un programme simple.

Algorithmique et Programmation, IMA

Cours 3b : Notions de complexité algorithmique et de correction de programme

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Complexité Algorithmique

2 Correction

Pourquoi la complexité ?

On désire :

- estimer à l'avance la **performance** en temps/mémoire d'un programme donné ;
- estimer les limites d'utilisation d'un programme.

▶ On va évaluer le nombre d'opérations de base, d'itérations, de cases mémoires, . . .

Définition

La **complexité d'un programme** est une fonction de ses variables d'entrée :

- valeurs demandées à l'utilisateur, données par des capteurs, . . .
- constantes (taille des tableaux par exemple)

Elle mesure :

- le nombre d'opérations,
- ou d'itérations (complexité en **temps**),
- ou de cases mémoire (complexité **mémoire**) ;

Définition - 2

La complexité se calcule :

- en moyenne sur toutes les exécutions possibles du programme,
- au mieux (le minimum),
- au pire (le maximum).

et s'exprime (en général), **asymptotiquement**, c'est-à-dire comme une limite pour de grandes valeurs des paramètres d'entrées.

Exemple 1

Action *maxproc(a,b,maxi)*
  D: a,b : entiers
  R: maxi : entier
  Si *a<b* **alors**
    maxi ← b
  Sinon
    maxi ← a
  Fsi
FAction

▶ quels que soient a et b, on ne fait qu'un test. La complexité en temps/nb ops/mémoire, en moyenne, au pire, au mieux, est donc $O(1)$.

Exemple 2

Fonction *toto(n)*
  D: n : entier
  L: i,s : entiers
  s← 0
  Si *n>0* **alors**
    Pour i **de** *0* **à** *n* **Faire**
      $s \leftarrow s + i$
    Fpour
  Fsi
  Retourner *s*
FFonction

▶ La complexité est :

- au mieux 1 (si $n \leqslant 0$)
- au pire n (dans tous les autres cas)

Un peu de vocabulaire

Supposons que N soit un paramètre d'un programme/d'une fonction. Si la complexité est :

- $O(N)$, on dit que le programme est **linéaire** (au pire, en moyenne, . . .)
- $O(N^2)$: il est **quadratique**.
- $O($polynome en N$)$: **polynômial**.
- $O(2^N)$: **exponentiel**.

1 Complexité Algorithmique

2 Correction

Que veut-on garantir ?

On aimerait garantir d'un programme/une fonction satisfait ses **spécifications**, c'est-à-dire calcule le "bon résultat" quelles que soient ses paramètres (paramètres d'entrée, variables données par l'utilisateur, données de capteurs physiques, . . .).

▶ On montre la **correction** du pseudo-code et/ou de l'implémentation C à l'aide d'**invariants !**

Un exemple simple

Calcul de x^n par la méthode itérative "naïve" :

Fonction *expo(x,n) : entier*
  D: x,n : entiers
  L: i,exp : entiers
  exp ← 1
  Pour i **de** *1* **à** *n* **Faire**
    $exp \leftarrow exp * x$
  Fpour
  Retourner *exp*
FFonction

▶ Invariant "au $i$ème tour de boucle, exp contient x^i ". Prouvé par **récurrence** sur i.
▶ Conclusion ?

Et encore

D'autres exemples (complexité et calcul d'invariants) tout au cours des cours et TDs.

Remarque : on peut aussi vouloir prouver la **terminaison** d'un programme donné.

3.3 Actions/fonctions récursives

La notion de récursivité est une notion-clef en algorithmique. Une fonction **récursive** *est une fonction qui dans son code fait un appel à elle-même. Ce type de fonctions permet de réaliser des algorithmes complexes sans utiliser de boucles. Il convient néanmoins de faire attention à la terminaison du programme, en faisant en sorte que chaque appel récursif fasse décroitre strictement une certaine quantité. Au début de la fonction, un test sur cette quantité retournera directement le résultat voulu.*

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Qu'est-ce qu'une fonction récursive ?
- Savoir dérouler les appels récursifs d'une fonction.
- Savoir dire si une fonction est récursive terminale ou pas.
- Savoir transformer une procédure simple récursive en itérative et vice-versa.
- Calculer la complexité en terme de nombre d'appels récursifs.

Algorithmique et Programmation, IMA

Cours 3c - Récursivité

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

Définition

Un algorithme (une fonction, une procédure) est dit **récursif** si sa définition (son code) contient un appel à lui-même.

Un algorithme qui n'est pas récursif est dit **itératif**.

Utilisations usuelles

Utilisations variées (liste non exhaustive) :

- Calcul de suite **récursive** (numérique, graphique. . .)
- Calcul de type « diviser pour régner » : recherche, tri, . . .
- Calcul sur des structures de données **inductives** (listes, arbres, . . .) ▶ Prog avancéee (S6).

▶ Dans tous les cas, une version itérative est possible.

Quelques exemples classiques - 1

Factorielle : $n! = n.(n-1)!$

Fonction *fact(n) : entier*
  D: n : entier positif ou nul
  Si *n=0* **alors**
    Retourner *1*
  Sinon
    Retourner *n*fact(n-1)* {Appel récursif}
  Fsi
FFonction

▶ Attention au type de retour et à l'orthographe du nom de la fonction.
▶ Dérouler ! ▶ Complexité en **nb d'appels** ?

Quelques exemples classiques - 2

Fibonacci : $Fibo(n) = Fibo(n-1) + Fibo(n-2)$

Fonction *fibo(n) : entier*
- **D**: n : entier positif ou nul
- **Si** *n=0* **alors**
  - **Retourner** *1*
- **Sinon**
  - **Si** *n=1* **alors**
    - **Retourner** *1*
  - **Sinon**
    - **Retourner** *fibo(n-1)+fibo(n-2)* {Appel récursif}
  - **Fsi**
- **Fsi**

FFonction

- Dérouler ! ▶ Complexité en **nb d'appels** ?
- L'implémentation impérative est meilleure, pourquoi ?

Quelques exemples classiques - 3

Que fait cette fonction ?

Fonction *somme (n,r) :entier*
- **D**: n,r : entier positifs ou nul
- **Si** *n = 1* **alors**
  - **Retourner** *r + 1*
- **Sinon**
  - **Retourner** *somme (n -1 , r + n)*
- **Fsi**

FFonction

- r est appelé paramètre d'**accumulation**.

Récursivité terminale (ou pas ?)

Un algorithme récursif est dit récursif **terminal** si l'appel récursif est la dernière instruction réalisée.
- Stockage non nécessaire de la valeur obtenue par récursivité.

- Factorielle : $fact(n-1)$ puis multiplication par n, donc non récursif terminal.
- Somme : récursif terminal :
  $somme(5,0) = somme(4,5) = somme(...) \ldots = 15$

Dérécursivons !

Pour l'algorithme somme, la forme « accumulateur » et **l'invariant** $resu = \sum_{j=r}^{i} j$, fournit "rapidement" un algorithme itératif :

Fonction *somme2 (n)*
- **D**: n : entier positif ou nul
- r ← 0 {accumulateur}
- i ← n
- **Tq** *i⩾ 2* **faire**
  - r← r+i
  - i← i-1
- **Ftq**
- **Retourner** *r*

FFonction

- Autres exemples en TD

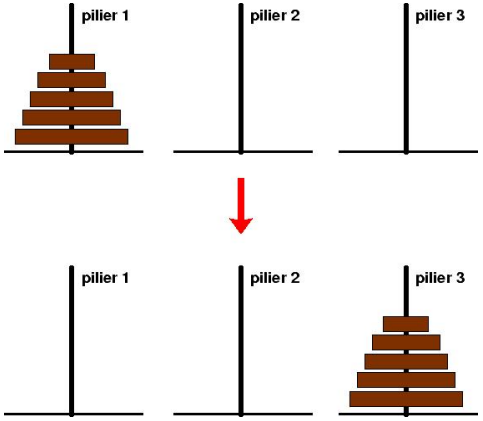
Et l'inverse ?

La procédure : **Action** *compter()*
| **L**: i :entier
| **Pour** i **de** *0* **à** *10* **Faire**
| | traitement(i)
| **Fpour**
FAction
se transforme en procédure récursive en :
Action *compter(i)*
| **D**: i :entier
| **Si** $i \leqslant 11$ **alors**
| | traitement(i)
| | compter(i+1))
| **Fsi**
FAction

Attention

Toujours bien vérifier que votre algorithme termine !

Un exemple plus exotique ! Tours de Hanoi - 1



- déplacement d'une seule rondelle à la fois
- rondelle jamais au dessus d'une plus grosse.

Un exemple plus exotique ! Tours de Hanoi - 2

- Si 0 rondelle, je sais faire !
- Si je sais faire avec $n-1$ rondelles, comment faire pour traiter le cas n rondelles ?

- Laissé en exercice !

Chapitre 4

Vecteurs/Tableaux

Lorsque l'on veut utiliser un grand nombre de variables dans un programme, ou lorsqu'on veut stocker un résultat de grande taille, on utilise une suite de cases adjacentes en mémoire, c'est-à-dire un vecteur (ou tableau, en C). Dans ce cours nous voyons comment déclarer et utiliser un **tableau statique** *en pseudo-code et en C. Des exemples classiques de tableaux d'entiers, de charactères, sont donnés. Les tableaux en deux dimensions (***matrices***) sont également abordés.*

Savoirs (liste non exhaustive) (en C et pseudo-code)

- Cas d'utilisation d'un tableau.
- Déclarer un tableau d'entiers de taille fixée à l'avance, et initialiser toutes ses cases (par exemple à 0).
- Connaître différentes façons de parcourir toutes les cases d'un tableau (avec et sans rupture prématurée de flot).
- Savoir déclarer et utiliser des matrices (tableaux 2d).
- Connaître l'encodage des chaînes de caractères sous forme de tableau avec marqueur de fin.
- Savoir concevoir des algorithmes de tableaux, de chaînes et **évaluer leur complexité**.
- Savoir utiliser la librairie `string.h`.
- Connaître la spécificité des tableaux en terme de paramètres (on ne peut retourner un tableau, on passe le tableau en paramètre modifiable, et tel quel en C).

Algorithmique et Programmation, IMA 3

Cours 4 : Vecteurs/Tableaux

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Vecteurs et Tableaux
2 Algorithmes sur les tableaux d'entiers
3 Algorithmes de mots
4 Tableaux2d - Matrices
5 Erreurs sur les tableaux - à la compilation et exécution

1 Vecteurs et Tableaux
2 Algorithmes sur les tableaux d'entiers
3 Algorithmes de mots
4 Tableaux2d - Matrices
5 Erreurs sur les tableaux - à la compilation et exécution

Vecteurs - Algo

Vecteur = suite de cases dont le contenu est de même type :

2	3	-5	-8

Déclaration (tableau de taille N **fixée**) :
v : Vecteur[N] de (type de base)
Les cases sont numérotées de 0 à $N-1$ (attention source d'erreurs !)

Accès à la case i : `v[i]`

Vecteurs - Syntaxe C

Déclaration

```
int t[23] ;          // tableau d'entiers
char chartab[900] ; // tableau de caracteres
```

Utilisation :

```
x = t[10];           // appel licite
y = chartab[1515]; // plantage a l'execution
z = t[expr compliquee];
int t[12]={0}; // declaration-init
```


Rappel : constantes

On utilisera beaucoup les constantes pour fixer la taille des tableaux.

```
#define N 10

void f(int t1[N], int t2[N]){
  int i;
  for (i=0;i<N;i++)
    t1[i] = t2[i] + 1;
}
```

▶ Pour changer la taille de tous les tableaux, il suffit de changer une seule ligne.

1 Vecteurs et Tableaux
2 Algorithmes sur les tableaux d'entiers
3 Algorithmes de mots
4 Tableaux2d - Matrices
5 Erreurs sur les tableaux - à la compilation et exécution

Accès direct

Exemple : **échange** de cases.

Action *swap(i,j,t)*
  D: i,j : entiers {Données}
  D/R: t : Vecteur[N] d'Entiers {Donnée/Résultat}
  L: tmp : entier
  tmp ← t[i];
  t[i] ← t[j];
  t[j] ← tmp;
FAction

Exercice : traduire en C et modifier pour les cas « pathologiques ».

Parcours d'un tableau - 1

Exemple : **impression** de tous les éléments.

```
Action printtab(t)
    D: t : Vecteur[N] d'Entiers                {Données}
    L: i : entier                              {Var d'itération}
    Pour i de 0 à N-1 Faire
        imprimeEntier(t[i]);
    Fpour
FAction
```

Exercice : traduire en C

Parcours d'un tableau - 2

Exemple : **copie** d'un tableau dans un autre.

```
Action copytab(t,resu)
    D: t : Vecteur[N] d'Entiers                {Donnée}
    D/R: resu : Vecteur[N] d'Entiers           {Donnée/Résultat}
    L: i : entier                              {Var d'itération}
    Pour i de 0 à N-1 Faire
        resu[i] ← t[i];
    Fpour
FAction
```

Exercice : traduire en C

Recherche dans un tableau - 1

Exemple : **recherche** du maximum.

```
Fonction maxtab(t) : entier
    D: t : Vecteur[N] d'Entiers                {Donnée}
    L: i : entier                              {Var d'itération}
    L: maxi : entier                           {Max temporaire}
    maxi = t[0];
    Pour i de 1 à N-1 Faire
        Si maxi<t[i] alors
            maxi ← t[i]
        Fsi
    Fpour
    Retourner (maxi)
FFonction
```

- **Correction** de ce programme ?

Recherche dans un tableau - 2a

Exemple : recherche d'une valeur **particulière**.

```
Fonction maxtab(val,t) : bool
    D: t : Vecteur[N] d'Entiers                {Donnée}
    D: val : entier                            {Valeur à rechercher}
    L: i : entier                              {Var d'itération}
    Pour i de 0 à N-1 Faire
        Si t[i]=val alors
            Retourner (Vrai)
        Fsi
    Fpour
    Retourner (Faux)
FFonction
```

Exercice : correction, puis traduire en C.

Recherche dans un tableau - 2b

Le même sans rupture prématurée du **flot**.

```
Fonction maxtabWhile(val,t) : bool
    D: t : Vecteur[N] d'Entiers                    {Donnée}
    D: val : entier                    {Valeur à rechercher}
    L: i : entier                          {Var d'itération}
    i← 0 ; fini← Faux
    Tq non (fini) et i<N faire
        Si t[i]=val alors
            fini← Vrai
        Fsi
        i← i+1
    Ftq
    Retourner fini
FFonction
```

Exercice : correction, puis traduire en C.

Encodages par tableaux

Les tableaux peuvent aussi servir à encoder :

- des ensembles (cf TD)
- des arbres

Retour sur les actions/fonctions

► Et si je veux retourner un tableau ?

Important Les tableaux sont des paramètres modifiables en C !

```
Action calculetab(tab)
    R: tab : Vecteur[1..1000] d'Entiers
    .....
    tab[42] = 7070
FAction

Programme Main
    L: t : Vecteur[1..1000] d'Entiers
    calculetab(t)
    Imprime(t[42])
    Retourner 0
FProgramme
```


1. Vecteurs et Tableaux
2. Algorithmes sur les tableaux d'entiers
3. Algorithmes de mots
4. Tableaux2d - Matrices
5. Erreurs sur les tableaux - à la compilation et exécution

Les chaînes de caractères

- Les chaînes de caractères sont souvent des types de base (`string` en Ocaml).
- En C, les chaînes de caractères sont des tableaux de charactères avec `\0` comme marqueur de fin de chaîne.

$'t'$	$'o'$	$'t'$	$'o'$	$'\backslash 0'$

Syntaxe C :

```
char ch[12] = {'t','o','t','o'};
char ch2[100] = "toto";
char ch3[] = "toto"; // ch3 aura 5 cases

char a = ch3[2];      // a est 't'
```


Parcours de chaîne

Exemple : nombre de 'a' dans un mot.

Fonction *nba(t) : entier*
  D: t : Vecteur[TMAX] de charactères {Donnée}
  L: i : entier {Var d'itération}
  L: nb : entier {nb de 'a' temporaire}
  nb = 0 ; i = 0 ;
  Tq *t[i]* $\neq$ *'\0'* **et** *i <TMAX* **faire**
    Si *t[i]='a'* **alors**
      nb← nb+1
    Fsi
    i ← i+1 {ne pas oublier !}
  Ftq
  Retourner *(nb)*
FFonction

▶ **Invariant de la boucle ?**

Algos de chaînes

D'autres algorithmes classiques (à voir en TD, TP, . . .)

- Un mot donné (avec sa taille) est-il un palindrôme ?
- Calculer la concaténation de deux mots ?
- Un mot est-il un sous mot d'un autre ?
- Combien de fois apparaît un mot donné dans un texte (mot plus long) ?

▶ **algorithmique du texte**

La librairie string

La librairie `string` fournit des fonctions de base sur les chaînes de caractères :

- lecture à partir de l'entrée standard
- copie
- comparaison de chaînes
- sous-chaîne, concaténation, . . .

▶ Voir à la fin du chapitre sur les pointeurs !

1. Vecteurs et Tableaux
2. Algorithmes sur les tableaux d'entiers
3. Algorithmes de mots
4. Tableaux2d - Matrices
5. Erreurs sur les tableaux - à la compilation et exécution

Matrices - Algo

Matrice = tableau 2D

2	3	-5
4	-13	42
1515	-77	0

Déclaration : matrice $N \times M$ (taille fixée)
m : Matrice[N][M] de (type de base)

Accès à la case $(i^{me} ligne, j^{me} colonne)$: `m[i][j]`

- Pour une ligne fixée, les cases sont numérotées de 0 à $M-1$ (il y a M colonnes).
- Pour une colonne fixée, les cases sont numérotées de 0 à $N-1$ (il y a N lignes).

Matrices - Syntaxe C

Déclaration

```
int t[23][42] ;          // matrices d'entiers
char chartab[900][12] ; // matrice de characteres
```

Utilisation :

```
x = t[10][10];           // appel
z = t[expr compliquee][exp2];
```

Important Les matrices sont des paramètres modifiables en C !

Ex : Impression de toutes les cases d'une matrice carrée

Action *ParcoursMat(t)*
  L: i,j : Entiers
  D: t : Matrice[N][N] d'Entiers
  Pour i **de** *0* **à** *N-1* **Faire**
    Pour j **de** *0* **à** *N-1* **Faire**
      Imprimer(t[i][j])
    Fpour
  Fpour
FAction

Ex : Impression de la diagonale d'une matrice carrée

```
Action ParcoursMat(t)
    L: i,j : Entiers
    D: t : Matrice[N][N] d'Entiers
    Pour i de 0 à N-1 Faire
        Imprimer (t[i][i]) ;
    Fpour
FAction
```


Ex : Recherche d'un élément dans une matrice rectangulaire

```
Fonction ParcoursMat(t,el) :Booleen
    L: i,j : Entiers
    D: t : Matrice[N][M] d'Entiers
    D: el : Entier
    Pour i de 0 à N-1 Faire
        Pour j de 0 à M-1 Faire
            Si (t[i][j]=el) alors
                Retourner Vrai
            Fsi
        Fpour
    Fpour
    Retourner Faux
FFonction
```

- On n'effectue pas tout le programme si on trouve l'élément.

Ex : Recherche d'un élément dans une matrice rectangulaire

Version **sans arrêt prématuré** du flot :

```
Fonction ParcoursMatWhile(t,el) :Booleen
    D: idem
    L: i,j : Entiers
    L: fini : Booléen
    fini ← Faux ; i← 0 ; j← 0
    Tq (non fini) et i<N faire
        Tq (non fini) et j<M faire
            Si (t[i][j]=el) alors
                fini ← Vrai
            Fsi
        Ftq
    Ftq
    Retourner fini
FFonction
```


1. Vecteurs et Tableaux
2. Algorithmes sur les tableaux d'entiers
3. Algorithmes de mots
4. Tableaux2d - Matrices
5. Erreurs sur les tableaux - à la compilation et exécution

Erreurs classiques :

- Tableau déclaré et pas initialisé : aucune erreur, impression du contenu courant de la case mémoire.
- Accès en dehors du tableau : pas d'erreur de compilation, `Segmentation Fault` ou valeur quelconque à l'exécution.
- Copie de tableau non case par case :

```
int t[12]={0};
int g[12];
g=t;
```

Erreur à la compilation :

```
imcompatible types when assigning to
type 'int[12]' from type 'int *'
```

Chapitre 5

Algorithmique du Tri

Premier chapitre d'algorithmique proprement dite ! Il s'agit ici de proposer des algorithmes pour trier un tableau d'entiers. Les algorithmes classiques sont ainsi vus, et leur complexité est évaluée. Ces algorithmes seront développés en TD et implémentés en C en TP.

Savoirs (liste non exhaustive) (en C et pseudo-code)
- Connaître les principes des principaux algorithmes d'entiers.
- Savoir dérouler les algos à la main sur des petits tableaux (même les algorithmes *récursifs*).
- Savoir produire le pseudo-code rapidement.
- Connaître les invariants de ces algorithmes.
- Connaître (oui, par coeur !) leur complexité. Savoir la calculer.

Remarque 1 *Le tribulle, algorithme classique mais peu efficace, n'est pas traité dans ce cours*

Algorithmique et Programmation, IMA 3

Cours 5b Algos de tri

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

1 Trions !

2 Considérations diverses sur les tris

Énoncé du problème

But

- On va trier des tableaux d'**entiers** de taille N.
- On dispose du test de comparaison entre entiers

Exemple :

11	-2	1515	42	2048	28	11	-78

devient

-78	-2	11	11	28	42	1515	2048

▶ Let's go !
▶ Attention transparents sans exemple ni dessin, donc, en faire !

Action auxillaire

On dispose de l'action auxilliaire **permuter** de signature (ou prototype) :

`permuter(T:Tableau[N] d'Entiers,ind1:Entier,ind2:Entier)`

qui permute les valeurs des éléments d'indices `ind1` et `ind2` du tableau T.

Tri Sélection

Principe :

- Je cherche le minimum du tableau et je le permute avec la case d'indice 0.
- Je cherche le minimum du tableau restant (le sous tableau $T[1..N-1]$) et je le permute avec la case indice 1
- Je cherche .....

▶ Algo, Correction, Complexité

Sélection

```
Action tri-sélection(T)
    D/R: T : Tableau[N] d'entiers
    L: ideb,i : Entiers
    L: imin : Entier  {Indice de l'élément minimum courant}
    Pour ideb de 0 à N-2 Faire
            {Recherche de l'indice de l'élément minimum}
        imin := ideb;
        Pour i de ideb + 1 à N-1 Faire
            Si T[i] < T[imin] alors
                imin := i
            Fsi
        Fpour
        permuter(T,ideb,imin)
    Fpour
FAction
```


Sélection : analyse

Correction : « L'algorithme tri-selection conserve les éléments du tableau T et les trie dans l'ordre croissant »

Invariant : « à la fin du tour $ideb$, le sous-tableau $T[0..ideb]$ contient les $ideb+1$ plus petits éléments de T, dans l'ordre croissant de leurs valeurs »

Coût (nb comparaisons) : $(N-1)+(N-2)+\ldots 1 = O(N^2)$.

Tri Insertion

«Tri des cartes à jouer» :

- Je trie les 2 premières cartes.
- Je regarde la troisième et l'insère à sa bonne place (par décalages vers la droite).
- ...

▶ Algo, Correction, Complexité

Insertion

```
Action tri-insertion(T)
    D/R: T : Tableau[N] d'entiers
    L: i : Entier
    L: élt : Entier                          {Valeur à insérer}
    L:  ins : Entier                {Indice d'insertion de élt}
    Pour i de 1 à N-1 Faire
        élt := T[i]                           {Initialisation}
        ins := i
        Tq (ins ⩾ 1 et T[ins-1] > élt) faire
            T[ins] := T[ins-1];
            ins := ins - 1
        Ftq
        T[ins] := élt                        {Insertion de T[i]}
    Fpour
FAction
```


Insertion : analyse

Correction : on prouve l'**invariant** suivant : « Au tour i, la boucle insère l'élément $T[i]$ dans le sous-tableau $T[0..i-1]$ déjà trié »

Coût : $O(N)$ au mieux, $O(N^2)$ au pire et en moyenne.
Améliorable en $N \ln_2(N)$.

Tri Fusion

Principe «diviser pour régner» :

- Un tableau de taille 1 est trié !
- Je découpe en deux le tableau
- Je trie chacun des sous-tableaux
- Je fusionne

▶ Algo récursif !, Correction, Complexité

Tri Fusion - 1

```
Action tri-fusion-bis(T,premier,dernier)
    D: premier,dernier : Entiers
    D: T : Tableau[N] d'entiers
    L: milieu : entier
    Si premier<dernier alors
        milieu ← (premier+dernier)/2
        tri-fusion-bis(T,premier,milieu)      {Appel récursif 1}
        tri-fusion-bis(T,milieu+1,dernier)        {Appel réc. 2}
        fusion(T,premier,milieu,dernier)
    Fsi
FAction
```

Tri Fusion - 2

Action *tri-fusion(T)*
| **D**: T : Tableau[N] d'entiers
| **Si** *N>1* **alors**
| | tri-fusion-bis(T,0,N-1)
| **Fsi**
FAction

- Il reste à écrire **fusion**.

Tri Fusion - 3

Dessin ! On va garder en mémoire deux curseurs, `c1` et `c2`, sur chacun des bouts de tableaux à fusionner.

Action *fusion(T,premier1,dernier1,dernier2)*
| **D**: premier1,dernier1,dernier2 : Entiers
| **D**: T : Tableau[N] d'entiers
| **L**: fus : Tableau[dernier2-premier1+1] d'entiers
| **L**: c1,c2,premier2 : Entiers
| premier2 := dernier1+1
| c2 := premier2
| c1 := premier1
| ... suite page suivante ...
FAction

Tri Fusion - 4

Parcours de fusion : le tableau local `fus` est rempli, puis recopié dans le tableau initial.

Pour *i* **de** *0* **à** *dernier2-premier1* **Faire**
| **Si** *(c1⩽dernier1* **et** *(t[c1]<t[c2]* **ou** *c2> dernier2))* **alors**
| | fus[i]← t[c1]
| | c1++
| **Sinon**
| | fus[i]← t[c2]
| | c2++
| **Fsi**
Fpour
Pour *i* **de** *0* **à** *dernier2-premier1* **Faire**
| t[premier1+i] ← fus[i]
Fpour

Fusion : analyse

On suppose que fusionne fait bien son travail
Correction : « l'appel à tri-fusion sur un tableau de taille i trie le tableau »

Coût : $O(N \ln_2 N)$ tout le temps. preuve au tableau

Tri Rapide

Principe du pivot

- Je partitionne le tableau en fonction d'un **pivot** (premier élement du tableau).
- Je trie récursivement sur chacun des tableaux à sa gauche et à sa droite.

▶ Algo, Correction, Complexité, en TD !

1 Trions !

2 Considérations diverses sur les tris

Tri de tableaux

Théorème

Un tri de tableaux d'entiers par comparaisons ne peut être réalisé en $o(n \ln_2 n)$ comparaisons en moyenne et dans le pire des cas.

▶ Un tri est alors **optimal** si il a une complexité de $\Omega(n \ln_2 n)$ en moyenne et dans le pire des cas.

Tri de tableaux - récapitulatif

On évalue le nombre de comparaisons

Algorithme	Meilleur des cas	En moyenne	Pire des cas
Tri par sélection	$O(N^2)$	$O(N^2)$	$O(N^2)$
Tri par insertion	$O(N)$	$O(N^2)$	$O(N^2)$
Tri fusion	$O(N \times \ln_2(N))$	$O(N \times \ln_2(N))$	$O(N \times \ln_2(N))$
Tri rapide	$O(N \times \ln_2(N))$	$O(N \times \ln_2(N))$	$O(N^2)$

Un tri linéaire !

Et si on connaît à l'avance les valeurs des éléments ?
Exemple : tri comptage (ou tri par casiers) :

- On crée autant de casiers que de valeurs possibles
- On compte les occurrences de ces valeurs
- On utilise pour trier

Exemple :

1	10	3	1	3

Tableau d'occurences :

2	0	2	0	3	0	0	0	0	1

et finalement :

1	1	3	3	10

Caractères stable et en place

Stable

Un algo de tri est **stable** si deux valeurs identiques restent dans le même "ordre" à la fin de l'algo.

En place

Un algo de tri est en **place** si il trie sans création d'un tableau auxiliaire.

- Les algorithmes insertion, sélection sont en place

Chapitre 6

Variables modifiables en C : les pointeurs

6.1 Notions de base sur les pointeurs

Dans ce cours nous présentons l'implémentation C des variables modifiables : les pointeurs. En effet, le langage C donne un accès aux adresses de stockage des variables, et fournit un nouveau type **adresse** *que nous pouvons manipuler comme type de base. Afin de pouvoir comprendre finement ce qui se passe lors d'un appel de fonction C, nous aborderons aussi la notion de* **schéma d'exécution**, *qui se veut une abstraction de ce qui se passe en mémoire lors de l'exécution d'un programme C.*

Savoirs (liste non exhaustive) (en C et pseudo-code)
- Quelle est la différence entre passage de paramètres par valeur et passage par adresse ?
- Savoir faire le schéma d'exécution d'un programme simple.
- Déclarer et initialiser un pointeur d'entier, de caractère...
- Utiliser les pointeurs pour passer un paramètre par adresse.
- Le pointeur `NULL`.

Remarque 2 *Attention ! La compréhension de ce cours est un prérequis au cours de S6 "Programmation avancée"*

Algorithmique et Programmation, IMA 3

Cours 6a : Variables Modifiables, Pointeurs

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

d'après A. Miné (ÉNS Ulm)

1 Schémas d'exécution - variables modifiables
2 Les pointeurs
3 Quelques exemples
4 Validité des pointeurs

Passage de paramètres par valeur

Variables D en langage algorithmique.

Fonction *ajoute_un(a) : Entier*
| **D**: a : Entier
| **Retourner** *a+1*
FFonction

Appel :

Programme *Main*
| **L**: x,y :Entiers
| x ← 12
| y ← ajoute_un(x)
| Imprimer(y)
| **Retourner** *0*
FProgramme

- Que se passe-t-il lors de l'appel de ajoute_un ?

Passage de paramètres par adresse

Variables R ou D/R en langage algorithmique.

Exemple :

Action *inc(x)*
| **D/R**: x : Entier {Donnée/Résultat}
| x ← x+1
FAction

Que fait la suite d'instructions suivante :
y : Entier ;
y ← 100 ;
inc(y) ;inc(y) ;
Que vaut y à la fin ?

- Que se passe-t-il lors de l'appel de inc ?

Modèle mémoire simplifié

Pour expliquer : **Mémoire** $\simeq$ tableau d'octets.
Chaque octet a une adresse en mémoire.

- Chaque variable **déclarée** a une adresse (début du placement mémoire).

type	int	int	char	.....	int
nom	x	y	c	.....	z
adresse	3A00	3A04	3A08	.....	3A40
valeur	10	42	'a'	.....	...

- La place en mémoire dépend du **type** de la variable (1 octet pour un char, 4 pour un int (ou 8), . . .)

Schéma d'exécution d'une fonction - 1

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Appel :

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

- Représentons la mémoire.

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur

main

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	
int	y	3A20	

main

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	

main

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	

main

type	nom	adresse	valeur
int	a	3F10	1
int	retour	3F20	

ajoute_un

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	

main

type	nom	adresse	valeur
int	a	3F10	2
int	retour	3F20	

ajoute_un

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	

main

type	nom	adresse	valeur
int	a	3F10	2
int	retour	3F20	2

ajoute_un

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	2

main

type	nom	adresse	valeur
int	a	3F10	2
int	retour	3F20	2

ajoute_un

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'une fonction - 2

type	nom	adresse	valeur
int	x	3A10	1
int	y	3A20	2

main

Fonction *ajoute_un(a) : Entier*
  D: a : Entier
  Retourner *a+1*
FFonction

Programme *Main*
  L: x,y :Entiers
  x ← 1
  y ← ajoute_un(x)
  ...
FProgramme

Schéma d'exécution d'action

Action *inc(x)*
  D/R: x : Entier
  x ← x+1
FAction

Appel :
y : Entier ;
y ← 100 ;
inc(y) ;

- Dessin !

Résumé

- un morceau de mémoire indépendant par fonction/action
- la mémoire pour les variables locales **est libérée** après la fin de l'appel de fonction.
- lors d'un passage par valeur, il y a une copie des **valeurs**
- lors d'un passage par adesse, il y a une copie des **adresses**

1 Schémas d'exécution - variables modifiables
2 Les pointeurs
3 Quelques exemples
4 Validité des pointeurs

En C

A SAVOIR :
le passage par valeur est le passage par défaut en C SAUF pour les tableaux qui sont passés par adresse.

▶ Pour passer un paramètre par adresse, on utilise **les pointeurs**

Les adresses et le C

En C, on peut :

- obtenir l'adresse d'une variable existante (`&`),
- accéder au contenu stocké à une adresse valide (`*`),
- passer des adresses en argument, les retourner, les copier (`=`),
- effectuer des opérations limitées sur les adresses (`+`, `==`, ...).

▶ C'est l'objet des transparents suivants.

L'opérateur d'adresse &

Obtenir l'adresse d'un objet en mémoire :

```
&expr
```

expr doit être une *lvalue* (*i.e.*, modifiable) existante !

- variable scalaire,
- case d'un tableau.

Exemple :

```
int i, a[2];
&i                  /* adresse de i */
printf("%p",&i)     /*impression d'adresse*/
&(a[0])    &a[0]    &a /* adresse de a[0] */
&(i+1) /* error: lvalue required as unary '&' operand */
scanf("%d",&i) ;  /*acces a l'adresse*/
```

Déréférencement *

Accéder au contenu stocké à une adresse valide (*)
Si p est une adresse valide, alors $*p$ donne le contenu de la case située à l'adresse p.

```
void inc (int* px)
{
  *px=*px+1
}

(main)
int y=100;
inc(&y);
```

- De quel type est la variable **px** ?

Les types pointeur - 1

Exemple :

```
int i=4;                    /*i est un entier*/
int* pi;                    /* p pointeur d'entier*/
pi = &i                     /* adresse de i */
```

La **variable pointeur** pi contient l'adresse de la variable entière i. On dit souvent :

"pi pointe sur i"

Contenu de la mémoire : dessin !

Les types pointeur - 2

Type d'un pointeur

- `t*` est un pointeur sur un objet de type `t`.
- si `expr` a pour type `t`, alors `&expr` a pour type `t*`.

Attention ! Le type pointé est important ! Si `t1`$\neq$`t2`, alors `t1*` et `t2*` sont incompatibles.

Les variables pointeurs

Maintenant que `t*` est un nouveau type, on peut déclarer des variables de type `t*` :

```
t* var
```

`var` peut contenir l'adresse de tout objet de type `t`.

`var` ne peut pas contenir l'adresse d'un objet de type différent !

```
int i;
float f;
int* p ; /* p << pointeur d'entier >> */
p = &i;  /* p pointe sur i */
p = &f;   /*warning: assignment from incompatible pointe
p = &(i+1);  /* erreur de syntaxe */
```

Comparaison de pointeurs

On peut comparer deux pointeurs pour :

- l'égalité == **(pointent sur la même adresse ?)**
- la différence != **(pointent sur des adresses différentes ?)**.

1 Schémas d'exécution - variables modifiables
2 Les pointeurs
3 Quelques exemples
4 Validité des pointeurs

Quelques exemples de & et *

Rappel

- *p : contenu de la mémoire à l'adresse p.
- &x : adresse de la variable x.

```
int x,y;
int* pi = &x;
*pi = 2;        /* place 2 dans x */
 pi = &y;
*pi = *pi+1;   /* incremente y */
```


Ex : Permutation

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}

(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```

▶ Réalisons le **schéma d'exécution** de ce programme.

Ex : Permutation - 2

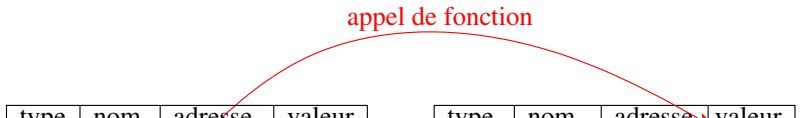
type	nom	adresse	valeur
int	a	3A10	1
int	b	3A20	3

main

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Ex : Permutation - 2



type	nom	adresse	valeur
int	a	3A10	1
int	b	3A20	3

main

type	nom	adresse	valeur
int*	px	3F10	3A10
int*	py	3F20	3A20
int	temp	3F30	

permuter

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Ex : Permutation - 2

type	nom	adresse	valeur
int	a	3A10	1
int	b	3A20	3

main

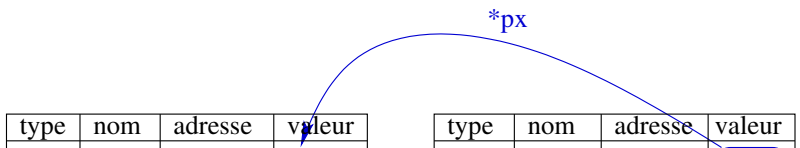
type	nom	adresse	valeur
int*	px	3F10	3A10
int*	py	3F20	3A20
int	temp	3F30	

permuter

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Ex : Permutation - 2



type	nom	adresse	valeur
int	a	3A10	1
int	b	3A20	3

main

type	nom	adresse	valeur
int*	px	3F10	3A10
int*	py	3F20	3A20
int	temp	3F30	1

permuter

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```

Ex : Permutation - 2

type	nom	adresse	valeur
int	a	3A10	1
int	b	3A20	3

main

type	nom	adresse	valeur
int*	px	3F10	3A10
int*	py	3F20	3A20
int	temp	3F30	1

permuter

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Quelques exemples

Ex : Permutation - 2

type	nom	adresse	valeur
int	a	3A10	3
int	b	3A20	1

main

type	nom	adresse	valeur
int*	px	3F10	3A10
int*	py	3F20	3A20
int	temp	3F30	1

permuter

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Quelques exemples

Ex : Permutation - 2

type	nom	adresse	valeur
int	a	3A10	3
int	b	3A20	1

main

```
void permuter (int* px,int* py)
{
  int temp;
  temp = *px;
  *px = *py;
  *py = temp;
}
```

```
(main)
int a=1,b=3;
permuter(&a,&b);
printf("%d %d",a,b);
```


Quelques exemples

Ex : Division

(paramètres R)

```
void divise(int a, int b, int* pdiv, int* prem){
  *pdiv = a / b;
  *prem = a % b;
}

void f(){
  int x, y;
  divise( 100, 10, &x, &y );
}
```

Attention ! C'est à l'appelant de fournir une adresse valide pour les valeurs de "retour".

Ex : Max de deux entiers

(passage en donnée/résultat, cf cours 3)

```
void minmaxproc(int* pa, int *pb)
{ //stocke le min dans pa, le max dans pb
  int tmp;
  if (*pa>*pb) {
    tmp=*pb ;
    *pb=*pa;
    *pa = tmp;
  }
}

a=42;b=12;
minmaxproc(&a,&b);
```


Ex : Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int* p = &x; int* q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
- `*p = *q; *p = -1;`

?

Ex : Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int* p = &x; int* q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
- `*p = *q; *p = -1;`

?

Ex : Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int* p = &x; int* q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1.` `x` est inchangé.
- `*p = *q; *p = -1;`

?

Ex : Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int* p = &x; int* q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1`. `x` est inchangé.
- `*p = *q; *p = -1;`
  `*q = y = 2` est placé dans `*p = x`,

?

Ex : Copies de pointeurs

Soit : `int x = 1, y = 2;`
`int* p = &x; int* q = &y;`

Que valent `x`, `y`, `p` et `q` après :

- `p = q; *p = -1;`
  `p` et `q` pointent sur `y`, (alias !)
  `y = -1`. `x` est inchangé.
- `*p = *q; *p = -1;`
  `*q = y = 2` est placé dans `*p = x`,
  puis `-1` est placé dans `*p = x`. `y` est inchangé.

!

Et les tableaux

En C, les tableaux sont passés par adresse :

Dans une expression, tout tableau unidimensionnel est remplacé par un pointeur vers son premier élément.

Utilité des pointeurs

Les pointeurs peuvent servir à

- passer des variables par adresse,
- « retourner » plusieurs valeurs,
- lire des données entrées au clavier **scanf**
- traverser des tableaux (non vu ici, voir TP)
- gérer des blocs de mémoire dynamique (Semestre 6).
- implémenter des listes (chaînées) (Semestre 6)
- passer des fonctions en paramètre (Semestre 6)

1 Schémas d'exécution - variables modifiables
2 Les pointeurs
3 Quelques exemples
4 Validité des pointeurs

Le pointeur NULL

NULL : valeur pointeur spéciale "vide", souvent utilisé pour dire "non défini". C'est souvent 0 mais pas toujours.

Attention Son déréférencement est impossible !

Utilisations standard :

- utilisée comme valeur pour "non définie",
- renvoyée par une fonction pour indiquer une erreur,
- passée en argument pour indiquer qu'on n'est pas intéressé par une valeur de retour.

Le pointeur NULL - ex

Ex d'utilisation :

```
#include <stdlib.h>
void divise(int a, int b, int* div, int* rem)
{
  if (div!=NULL) *div = a / b;
  if (rem!=NULL) *rem = a % b;
}
```

Note, si p est un pointeur :

- `if (p)` équivaut à `if (p!=NULL)`,
- `if (!p)` équivaut à `if (p==NULL)`.

Pointeurs valides et invalides

Attention ! Avant de déréférencer un pointeur par $*$, assurez-vous qu'il pointe vers un objet valide !
Pointeurs **valides** :

- pointeur vers une variable globale,
- pointeur vers une variable locale existante.

Pointeurs **invalides** :

- pointeur `NULL` ou non initialisé,
- pointeur en dehors des bornes d'un tableau,
- pointeur vers une variable locale détruite,
  $\Rightarrow$ ne jamais retourner un pointeur vers une variable locale !

Attention : la durée de vie d'une variable–pointeur peut dépasser celle de l'objet sur lequel elle pointe !

Exemples incorrects

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

void main() {
  int* z;
  g(z);     /*avec -Wall :
  warning: 'z' is used uninitialized in this function */
  {
    int k;
    z = &k;
    g(z);  /* equivalent a g(&k) : g modifiera k */
  }
  g(z); /*k n'existe plus, z est invalide
          mais pas d'erreur de compilation*/
}
```


Exemples incorrects

```
int* f(){
  int z = 12;
  return &z;
}

void main()
{
  int* x = f();
  *x = 13;        /* ERREUR: z n'existe plus
                     mais pas d'erreur de compilation !*/
}
```

Note : l'adresse d'une variable locale change entre deux appels d'une même fonction !

6.2 Pointeurs et tableaux et chaînes de caractères

Dans ce cours nous présentons l'utilisation des pointeurs pour parcourir des tableaux et des chaînes de caractère. Nous faisons une petite introduction à **l'arithmétique des pointeurs**.

Savoirs (en C)
- Utiliser des pointeurs pour parcourir un tableau unidimensionnel.
- Utiliser les fonctions de base de la bibliothèque `string.h`.

Algorithmique et Programmation, IMA 3

Cours 6b : Un peu plus sur les pointeurs

Laure Gonnord
http://laure.gonnord.org/pro/teaching/
Laure.Gonnord@polytech-lille.fr

Université Lille 1 - Polytech Lille

d'après A. Miné (ÉNS Ulm)

1 Pointeurs et tableaux
2 Le cas particulier des chaînes de caractères
3 Et encore ...

Les tableaux

Rappel : en C, les tableaux sont passés par adresse :

Dans une expression, tout tableau unidimensionnel est remplacé par un pointeur vers son premier élément.

Arithmétique de pointeurs

Si `p` pointe sur une case d'un tableau :

- `p+i` ou `i+p` pointe i cases après `p`
- `p-i` pointe i cases avant `p`

  (ajouter `i` $\simeq$ se déplacer de `i` $\times$ `sizeof(*p)` octets...)

▶ **Les raccourcis** `+=`, `-=`, `++`, `--` marchent également.

Attention, pour faire cela, il faut :

- Déplacer un pointeur sur un tableau valide (déclaré, alloué) de taille >1.
- Ne pas dépasser la taille du tableau

Comparaison de pointeurs/tableaux

Si `p` et `q` pointent dans le même tableau, on peut :

- comparer les indices des cases : `p < q`, `p <= q`, etc.
- calculer la distance en cases : `p - q`.

Pointeurs et tableaux unidimensionnels

Dans une expression, tout tableau unidimensionnel est remplacé par un pointeur vers son premier élément.

Équivalences

`tab`	$\simeq$	`&tab[0]`
`tab+i`	$\simeq$	`&tab[i]`
`*tab`	$\simeq$	`tab[0]`
`*(tab+i)`	$\simeq$	`tab[i]`
`i[tab]`	$\simeq$	`tab[i]` **(!)**

Exception : `sizeof(tab)` renvoie la taille du type de `tab`. (attention si `tab` est un argument !)

Tableaux multidimensionnels : c'est plus complexe et moins utilisé.

Pointeurs et tableaux unidimensionnels - Ex 1

Parcours d'un **tableau d'entiers** de taille N :

```
void imprimeTout(int t[N]){
  int i;
  for (i=0;i<N;i++)
    printf("%d, ",t[i]);
}
```

Avec un pointeur `pi` ($t = \&t[0]$)

```
void imprimeToutouPas(int* t,int N){
  int* pi;
  for (pi=t;pi<&t[N];pi++)
    printf("%d, ",*pi);
}
int a[100];
monparcours(a,30);
```

- Attention, on n'a jamais vérifié que l'accès est valide !

Pointeurs et tableaux unidimensionnels - Ex 2

Parcours d'une **chaîne** de caractères :

```
void monparcours(char s[N]){
  int i=0;
  while (i<N && t[i] != '\0')    {
    printf("%c, ",t[i]);
    i++;}
}
```

Avec un pointeur :

```
void monparcours(char* s)
{
  char* ch= s; // pointeur
  while (*ch != '\0'){
    printf("%c, ",*ch);
    ch++;}
}
```

- Pas "besoin" de la taille de la chaîne ...

Pointeurs et tableaux unidimensionnels - Ex 3 - 1/2

La **recherche** dans un sous-tableau ($imin + nb < N$) :

```
bool cherche_zero(int tab[N], int imin, int nb) {
  for (int i=imin; i<=(imin+nb); i++) {
    if ( tab[i]  == 0 ) return true;
  }
  return false;
}

void f()
{
  int a[100];
  if (cherche_zero(a,10,5)) ...
}
```


Pointeurs et tableaux unidimensionnels - Ex 3 - 2/2

Avec l'utilisation de l'adresse du premier élément du sous-tableau dans lequel on recherche :

```
bool cherche_zero(int* tab, int nb)
{
  for (; nb>0; nb--, tab++ )
    if ( *tab  == 0 ) return true;
  return false;
}

void f() {
   int a[100];
   if (cherche_zero(&a[10],5)) ...
}
```

Avantage : remplace un couple tableau + indice.

Limitations des tableaux statiques

Du fait qu'un tableau est un pointeur constant :

- On ne peut créer de tableau dont la taille est une variable du programme
- On ne peut créer de tableau bidimensionnel dont les lignes n'ont pas le même nombre d'éléments.

▶ Ces opérations deviennent possibles dès que l'on manipule des pointeurs alloués dynamiquement (**malloc**, Semestre 6).

1 Pointeurs et tableaux

2 Le cas particulier des chaînes de caractères

3 Et encore ...

Chaîne de caractères

Maintenant nous savons qu'une chaîne de caractères (tableau) est en fait un `char*` :

```
int  main()
{
  int i;
  char *chaine;
  chaine = "chaine de caracteres";
  for (i = 0; *chaine != '\0'; i++)
    chaine++;
  printf("nombre de caracteres = %d\n",i);
}
```


Bibliothèque string

Il est possible d'utiliser les fonctions de la bibliothèque `string.h` :

```
man 3 fgets RET
```

```
char *fgets(char *s, int size, FILE *stream);
```

```
fgets() reads in at most one less than size characters from stream  and
stores  them  into  the buffer pointed to by s.  Reading stops after an
EOF or a newline.  If a newline is read, it is stored into the  buffer.
A '\0' is stored after the last character in the buffer.
[...]
fgets() return s on success, and NULL on error or  when  end
of file occurs while no characters have been read.
```


Lire une chaîne sur l'entrée standard

```
#define N 1024

char* monfgets(int size)
{
  assert(size<N);
  char input[N]; // preparation du buffer
  fgets(input,size+2,stdin); //get !
  char *resu = strndup(input,strlen(input)-1);
// enleve le saut de ligne
  return resu;
}
char *resu = monfgets(4); //4premiers chars !
printf("%s\n",resu);
```


Fonctions utiles

La librairie `string` fournit entre autres :

- `strlen(s)` retourne la taille d'une chaîne :

  ```
  calculates  the  length  of  the string s, not
  including the terminating '\0' character.
  ```

- `strncmp(ch1,ch2)` compare deux chaînes :

  ```
  compares the two strings s1 and s2.  It returns
  an integer less than, equal to, or greater than
  zero if  s1  is  found, respectively, to be less
  than, to match, or be greater than s2.
  ```

1 Pointeurs et tableaux

2 Le cas particulier des chaînes de caractères

3 Et encore ...

Priorité des opérateurs

Du plus prioritaire au moins prioritaire.

`[]`	accès dans un tableau
`++ --`	incrémentation et décrémentation
`*`	déréférencement de pointeur
`&`	prise d'adresse
`* / %`	opérateurs multiplicatifs
`+ -`	opérateurs additifs
`== < > ...`	opérateurs de comparaison
`&& ||`	opérateurs booléens
`= op =`	opérateurs d'affectation

Exemple : `*p++` signifie `*(p++)`, pas `(*p)++;`

- dans le doute : mettre des parenthèses.

Priorité dans les déclarations

Attention à la priorité de `*` et `,` dans les déclarations.

- `int *a,b;`
  `b` a pour type `int`, pas `int*`.
- `int *a,*b;`
  `a` et `b` ont le type `int*`.

Pointeurs complexes

Exemples complexes :

- `int** x;` pointeur sur un pointeur sur un `int`,
  `*x` : pointeur sur un `int`,
  `**x` : `int`.
- `int *x[10];` tableau de 10 pointeurs sur des `int`,
  `x[1]` : pointeur sur un `int`,
  `*(x[1])` : `int`.
- `int (*x)[10];` pointeur sur un tableau de 10 `int`.
  (inutile : on préférera un pointeur sur un élément du tableau)

Autres documents utiles

Cette section comprend :
- les objectifs du cours : algorithmique et C
- un poly récapitulatif de syntaxe algorithmique
- un glossaire Algo/Chinois

POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S5) **Programmation Structurée 2011/2012**

http://laure.gonnord.org/pro/teaching/

Objectifs du Cours

1 Compétences attendues en Algorithmique

- Connaître la syntaxe du pseudo langage algorithmique
- Connaître les notions suivantes : les variables, les constantes, les fonctions, les actions, les boucles tant que, les boucles pour.
- Savoir écrire précisément la déclaration d'une action et l'appel de cette action (types, syntaxe, ...). Pareil pour une fonction. Savoir décrire une exécution à l'aide de schémas d'exécution. Savoir ce qu'est la signature d'une fonction.
- Connaître les principes des tris courants (insertion, sélection, bulle, fusion et rapide). Savoir retrouver les algorithmes en pseudocode. Savoir très rapidement écrire le tri sélection.
- Les tableaux : déclaration, initialisation, savoir parcourir un tableau avec une boucle *pour*, savoir parcourir avec une boucle *tant que* lorsque l'on veut s'arrêter avant.
- Les matrices ou tableaux 2d : idem
- Les chaînes de caractères codées avec marqueur de fin : parcours, et les algorithmes courants (longueur, concaténation, ...)
- Les variables modifiables : utilisation à bon escient, et déclaration à l'aide des variables D et/ou R.
- Les structures : déclaration statique.
- Conception/Analyse : savoir analyser un problème et le découper en petits algorithmes. Savoir choisir entre fonction et action, et déterminer les variables d'entrée nécessaires.
- Savoir analyser son algorithme en terme de coût. Connaître le coût en moyenne des principaux tris.

2 Compétences attendues en Programmation C

- La même chose que la partie algorithmique : variables, types, constantes, fonctions, procédures, appels de fonctions, tableaux et matrices, structures...
- Booléens : savoir qu'ils n'existent pas en C en tant que tels, mais que l'on peut (et l'on doit) utiliser `stdbool`.
- Pointeurs : utiliser * et & à bon escient. Utilisation des pointeurs dans le cas où l'on utilise des paramètres modifiables en algorithmique.
- Syntaxe des tableaux et matrices : déclaration, accès à une case, parcours dans tous les sens.
- Les chaînes de caractères en C. La différence entre "a" et 'a'.
- Entrées/sorties : printf, scanf, et utilisation pour demander des informations à l'utilisateur du programme. Utilisation de getc et getline.
- Principes généraux de compilation pratique : ce que sont les .o, .h (savoir faire un .h), les librairies, et comment compiler à la ligne de commande ou avec un Makefile. Différence entre compilation et exécution.
- Utiliser les fonctions données dans les librairies, par exemple dans `string.h`.

3 Compétences TP

et plus spécifiquement en ce qui concerne le C :

- Utiliser un éditeur efficace pour éditer du C et éventuellement compiler
- Compiler, exécuter à la ligne de commande.
- Savoir lire l'énoncé, répondre aux questions posées, papier, crayon, expliquer, faire des tests pertinents.
- Commenter, documenter.
- Savoir un peu utiliser gdb.

POLYTECH LILLE

Université Lille1 Sciences et Technologies

Département IMA / 3A (S5) **Programmation Structurée 2011/2012**

http://laure.gonnord.org/pro/teaching/

Syntaxe Algorithmique

Nom	Syntaxe	Exemple	Commentaire
Affectation	$\leftarrow$	$x \leftarrow 42$	x doit être déclaré
Type entier	Entier	x :Entier	déclaration de x entier
Type réel	Réel	x :Réel	déclaration d'un réel, en machine ce sera un flottant
Type charactère	Charactère	c :Charactère	déclaration d'un caractère ; les constantes sont 'a', 'b', ...
Type booléen	Booléen	b :Booléen	déclaration d'un booléen ; les constantes sont `Vrai` et `Faux`
Type chaîne	Chaîne	$s \leftarrow$ "*toto*"	affectation d'une chaîne, s doit être déclarée.
Tableau	Vecteur	Vecteur[10] d'Entiers	tableau de 10 entiers indexés de 0 à 9
constante	Constante	Constante N : 10	déclare une constante symbolique N qui vaut "10"

Tests

Si *condition* **alors**
| instructions si vrai
Sinon
| instructions si faux
Fsi

Si *condition* **alors**
| instructions si vrai
Fsi

Boucles

Pour *i* **de** *inf* **à** *sup* **Faire**
| instructions
Fpour

Tq *condition* **faire**
| instructions
Ftq

Programme/fonction/action

Programme *Main*
| ....
| ....
| ....
| **Retourner** *0*
FProgramme

Fonction *fonct(c) : Entier*
| **D**: c :Charactère
| **L**: s :Entier
| ....
| ....
| ....
| **Retourner** *s*
FFonction

Action *monact(a,b,c)*
| **D**: a : Entier
| **D/R**: b : Entier
| **R**: c : Charactère
| ...
| ...
| ...
FAction

- *fonct* est une fonction `Caractère -> Entier` :
  L'unique paramètre (la *donnée*) est un caractère, nommé c dans la suite de la fonction. La variable s est *locale*. L'appel : $resu \leftarrow fonct(d)$ où d a une valeur et $resu$ est déclaré de bon type.
- *monact* est une action à deux arguments. Les modifications apportées au premier paramètre ne sont pas enregistrées (c'est une *donnée*). Par contre les modifications apportées aux paramètres 2 et 3 sont enregistrées (*donnée-résultat* et *résultat*). Appel : `monact(a,b,c)` avec a, b ayant une valeur, et c étant déclaré.

Termes informatiques en chinois --- Programmation Structurée --
Glossaire réalisé avec l'aimable collaboration de Quanquan WANG, IMA3 en 2010/2011

	Exemple	Traduction
algorithme		算法
implémentation (en C)		植入，实现
variable		变量
type		类型
booléen	true,false	布尔函数（真，假）
entier	12,-7	整数
réel	42.67	实数
caractère	a'	字符
chaîne de caractères	“toto”	字符串
expression		表达式
expression numérique	2x+56	数字表达式
expression booléenne	b ou true	布尔函数表达式
affectation	X := 7	赋值
calculer		计算
stocker		储存
crochet	[，]	方括号
accolade	{ , }	大括号
chevron	<, >	角括号
action / procédure		程序
fonction		函数
retourner		返回
paramètre		参数
paramètre d'entrée		输入的参数

paramètre de sortie		输出的参数
appel de fonction		函数的调用
passage de paramètres par valeur		按值传递参数
vecteur, tableau	[1|2|-2| …]	向量，数组
indice		下标，标号
case		案例，个案
accès en lecture		可读
accès en écriture		可写
pointeur		指针
mémoire		内存
Allocation / allouer		配置，给予（开辟空间）
adresse		地址
complexité, coût d'un programme		复杂程度，程序的成本
algorithme linéaire		线性算法
algorithme quadratique		二次算法
algorithme polynômial		多项式算法
algorithme exponentiel		指数算法
déboguer		调试
tester		测试
compiler		编译
éditer un fichier		编辑文件