Polycopié de CS101

— Version 2022/2023 —

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 poly n'hésitez pas à me soumettre toute critique, suggestion, remarque ou correction, dans mon casier ou, électroniquement, à l'adresse

`Laure.Gonnord@esisar.grenoble-inp.fr`

Table des matières

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 `clang` le fichier `toto.c` ?

Remarque 1 *Ce polycopié est un* remake *du polycopié de 3A Polytech, de la même autrice, de 2012-2013. Le pseudo-code est supprimé petit à petit, à part pour la partie algorithmique du tri.*

Algorithmique et Programmation CS101

#1 : Introduction et Hello World

Laure Gonnord

Grenoble INP/Esisar

2022-23

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

CS101 : Algo et Programmation : objectifs

- **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 : **invariants**.
- Réflexions sur la pertinence des solutions et leur coût d'exécution : **complexité**.

Chamilo, mails

- Ces transparents ainsi que toutes les ressources du cours sont/seront sur Chamilo. http://chamilo.grenoble-inp.fr
- Mails : Mettez svp [CS101] dans vos objets, avec une description.

Placement dans les enseignements du premier cycle

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

mais aussi analyse numérique, électronique, automatique !

Une remarque importante

L'objectif n'est PAS de connaître la syntaxe du langage C par coeur.

Premier programme

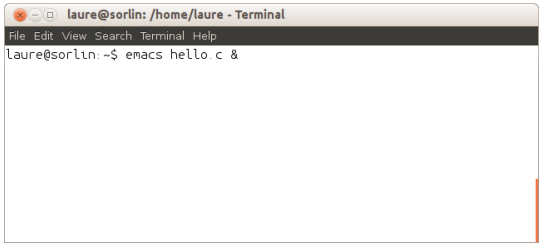
```c
#include <stdio.h>

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

Effet :

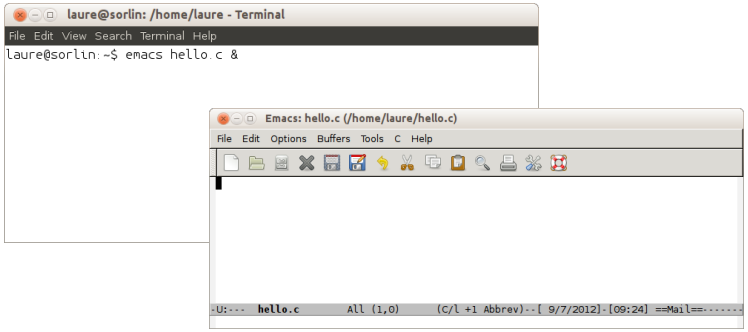
- affiche `Bonjour tout le monde!`,
- retourne le code 0 (tout s'est bien passé).

Édition, compilation et exécution



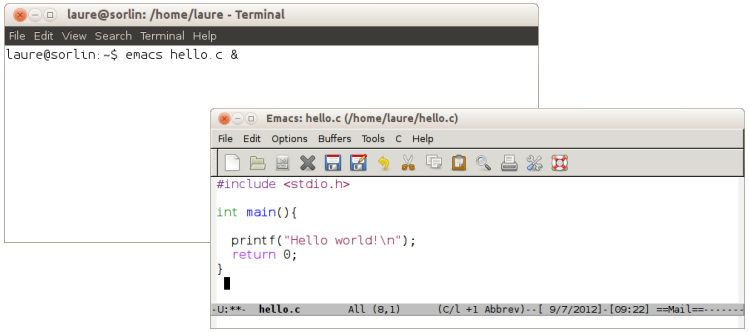
Lancement de l'éditeur en tâche de fond (&).

Édition, compilation et exécution



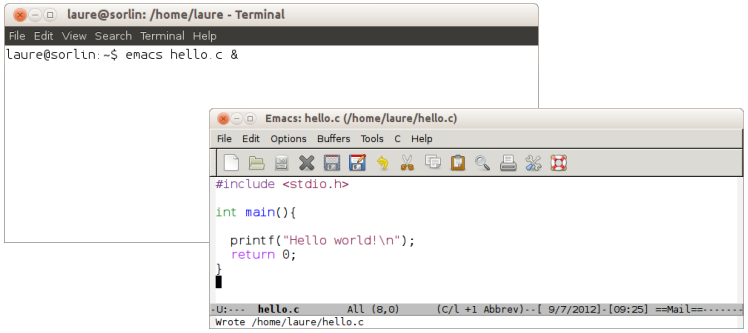
Lancement de l'éditeur en tâche de fond (&).

Édition, compilation et exécution



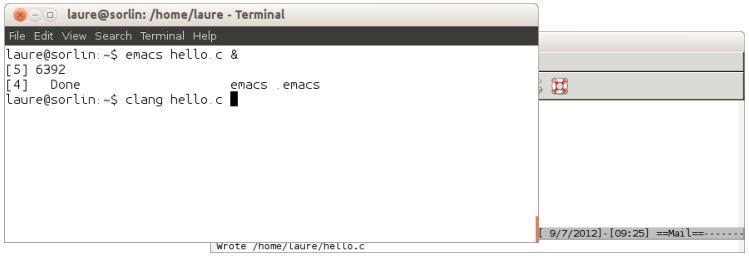
On tape le texte du programme : **édition.**

Édition, compilation et exécution



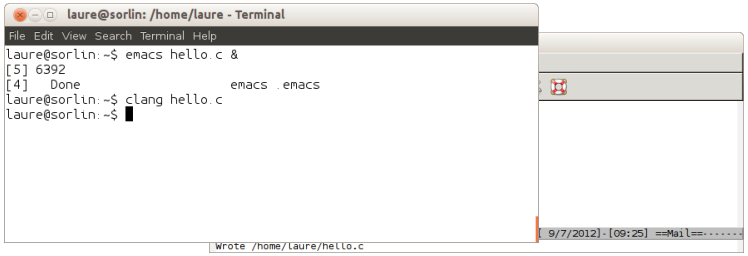
Il ne faut pas oublier de sauvegarder.

Édition, compilation et exécution



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

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

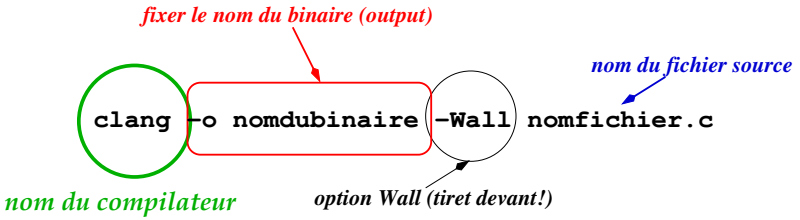
Le programme s'**exécute** et rend la main.

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 : `clang hello.c -o hello` et ensuite l'exécuter avec :`./hello`

- `clang` est un **compilateur**. On peut aussi utiliser `gcc` (plus courant)

Ligne de compilation à connaître



ou encore (l'ordre est indifférent) :

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

`clang` peut être remplacé par `gcc` (autre compilateur).

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. Le langage C sert de support

Savoirs (liste non exhaustive) (en C)

— 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).

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 CS101

#2 C Éléments de syntaxe & Programmes

Laure Gonnord

Grenoble INP/Esisar

Note importante

Je n'utilise pas de pseudo-code.

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
```


Les variables et les types, pourquoi ?

On veut stocker des informations qui ont un **nom** :

- un entier x pour pouvoir exprimer la fonction $x \mapsto x + 1$
- une chaîne de caractères s qui contient le prénom de l'utilisateur

x, s peuvent prendre des **valeurs** différentes dans un programme donné, ce seront donc des **variables**.

De plus, on veut qu'il soit interdit de stocker autre chose qu'un entier pour x, autre chose qu'une chaîne pour s, on va donc leur **donner un type**.

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 en C

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


Type booléen

Caractéristiques :

- N'existe pas nativement 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 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 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 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.

Expressions, pourquoi ?

On veut pouvoir effectuer des opérations avec les variables d'un programme, par exemple :

- Sommer des variables entières
- Tester si une variable entière est plus petite qu'une autre

▶ Les opérations numériques seront des **expressions numériques**, les opérations de tests seront des **expressions booléennes**.

Expression numérique, expression booléenne

Expression **numérique** (C) :

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

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, pourquoi ?

On veut **stocker** des valeurs numériques dans des variables entières, des valeurs booléennes dans des variables booléennes, ...

▶ Cette opération est appelée **affectation**.

Expression-affectation

En **C** :

```
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 .....

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 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 */
```


Notion d’instruction

Une **instruction** est une ligne de 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)
```

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=4; // déclaration et affectation
  z=42+x;
}
{
  x=2; //erreur, x non declare dans le bloc
}
```


Conditionnelle en C - "if then else"

Aussi appelé "test", mais le terme est ambigu :

```
  if <expression> action1 else action2 // il n'y a pas de then!
  if <expression> action1
```

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

```
if (2x+5<=b) printf("blabla"); // 2Â° branche optionnelle
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
```


Conditionnelle - 2

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

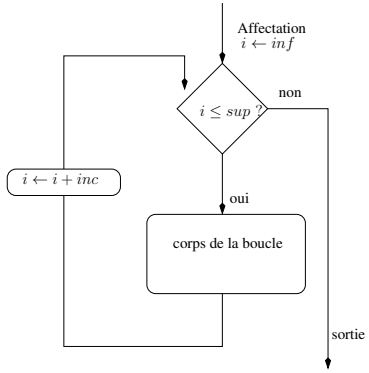
Instruction d'itération : POUR - 1 (version simple)

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" (for) est en fait plus générale/complexe en C.

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

En C :

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

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

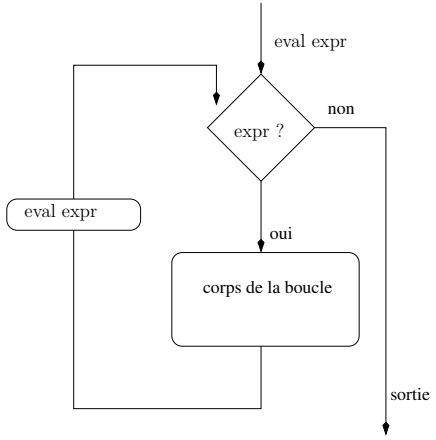
- Si la condition est initialement fausse, le bloc n'est jamais exécuté.
- La condition est retestée après chaque tour de boucle.
- Les parenthèses autour de la condition sont **obligatoires**.
- Si une seule instruction : { et } facultatifs.

Instruction d'itération : TANTQUE - Exemples

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

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


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

Pour le folklore :

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


Syntaxe générale d'un programme

```
programme   ::=   liste_declarations (option)
                  liste_defs_fonctions (option)
                  definition_main
```

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

Syntaxe générale du main - C

Le main est un cas particulier de **fonction** :

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


Exemple : Anatomie de `bonjour.c`

```
#include <stdio.h>

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

Par convention, la fonction `main` renvoie un code de retour (au shell) :

- la valeur du retour est de type `int` (entier),
- la convention est de retourner 0 si tout se passe bien,
- les parenthèses de `return` sont facultatives

Exemple : Anatomie de `bonjour.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`

```
#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".

Lire une information au clavier : scanf

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

```c
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,...

```c
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.

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

Qu'est-ce que c'est ?

Lorsque le fichier source n'est pas **correct**, le compilateur (clang, gcc) génère des erreurs de compilation.

Remarque : les schémas d'erreurs sont différents selon les compilateurs. Certains compilateurs récents (clang) ont des messages plus explicites.

Exemple d'erreur

```c
#include <stdio.h>

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

Compilation : `clang hello.c -Wall -o bonjour`

```
hello.c:5:27: error: expected ';' after expression
  printf("Hello world!\n")
                          ^
```


Exemple d'avertissement

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

Compilation : `clang hello.c -o bonjour`

```
hello.c:3:3: warning: implicitly declaring library function 'printf'
      with type 'int (const char *, ...)'
  printf("Hello world!\n");
  ^
hello.c:3:3: note: please include the header <stdio.h> or explicitly
      provide a declaration for 'printf'
1 warning generated.
```

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.

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 ;

Variantes :

- modifier le programme pour qu'il affiche la moyenne des entiers strictement positifs
- modifier ... entiers pairs

Chapitre 3

Fonctions et procédures

Les deux sections de ce cours font l'objet du même jeu de transparents.

3.1 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/procédure**, *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 en C.*

Savoirs (liste non exhaustive) (en C)

- Quand utilise-t-on les fonctions et les actions ?
- Fonctions : usage, syntaxe de la définition d'une fonction, de l'appel. Les procédures sont des cas particulier de fonctions sans valeur de retour.
- Une fonction ne retourne qu'un seul résultat dont on précise le type.
- Savoir écrire une fonction ou une procédure simple en C.
- Savoir simuler à la main l'exécution d'une fonction ou d'une procédure.

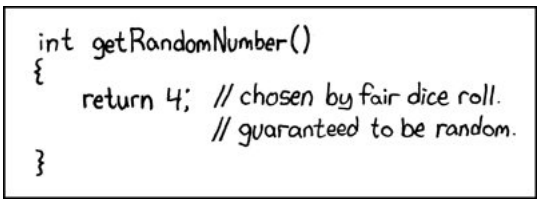


FIGURE 3.1 – `http://xkcd.com/221/`, sous License Creative Commons

3.2 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, une conditionnelle sur cette quantité retournera un résultat dit "de base".*

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

— Qu'est-ce qu'une fonction récursive ?

— Savoir dérouler les appels récursifs d'une fonction.

— Calculer la complexité en terme de nombre d'appels récursifs.

Algorithmique et Programmation CS101

Cours 3 : Fonctions (et procédures)

Laure Gonnord

Grenoble INP/Esisar

2022-23

Pourquoi les fonctions ?

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

Fonctions - Définition

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

```
int max(int a, int b){
 int m;
 if (a<b) m=a; else m=b;
 return m; // le maximum des deux entiers (paramètres) a,b
}
```

Remarquer :

- le type de retour, et le mot clé **return**
- les paramètres a,b

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 - Exercices

- Écrire une fonction d'entête :

  ```
  int puissance(int x, int n)
  ```

  qui calcule x^n par multiplications successives par x.
- Écrire une fonction qui retourne true ou false suivant si son paramètre est un multiple de 3 ou non.

▶ Toujours se poser la question des **types d'entrée, types de retour**

Fonctions en C - Erreurs courantes

- Oubli du return :

```
toto.c:11:1: warning: control reaches end of non-void
 function [-Wreturn-type]
}
^
```

- appel avec des arguments du mauvais type : par exemple `max("tsoin",4)` :

```
toto.c:42:18: warning: incompatible pointer to integer conversion passing
      'char [6]' to parameter of type 'int' [-Wint-conversion]
  int toto = max("tsoin",4);
                 ^~~~~~~
toto.c:6:15: note: passing argument to parameter 'a' here
int max ( int a , int b )
```

Les procédures

En C les 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);
}
```

Il n'y a donc pas de résultat d'une procédure.

Quel usage ?

Les 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 passés par adresse (voir plus tard)

Procédures en C - Exemples

Écrire les procédures suivantes :

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

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, ...) ▶ plus tard

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

Quelques exemples classiques - 1

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

```
int facto(int n){
  if (n==0) return 1;
  else return n*facto(n-1);
}
```

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

```
int fibo(int n){
  if (n==0) return 1;
  else if (n==1) return 1;
  else return fibo(n-1)+fibo(n-2);
}
```

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

Quelques exemples classiques - 3

Que calcule somme(5,0) ?

```
int somme(int n, int r){
 if (n==1) return r+1;
 else return(somme(n-1,r+n));
}
```

- 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 : $somme(5,0) = somme(4,5) = somme(...) \ldots = 15$

Récursif, itératif ?

```
int facto(int n){
 if (n==0) return 1;
 else return n*facto(n-1);
}
```

et

```
int facto(int n){
 int resu=1;
 for ...

}
```


Attention

Toujours bien vérifier que vous avez les bons cas de base de récursion pour terminer.

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).

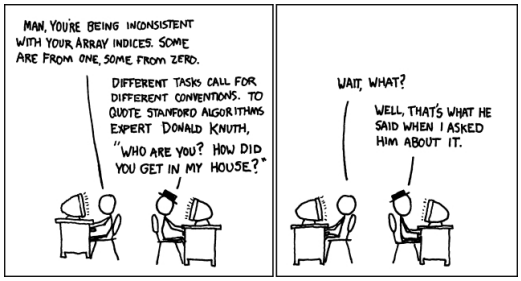


FIGURE 4.1 – `http://xkcd.com/163/`, sous License Creative Commons

Algorithmique et Programmation CS101

Cours 4 : Tableaux

Laure Gonnord

Grenoble INP/Esisar

2022-23

Abstraction de la mémoire

On considère la mémoire linéaire, et de manière abstraite, un tableau est une suite de cases dont le contenu est de même type :

2	3	−5	−8

Le type de la case (**cellule**) permet de préciser quelle place en mémoire elle prend (1, 4, 8 octets).

Tableaux (statiques) - Syntaxe C

Déclaration

```
int t[23] ; // tableau de 23 entiers (indices de 0 à 22)
char chartab[900] ; // tableau de 900 caractères
```

Utilisation :

```
x = t[10]; // lecture licite (car 0 <= 10 <= 22)
chartab[3]='d'; // écriture licite
y = chartab[1515]; // erreur à l'exécution
z = t[expr compliquee]; // l'expression doit s'évaluer en un
    entier
int t[12]={0}; // déclaration-initialisation à constante
```


Tableaux en C

Les tableaux que nous utilisons ici ont une taille fixée à l'avance (à la déclaration), ce sont des **tableaux statiques**.

- Comment écrire des fonctions qui fonctionnent pour des tableaux de taille quelconque ? Deux possibilités :
  - Utiliser des constantes symboliques à la déclaration des tableaux et dans l'écriture des fonctions :
  - Utiliser des fonctions dans lesquelles la taille du tableau est un nouveau paramètre.

Passage de tableau statique en paramètre

Première façon :

```
#define N 10
void f1 (int t[N]) { // 1 paramètre: t, tableau d'entiers
  // utilisation de t statique de taille N
}
```

- Remarquez la syntaxe qui ressemble à la déclaration !

```
void f2(int t1[], int size){ // t1 est passé en paramètre
    // utilisation de t1 supposé de taille size (rien ne le dit)
  }
```

- ici on passe aussi la taille.

Fonctions : valeurs de retour tableaux ?

Important Les tableaux sont des paramètres modifiables en C (ils sont passés par adresse). L'explication viendra quand on fera les pointeurs.

```
void init_tab(int t[], int size
    , int cte){
  // chaque t[i] initialisé à
     cte
  // ( code coupé )
  }
```

```
int main(){
  int t1[4]
  init_tab(t, 4, 12);
  printf("t[%d]=%d \n",
        2, t[2]);
```

- Après l'appel `init_tab(tab,12)` le tableau tab est modifié.

Accès direct

Échange de valeurs

Écrire une suite d'instructions qui réalise l'échange de deux valeurs entières (contenues dans 2 variables.

Échange de valeurs dans un tableau

Écrire une suite d'instructions qui réalise l'échange de deux valeurs contenues dans un tableau, à l'aide de leurs indices.

Échange, en C, avec constante symbolique (V1)

```c
#define N 12
void swap(int t[N],int i,int j){
  int tmp;
  if ( i >=0 && i < N
    && j>=0 && j<N){
    tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
  }
 }
```

```c
int main(){
  int t[N];
  // ...
  swap(t,2,3);
}
```


Échange en C, V2, en passant la taille en paramètre

```c
void swap(int t[], int i,
    int j, int size){
  int tmp;
  if (i >=0 && i < size && j>=0
      && j<size){
    tmp = t[i];
    t[i] = t[j];
    t[j] = tmp;
  }
 }
```

```c
int main(){
  int size = 12;
  int t[size];
  // ...
  swap(t,2,3,size);
}
```

Parcours d'un tableau - 1 (impression) Code C

Énoncé

Écrire une fonction qui imprime sur le terminal chacun des éléments d'un tableau d'entiers passé en paramètre.

Prendre des notes

Parcours d'un tableau - 2 (copie) Code C

Énoncé

Écrire une fonction qui prend deux tableaux d'entiers de taille N en paramètres et qui copie le contenu complet du deuxième dans le premier.

Prendre des notes

Attention `t1 = t2;` ne donne pas ce que l'on veut.

Parcours de tableau - recherche

Énoncé

Écrire une fonction qui prend un tableau d'entiers en paramètre et qui retourne le maximum contenu dans les cellules du tableau.

Prendre des notes - à finir pour le prochain cours.

Recherche dans un tableau d'une valeur particulière

Énoncé

Écrire une fonction qui prend un tableau d'entiers en paramètres, ainsi qu'un entier, et qui retourne true si l'élément existe dans le tableau, false sinon.

On va faire deux versions :

- Algorithme **avec rupture prématurée du flot**.
- Le même sans rupture prématurée du **flot**.

Encodages par tableaux

Les tableaux peuvent aussi servir à encoder :

- des ensembles.
- des arbres.

Les chaînes de caractères

- Dans les langages de programmation, les chaînes de caractères sont souvent des types de base (`string` en OCaml).
- En C, les chaînes de caractères sont des tableaux de caractères avec `\0` comme marqueur de fin de chaîne.

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

Syntaxe C :

```
char ch2[100] = "toto";//'\0' ajouté
char ch3[] = "toto"; // ch3 aura 5 cases

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


Parcours de chaîne

Taille de chaîne

Écrire une fonction qui prend une chaîne de caractères sous forme d'un tableau statique avec une taille fixée, et qui renvoie la taille de la chaîne (ie la position de `\0`).

Nombre d'occurrences d'un caractère particulier

Écrire une fonction qui prend en paramètre une chaîne de caractères dans un tableau statique, et un caractère particulier, et qui renvoie le nombre d'occurences de ce caractère dans la chaîne.

Algos de chaînes

Palindrôme

Écrire une fonction qui prend en paramètre une chaîne de caractères dans un tableau statique, et qui retourne true si la chaîne est un palindrome, false sinon. (kayak est un palindrôme).

D'autres algorithmes classiques :

- 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 !

Matrices - Algo

Matrice = tableau 2D

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

Déclaration (v1) : matrice $N \times M$ (taille fixée)
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 charactères
```

Utilisation :

```
x = t[10][10]; // lecture
chartab[56][7] = 'a'; // écriture
z = t[expr compliquee][exp2];
```

Important Les matrices sont des paramètres modifiables en C (passées par adresse).

Exemples, exercices

Exercice “matrices statiques” classiques

- Impression de toutes les cases d'une matrice carrée
- Impression de la diagonale d'une matrice carrée
- Recherche d'un élément dans une matrice rectangulaire

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 :

```
tab.c:46:4: error: array type 'int [12]' is not assignable
  g=t;
  ~^
1 error generated.
```

Chapitre 5

Complexité, et 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.
- Savoir prouver la correction d'un algorithme récursif.

Algorithmique et Programmation CS101

Cours 05: Notions de complexité algorithmique et de correction de programme

Laure Gonnord

Grenoble INP/Esisar

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.

▶ On précisera plus tard les définitions des "grand O".

Exemple 1

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

▶ 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 ← s + i
    Retourner s
```

▶ 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**.

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 $\leftarrow$ 1
- **Pour** i **de** 1 **à** n **Faire**
  - $exp \leftarrow exp * x$
- **Retourner** exp

- 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.

Maximum d'un tableau

```
Fonction maxtab(t) : entier
    D: t : Tableau[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]
    Retourner (maxi)
```

▶ **Correction** de ce programme ? en TD !

Recherche dans un tableau

```
Fonction is_in_tab(val,t) : bool
    D: t : Tableau[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
        i← i+1
    Retourner fini
```

en TD

Chapitre 6

Algorithmique du Tri

Parmi les algorithmes classiques sur les tableaux, les tris de tableaux sont incontournables. Les algorithmes classiques sont ainsi vus (à l'exception du tri bulle et du tri rapide), et leur complexité est évaluée. Ces algorithmes seront 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 2 *Le tribulle, algorithme classique mais peu efficace, n'est pas traité dans ce cours*

Algorithmique et Programmation CS101

Cours 06 Algos de tri

Laure Gonnord

Grenoble INP/Esisar

2022-23

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
        permuter(T,ideb,imin)
```


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:** 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 $\geqslant$ 1 **et** T[ins-1] $>$ élt) **faire**
    - T[ins] := T[ins-1];
    - ins := ins - 1
  - T[ins] := élt {Insertion de T[i]}

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)

Tri Fusion - 2

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

- 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 ...

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++

Pour i **de** 0 **à** dernier2-premier1 **Faire**
- t[premier1+i] ← fus[i]

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 !

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 7

Schéma d'exécution de fonction

Dans ce cours nous abordons la notion de schéma d'exécution, qui est une abstraction de ce qui se passe en mémoire lors de l'exécution d'un programme C.

Savoirs (liste non exhaustive)

— Savoir faire le schéma d'exécution d'un programme itératif.
— Comprendre le passage de paramètres par valeur en C.
— Savoir faire le schéma d'exécution d'un programme récursif.

Algorithmique et Programmation CS101

Cours 07: schéma d'exécution des fonctions

Laure Gonnord

Grenoble INP/Esisar

2022-23

Un exemple simple

(cf le cours de fonctions)

```
int max(int a, int b){
  int m;
  if (a>b) m=a; else m=b;
  return m;
}
int main(){
  int res = max(23,42);
}
```

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

Modèle mémoire - variables

On modélise la **mémoire** par un tableau d'octets, indexé par des adresses.

- Chaque variable **déclarée** a une adresse (début du placement mémoire). L'emplacement dépend de la portée (globale, locale)

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

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

Schéma d'activation simple

Lors de l'exécution de notre programme, on prend une "photo" de la mémoire associée à chaque fonction.

Exemples

Au début de l'exécution du main :

type	nom	adresse	valeur
int	res	3A10	xxxxx

`main`

Juste avant l'appel de la fonction `max` :

type	nom	adresse	valeur
int	res	3A10	xxxxx
int	arg1	3A14	23
int	arg2	3A18	42

`main`

Schéma d'activation : appel

- Copie des valeurs des paramètres vers le schéma de la fonction appellée. C'est le **passage des paramètres par VALEUR**.
- Lors de l'exécution de la fonction appelée, les schémas de l'appelée et de l'appelante cohabitent en mémoire.
- **La place pour le résultat est prévue**.

type	nom	adresse	valeur
int	a	3B10	23
int	b	3B14	42
int	m	3B18	xxx
int	returnval	3B1A	xxx

`max`

type	nom	adresse	valeur
int	res	3A10	xxxxx
int	arg1	3A14	23
int	arg2	3A18	42

`main`

Schéma d'activation : retour

- Copie du résultat dans la variable prévue par l'appelante.
- Puis le schéma d'activation de la fonction disparaît.
- On revient à la fonction appelante.

▶ Il reste à expliquer comment on peut "revenir".

type	nom	adresse	valeur
int	res	3A10	42
int	arg1	3A14	23
int	arg2	3A18	42

`main`

Modèle mémoire - le code

Le code (compilé) de notre programme est aussi en mémoire :

```
0000000000001149 <max>:
    1149: f3 0f 1e fa           endbr64
    114d: 55                    push   %rbp
    114e: 48 89 e5              mov    %rsp,%rbp
    [...]
 0000000000001172 <main>:
    [...]
      1188: e8 bc ff ff ff        call   1149 <max>
```

desassemblage obtenu avec la commande `objdump -d`

▶ À l'exécution, l'adresse de la **prochaine instruction** à exécuter est calculée.

La pile d'exécution

Pour réaliser correctement l'appel d'une fonction, il faut :

- mémoriser le **contexte** appelant lors de chaque appel de fonction (adresse des variables, mais aussi (l'adresse de) la prochaine instruction, ...)
- restituer ce contexte au retour.

Une pile d'exécution est réalisée. (mais la programmeuse ne voit rien)

▶ Des détails sur le fonctionnement de cette pile d'exécution seront donnés plus tard (en cours d'architecture des microprocesseurs, en particulier).

Représentation simplifiée de la pile d'exécution

Les schémas d'activations sont empilés :

fonction	type	nom	adresse	valeur
max(23,42)	int	a	3B10	23
	int	b	3B14	42
	int	m	3B18	xxx
	adresse de l'instruction suivante ds main			
main()	int	returnval	3B1A	xxx
	int	res	3A10	xxxxx
	int	arg1	3A14	23
	int	arg2	3A18	42

▶ sur papier, on représente les variables que l'on veut regarder.

Pile d'exécution

Exercice : réaliser les étapes de l'exécution du programme suivant :

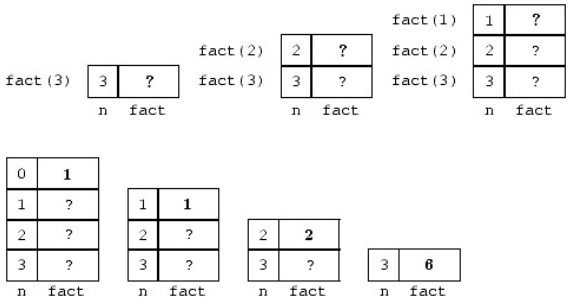
```
int f(int x){
  return (x+1);
}
int g(int y){
  return (y+2);
}
int main(){
  int res = g(f(12));
}
```

(cf TD)

La pile d'exécution - programme récursif

Exemple sur `Facto(3)` :

```
int fact(int n){
  if (n==0)
    return 1;
  else
    return n*fact(n-1);
}
```



Conclusion

- Notion de schéma d'activation, utile pour expliquer le déroulement de programmes **à l'exécution**.
- On n'a pas expliqué comment passer les paramètres "modifiables", ie le cas des tableaux, du scanf. **passage par adresse/notion de pointeur, en 2A**. On peut considérer pour l'instant que les tableaux sont "globaux", ie accessibles tout le temps.

Annexe A

Quelques éléments sur les entrées/sorties, utiles pour PIX

Algorithmique et Programmation CS101

Support PIX 111: quelques éléments de C supplémentaires

Laure Gonnord

Grenoble INP/Esisar

2022-23

Codes de sortie du main

```
exit(EXIT_FAILURE); // macro pour le code d'erreur
return EXIT_SUCCESS; // Pareil que return 0 mais plus lisible.
```


Argc/Argv

```
int main(int argc, char** argv){
 printf("nombre d'arguments = %d\n", argc-1);

 // argv[0] est une chaîne de caractères
 printf("le nom du binaire est %s\n", argv[0]);

 // chaque argv[i] (pour i de 1 à argc -1) est une chaîne
 for (int i=1; i< argc; i++){
  printf("argument numero %d est %s\n", i, argv[i]);
 }
 //...
```

Entrées : scanf - Sorties printf

```c
printf("voici une chaîne formattée %d %s\n", 32, "hello");

int x;
printf("Donne moi un entier stp?");
scanf("%d",&x);

char line[12]={0};
printf("Donne moi des caractères stp?");
scanf("%s", line); // ch est déjà  un tableau (donc modifiable)
printf("%s\n", line);
puts(line); // une autre façon d'imprimer la chaîne.
```


getchar, putchar

```c
while ((getchar()) != '\n'); // flush (il reste un \n non lu...)

printf("Donne moi un caractère stp?");
char c = getchar();
int res=putchar(c); // cast to int if success, EOF on error
printf(", res= %d\n", res);
```

(et leurs amies puts, gets . . .) cf fichier `es.c`

Fichiers

```c
FILE* f = fopen(filename, "r"); // ouverture en lecture seulement
if (f==NULL){
  fprintf(stderr, "open error\n"); // impression de message sur
     sortie d'erreur
  exit(EXIT_FAILURE); // macro pour le code d'erreur
 } else {
  printf("Successfully open!\n");
  while(!feof(f)){ // tant que l'on n'a pas la fin de fichier !!!
   // utiliser fscanf par exemple.

  }
}
```


Boucle réactive

Très utilisée dans les IHM, et en PIX pour réaliser des jeux qui nécessitent une interaction avec l’utilisateur·rice, voici un schéma général :

```c
bool fini = false;
while (!fini){ // ou boucle infinie
  demande_info();
  calcule();
  if (condition) fini = true;
}
```