Programmation Impérative
Introduction au langage C

Emilie Morvant

Faculté des Sciences et Techniques
Université Jean Monnet de Saint-Etienne

Licence 2 Informatique
Semestre 3

Qui suis-je ?

- Maître de Conférences — Responsable de la L2

  à l'Université Jean Monnet, St-Etienne, France
  au Laboratoire Hubert Curien dans le Groupe "Data Intelligence"

- Domaine de recherche :
  - Domaine principal : Apprentissage Automatique *(Machine Learning)*
  - Théorie de l'apprentissage automatique statistique

- Comment me joindre :

  Mail : `emilie.morvant@univ-st-etienne.fr`

Objectifs du cours

- Découvrir le langage de programmation impérative C
  - après Python en L1
- Maitriser l'analyse et la programmation impérative simple

Références

- Il y en a des tonnes !
- Un cours de programmation C :
  `https://www.rocq.inria.fr/secret/Anne.Canteaut/COURS_C/cours.pdf`
  (Le cours du semestre 4 se base sur ce poly)

Introduction

Objectifs

- Paradigmes de programmation
- Interprétation, compilation
- Conception de programmes
- Aspects pratiques de la conception de programmes

Notion d'algorithme

Qu'est ce qu'un algorithme ?

$\hookrightarrow$ C'est une suite d'actions exécutées en séquence

Exemple

1. Avancer 3 pas
2. *(puis)* Tourner à droite
3. *(puis)* Dire "Salut !"

On peut aussi faire appel à des **actions conditionnelles "if...then...else"**

Exemple

si Le temps est beau **alors**
    Chanter "Hello, le soleil brille brille brille..."
sinon
    Dire "Oh non, il pleut !"
fin si

Notion d'algorithme

Exemple de problème d'algorithme

Avancer de 55 pas en ligne droite
(puis)
si il y a un mur **alors**
  dire "Le mur est là"
sinon
  dire "Pas de mur"
fin si

Que se passe-t-il lorsque l'on ne peut pas exécuter les 55 pas ? !
(ex : le mur arrive avant les 55 pas)

$\Longrightarrow$ ERREUR !

Solution

Il faut pouvoir tester à chaque pas

Notion d'algorithme

On va utiliser une **instruction répétitive, en boucle "while"**

Exemple

nombre_de_pas $\leftarrow$ 0 /* on stocke le nombre de pas dans une variable */
tant que il n'y a pas un mur **et** *nombre_de_pas* < 55 **faire**
  Avancer de 1 pas en ligne droite /* on fait un pas de plus */
  nombre_de_pas $\leftarrow$ *nombre_de_pas* $+ 1$ /* on incrémente la variable */
fin tant que
si il y a un mur **alors**
  dire "Le mur est là"
sinon
  dire "Pas de mur"
fin si

Notion d'algorithme

Une autre **instruction répétitive, en boucle "do. . .while"**

Exemple

nombre_de_pas $\leftarrow$ 0
repéter
  Avancer de 1 pas en ligne droite
  nombre_de_pas $\leftarrow$ *nombre_de_pas* + 1
jusqu'à arriver au mur **ou** *nombre_de_pas* == 55
si il y a un mur **alors**
  dire "Le mur est là"
sinon
  dire "Pas de mur"
fin si

Notion d'algorithme

Il y a donc 2 types d'instructions répétitives

Boucle "while"

tant que condition/test **faire**
  suite d'opérations
fin tant que
opération(s) suivante(s)

Boucle "do...while"

repéter
  suite d'opérations
jusqu'à condition/test
opération(s) suivante(s)

La différence principale

La "suite d'opérations"
- peut **ne pas** être exécutée avec un "while"
- est exécutée **au moins une fois** avec un "do...while"

De l'algorithme au programme dans l'ordinateur

Définition : Ordinateur (source : wikipedia)

Un ordinateur est une machine électronique qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques sur des chiffres binaires. Dès sa mise sous tension, un ordinateur exécute, l'une après l'autre, des instructions qui lui font lire, manipuler, puis réécrire un ensemble de données. Des tests et des sauts conditionnels permettent de changer d'instruction suivante, et donc d'agir différemment en fonction des données ou des nécessités du moment.

Quels composants d'un ordinateur ?

De l'algorithme au programme dans l'ordinateur

Définition : Ordinateur (source : wikipedia)

Un ordinateur est une machine électronique qui fonctionne par la lecture séquentielle d'un ensemble d'instructions, organisées en programmes, qui lui font exécuter des opérations logiques et arithmétiques sur des chiffres binaires. Dès sa mise sous tension, un ordinateur exécute, l'une après l'autre, des instructions qui lui font lire, manipuler, puis réécrire un ensemble de données. Des tests et des sauts conditionnels permettent de changer d'instruction suivante, et donc d'agir différemment en fonction des données ou des nécessités du moment.

Quels composants d'un ordinateur ? Disque dur, mémoire, processeur
C'est le processeur qui fait les opérations (calcul, test)
Le programme est la suite des opérations

- Opérations de haut niveau (ex : "si $t > 10$ alors ...")
  Elles sont transformées en de multiples opérations de bas niveau
- Opérations de bas niveau
  ex : INC R3 $\rightarrow$ ajoute 1 à une case mémoire désignée par R3

De l'algorithme au programme dans l'ordinateur

En fait, l'ordinateur, ou plutôt le processeur, travaille en **langage machine** qui est une succession de chiffres binaire (Bit)

0 : le courant ne passe pas

1 : le courant passe

- le stockage physique des bits dépend des technologies employées :

  différentiels de tension (condensateurs), moments magnétiques, cuvettes ou surfaces planes, émission de photons, etc

- Composition de bit : octet (=byte) — 8, 16, 32 ou 64 bits
- Codage en base 2 :

$$0001 = 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$
$$0010 = 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 2$$
$$0011 = 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$$
$$0100 = 0 \times 2^3 + 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4$$

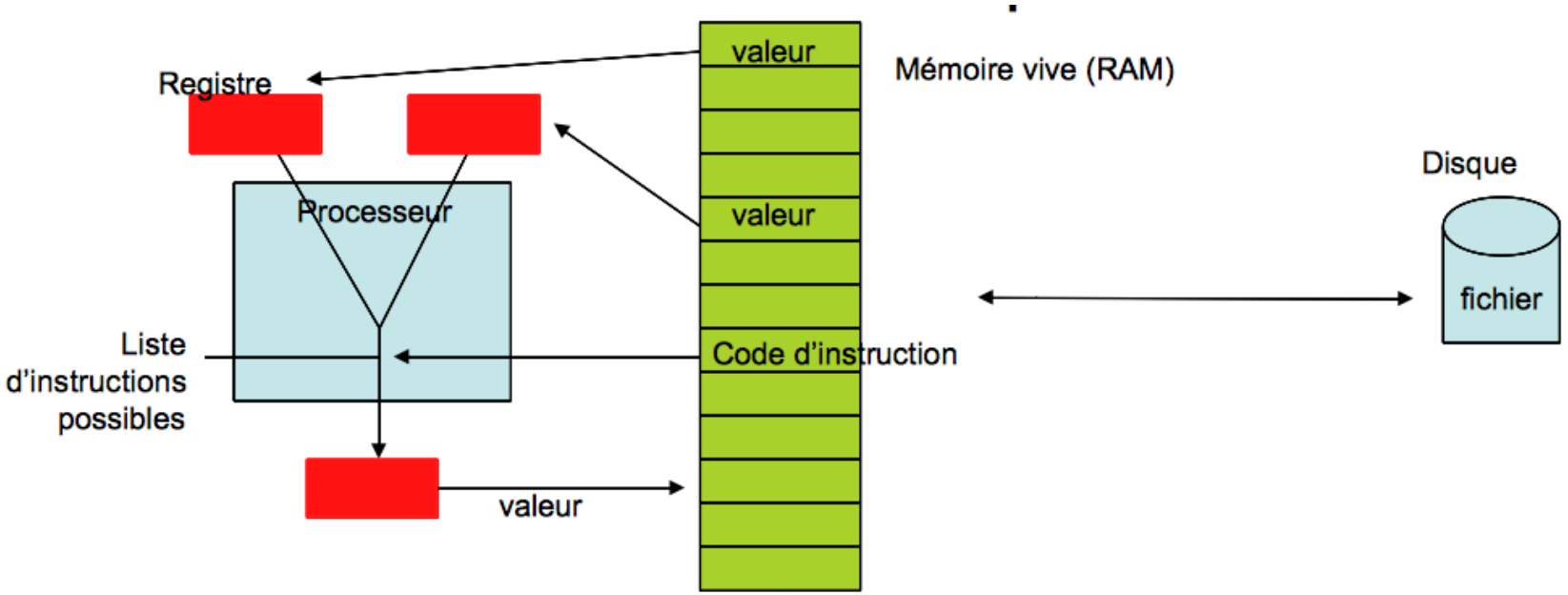
De l'algorithme au programme dans l'ordinateur

- Zone mémoire
- Un registre porte une valeur : R3 00111000
- Instruction du processeur : ADD R3, R4
- ADD porte un numéro d'instruction (ex : 01010011)

Un programme est une suite **d'instructions élémentaires** traitant des valeurs

De l'algorithme au programme dans l'ordinateur

- Mémoire de stockage
  ↪ stocke **durablement** des valeurs et des instructions
- Mémoire vive
  ↪ stocke **temporairement** des valeurs et des instructions
- Registre → porte une valeur
- Processeur → réalise des opérations



De l'algorithme au programme dans l'ordinateur

Les ordinateurs opèrent des instructions de bas niveau via le langage machine

$\Rightarrow$ Trop élémentaire et donc trop long à programmer pour l'humain

$\Rightarrow$ Nécessité de construire un langage de programmation pour avoir

- des programmes à un niveau d'abstraction plus élevé
- une meilleure expressivité et générécité
- moins de risques d'erreurs

$\Rightarrow$ Besoin de passer du langage de programmation à une exécution machine

$\Rightarrow$ Un programme doit être successivement

- saisi et enregistré (édition) $\rightarrow$ fichier texte
- traduit (compilation / interprétation) $\rightarrow$ code binaire
- "relié" (édition de liens) $\rightarrow$ fichier exécutable
- exécuté, testé, mis en service (débogage)

De l'algorithme au programme dans l'ordinateur

Saisir un programme à l'aide d'un éditeur de texte

- Les mauvais choix : Notepad, MS-word, etc.
- Les bons choix : Les éditeurs spécialisés

Les avantages des éditeurs spécialisés :

- Vocabulaire du langage reconnu
- Aide à la saisie
- Environnement dédié
- Exemples : **emacs**, codeblocks

De l'algorithme au programme dans l'ordinateur

Compiler un programme

Un programme dans un langage **est traduit** en langage machine pour être exécuté

- Écrire un programme dans un ou plusieurs fichiers : **le(s) fichier(s) source(s)**
- Soumettre au compilateur les fichiers

$\rightarrow$ Traduction dans un langage exécutable par la machine : **fichier cible**

- Ensuite, on peut lancer l'exécution

IMPORTANT : Le compilateur est mon ami

- Il indique les erreurs de compilation

> Il faut corriger UNIQUEMENT la première erreur pointée, puis recompiler

- Il emet des "warning" lorqu'il détecte des incohérences dans le code

Un programme correct doit compiler sans avertissement (nécessaire, mais pas suffisant...)

De l'algorithme au programme dans l'ordinateur

Quelques remarques importantes

- Un exécutable **n'est valable que** pour un couple processeur/SE

  $\Rightarrow$ Il peut être nécessaire de "porter" un programme

- L'écriture et la compilation doivent être répétées tant qu'il reste
  - des erreurs de syntaxe (messages d'erreur du compilateur)
  - des erreurs de sémantique (prouver un programme)

De l'algorithme au programme dans l'ordinateur

Interpréter un programme (action automatique)

- Traduire **et** exécuter au fur et à mesure (à la volée) : instruction par instruction
  - Écriture d'un programme dans un fichier
  - Soumission à l'interpréteur
  - Évaluation (traduction + exécution) des instructions l'une après l'autre

- Plus lent que la compilation

  puisque la traduction se fait à chaque exécution via un interpréteur, au lieu d'une fois pour toute par un compilateur

De l'algorithme au programme dans l'ordinateur

Exemples de langages de programmation

- Langages compilés : C, C++, Ada, Pascal, Cobol, ...
- Langages (semi)-interprétés : Caml, PHP, Python, Java, SQL, HTML

Machines virtuelles

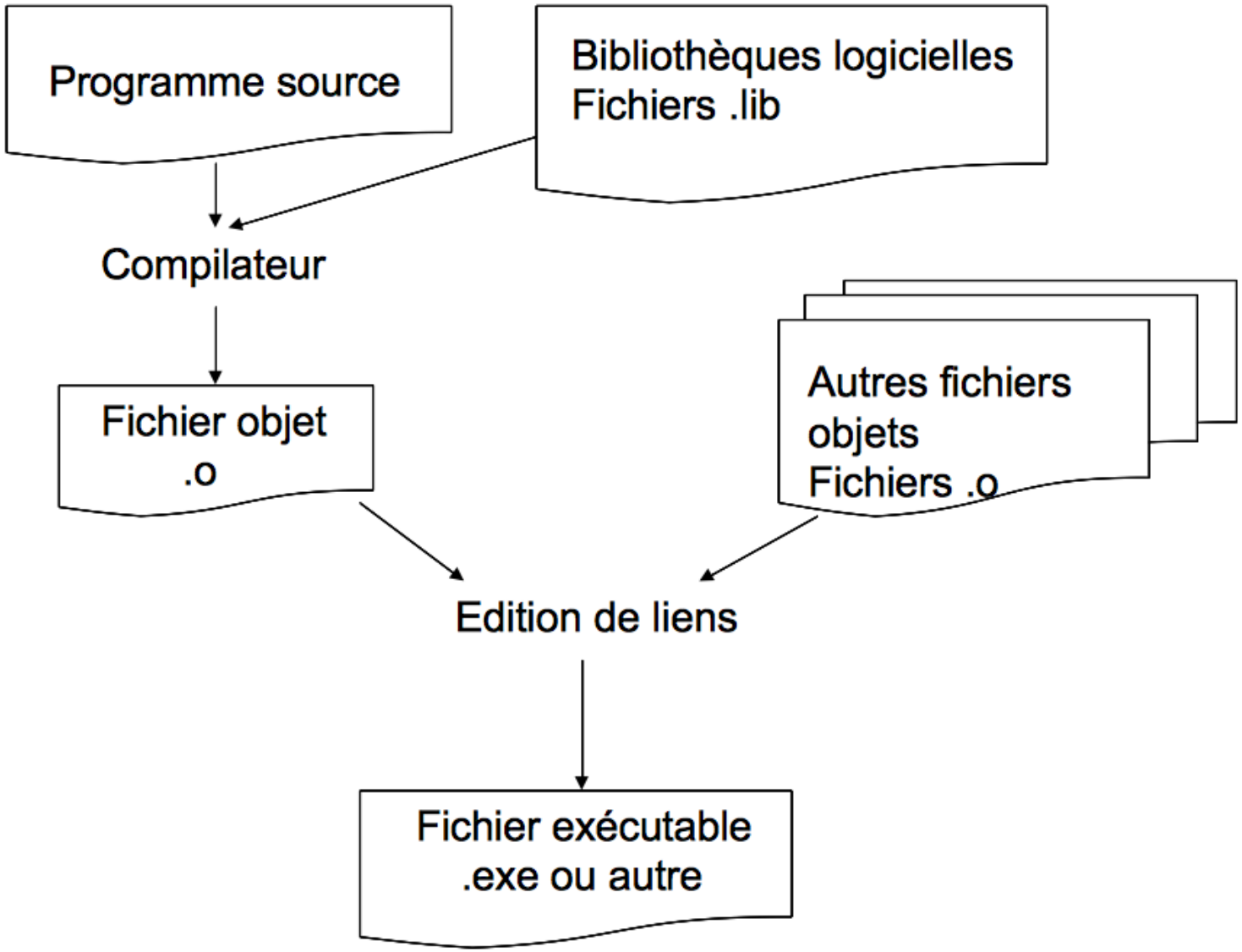
- Compilation d'un langage L dans un langage intermédiaire L'
- La machine virtuelle permet d'exécuter le programme en l'interprétant (L' proche du langage machine pour être efficace)
- La machine virtuelle est spécifique à l'ordinateur utilisé
- Langages dits semi-interprétés comme Java

De l'algorithme au programme dans l'ordinateur

Edition de Liens

- **concerne les langages compilés**
- Permet d'écrire de gros programmes (dans plusieurs fichiers)
- Permet la réutilisation de modules
- Compilation : chaque fichier source est compilé
  et produit un fichier "objet"
- **Edition de liens : fusion des fichiers objets pour créer le programme binaire exécutable**

De l'algorithme au programme dans l'ordinateur



De l'algorithme au programme dans l'ordinateur

Les paradigmes de programmation

- Programmation impérative

  Le programmeur spécifie explicitement l'enchaînement des instructions à exécuter (C, Java, C++, Ada, etc.)

- Programmation fonctionnelle

  Un programme est un ensemble de fonctions
  L'exécution est une évaluation de fonction (LISP, CAML)

- Programmation logique

  Un programme est un ensemble de théorèmes
  L'exécution en est une preuve (Prolog)

De l'algorithme au programme dans l'ordinateur

Étapes de la conception d'un programme

1. Spécifications des entrées et sorties

   Spécifier les entrées
   (données du programme, fichiers ou saisies par l'utilisateur)

   Spécifier les traitements

   Spécifier les sorties (fichier et/ou écran)

2. Spécifications des traitements :
   algorithmes, texte compréhensible décrivant les traitements

3. Programmation :
   traduction des algorithmes dans un langage de programmation

Les étapes 1 et 2 sont les plus importantes ($\simeq 90\%$)
On ne programme pas sans savoir où l'on va...

Durant ce cours...

- Langage C
- Editeur de texte emacs
- Compilateur gcc
- Edition de liens ld
- Fichier d'exécution (nom par défaut) "a.out"

Le langage C
Historique et caractéristiques

Historique du langage C et caractéristiques

- Date : début années 1970
- Auteurs : Kernighan et Ritchie, Bell Labs / ATT
- But : proposer un langage impératif compilé, à la fois de haut niveau et "proche de la machine" (rapidité d'éxécution)
- Conçu pour être le langage de programmation d'Unix, premier système d'exploitation écrit dans un langage autre qu'un langage machine
- Diffusé grâce à Unix
- Popularisé par sa concision, son expressivité et son efficacité
- Disponible actuellement sur quasiment toutes les plate-formes

Historique du langage C et caractéristiques

- Proche de la machine :

(+) rapidité, programmation facile sur nouvelles architectures matérielles

(–) exécutable non portable

- Langage simple :

(+) compilateur simple

(–) (dans les 1ères version) peu de vérification à la compilation, plantages

- Langage vieux (1970) et populaire :

(+) beaucoup d'outils annexes, de bibliothèques réutilisables

(–) ne supporte pas les "nouveautés" (orienté objet, exception, ramasse-miettes...)

Historique du langage C et caractéristiques

Langage impératif et de haut niveau

- Programmation structurée (= pas d'instruction "Go to" dans le programme, pour éviter le plat de spaghettis)
- Organisation des données (regroupement en structures de données)
- Organisation des traitements (fonctions/procédures avec paramètres)

  Possibilité de programmer "façon objet"

Langage de bas niveau

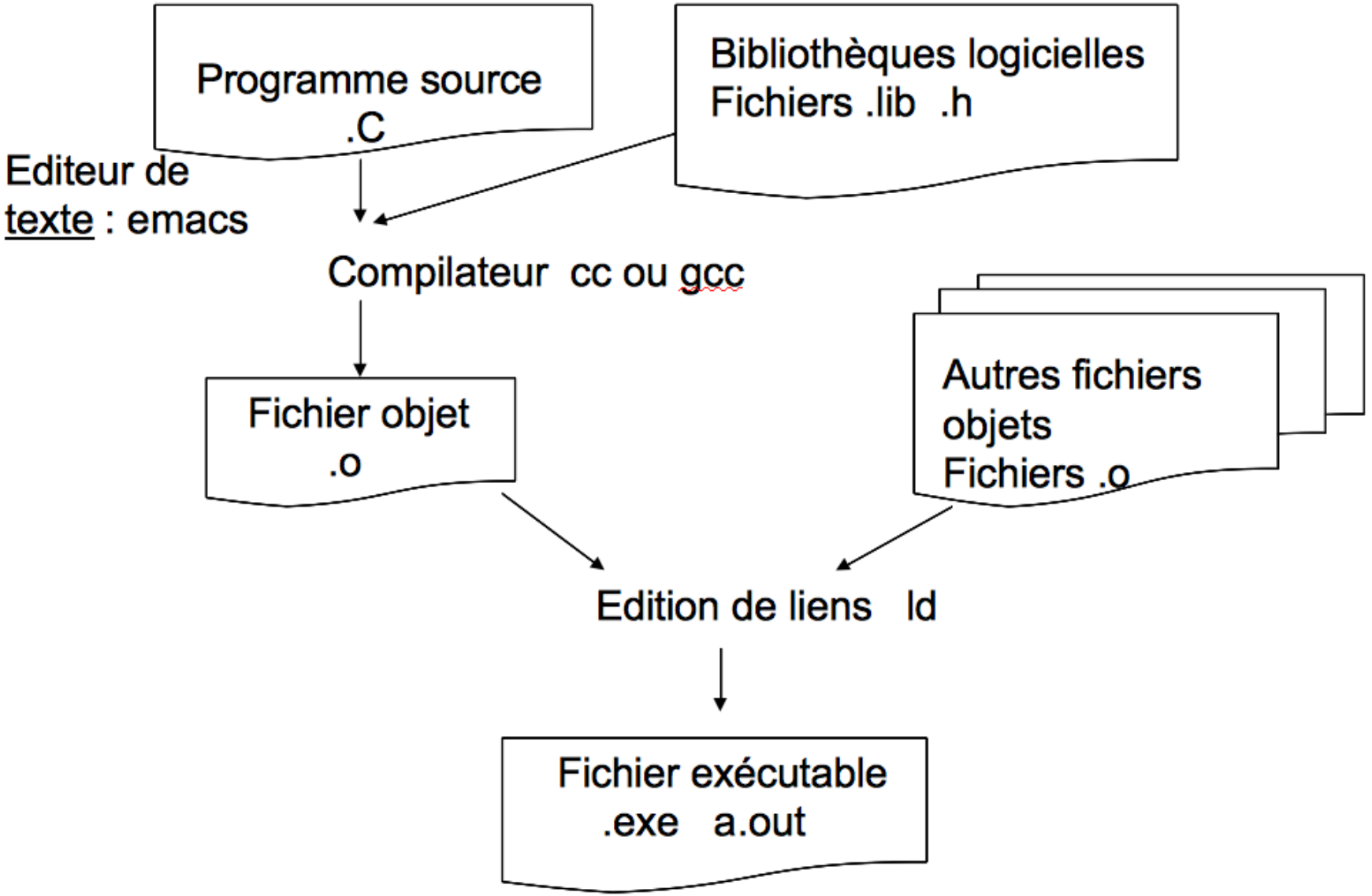
- Conçu pour être facilement traduit en langage machine
- Gestion de la mémoire "à la main"
- Attention : pas de gestion des exceptions

La compilation en C

Le langage C est un langage compilé

- Le programmeur écrit son programme sous la forme d'un code source, contenu dans un ou plusieurs fichiers texte d'extension ".c"
- Un programme appelé compilateur (habituellement nommé cc, ou gcc) vérifie la syntaxe du code source et le traduit en code objet qui sera compris par le processeur
- Le programme en code objet *(par défaut "a.out")* ainsi obtenu peut alors être exécuté sur la machine

Schéma de production de logiciel en C



Exemple de programme simple

Fichier texte “bonjour.c”

```
#include <stdio.h>      /* bibliothèque utilisée = liste de fonctions */
main ()                 /* mot clé obligatoire pour le début du prog */
{                       /* début d'un ensemble d'instructions = bloc */
printf ("Bonjour!\n");  /* instruction d'affichage */
}
```

Lignes de commandes

```
% gcc bonjour.c         Lancement de la compilation
                        note : gcc fait aussi l'édition de liens
% a.out                 Lancement du programme (nom par défaut)
Bonjour!                Résultat de l'exécution
%
```

Un autre exemple de programme : “diviseur.c”

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>
#include <stdlib.h>
void usage (char *s) {
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {
  int n, i;
  if (argc < 2) {
     usage(argv[0]);
     exit(-1);
  }
  n= atoi(argv[1]);
  for (i=1 ; i <=n; i++)
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  exit(0);
}
```

Un autre exemple de programme : “diviseur.c”

La compilation de “diviseur.c”

```
% gcc -Wall -ansi diviseur.c -o diviseur
```

diviseur.c (+stdio.h + stdlib.h) $\longrightarrow$ diviseur

Les options de compilation

- `-o nomfichier` : nom du fichier de sortie/exécutable (défaut : “a.out”)
- `-Wall` : pour avoir tous les avertissements du compilateur
- `-ansi` : pour compiler du C standard
- etc.

Un autre exemple de programme : “diviseur.c”

La compilation de “diviseur.c”

```
% gcc -Wall -ansi diviseur.c -o diviseur
```

diviseur.c (+stdio.h + stdlib.h) $\longrightarrow$ diviseur

Exécution de diviseur

```
% diviseur
Usage : diviseur <entier>, <entier> positif
% diviseur 24
1 2 3 4 6 8 12 24
%
```

Comprendre un programme en C

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                          /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                      /* Déclaration d'une fonction */
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {         /* Déclaration d'une fonction */
  int n, i;                                 /* Déclaration de variables   */
  if (argc < 2) {                           /* Suite d'instructions dans  */
     usage(argv[0]);                        /* des blocs { ... }          */
     exit(-1);                              /* chaque instruction se      */
  }                                         /* termine par un ;           */
  n= atoi(argv[1]);
  for (i=1 ; i <=n; i++)
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  exit(0);
}
```

Comprendre un programme

Un programme C est classiquement composé de

- commentaires `/* ça peut aider à comprendre */`
- directives d'inclusions de fichiers/librairies `#include`
- (variables globales)
- définitions de fonctions, dont la fonction `main()`

Comprendre un programme

Les commentaires

- Encadrés par `/*` et `*/`
- Ignorés par le compilateur ! !
- Texte libre, multi-ligne
- Utile pour relire et comprendre un programme (description en langage clair, choix effectués, etc.)

Exemple

```
/*** //* tout ceci est du "commentaire"
et ceci aussi : main( ) { i++;} */
```

Comprendre un programme

Directives d'inclusions

- en début de ligne : `#include`
- Biliothèques "système" :

  `#include <nomBiblio>`

  utilise un répertoire connu du compilateur, généralement : `/usr/include`

  Il existe :

  `stdio, stdlib, ctype, string, time, math, locale, setjmp`, etc.

- Mes propres fichiers :

  `#include "monProjet/nomFichier"`

  utilise le chemin et le fichier indiqués

Comprendre un programme

Définition de fonctions

- Un programme C est un ensemble de "fonctions"

  Par convention les noms de fonctions commencent par une minuscule

- Structure :

```
type_sortie nom_fonction (type_1 argument_1, type_2 argument_2, ...){
        type_variable_1 nom_variable_1 ;
        type_variable_2 nom_variable_2 ;
        ...
        instruction_1 ;
        instruction_2 ;
        ...
}
```

- Une fois définie (*i.e* "au dessus" dans le texte),
  la fonction peut-être appelée dans une autre fonction de la manière suivante : `nom_fonction(valeur_1,valeur_2,...) ;`

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                          /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                      /* Déclaration d'une fonction */
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {         /* Déclaration d'une fonction */
  int n, i;                                 /* Déclaration de variables   */
  if (argc < 2) {                           /* Suite d'instructions dans  */
     usage(argv[0]);                        /* des blocs { ... }          */
     exit(-1);                              /* chaque instruction se      */
  }                                         /* termine par un ;           */
  n= atoi(argv[1]);
  for (i=1 ; i <=n; i++)
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  exit(0);
}
```

Comprendre un programme

Définition de fonctions

- Une fonction est spéciale : `int main (int argc, char * argv[])`
  - Doit TOUJOURS être présente
  - C'est le point de commencement du programme :
    c'est la fonction **principale**, **exécutée en premier**
  - Peut prendre 2 arguments : `argc` et `argv`
    `argc` : nombre d'arguments sur la ligne de commande ("c" = count)
    `argv` : liste des arguments ("v" = value)

Exemple

```
% diviseur 24
```
au début du programme, `argc = 2` et `argv = [''diviseur'', ''24'']`

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                          /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                      /* Déclaration d'une fonction */
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {         /* Déclaration d'une fonction */
  int n, i;                                 /* Déclaration de variables   */
  if (argc < 2) {                           /* Suite d'instructions dans  */
     usage(argv[0]);                        /* des blocs { ... }          */
     exit(-1);                              /* chaque instruction se      */
  }                                         /* termine par un ;           */
  n= atoi(argv[1]);
  for (i=1 ; i <=n; i++)
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  exit(0);
}
```

Comprendre un programme

Exemples de fonctions pré-définies

- La fonction d'affichage

  `printf(<chaîne de caractère>{, <liste>})`

  - `<chaîne>` : chaine de caractères, contenant aussi des formats comme :
    `%d` entier, `%f` flottant, `%c` caractère, `%s` chaîne de caractères
    - caractères spéciaux : `\n` saut de ligne, `\t` tabulation, `\\` caractère `\`

  - `<liste>` : liste d'expressions (associées aux formats)
    Chaque format correspond à une expression (dans l'ordre)

- Effet : affiche la chaîne de caractères en la "formatant" selon le format spécifié et les expressions calculées

Comprendre un programme

Exemples de fonctions pré-définies

Exemple d'utilisation de la fonction `printf()`

```
int x;
x = 5;
printf ("Le carré de x est %d.\nMerci.\n", x*x);
printf("Pour afficher \\, il faut écrire \\\\.\n");
```

Affiche :

```
Le carré de x est 25.
Merci.
Pour afficher \, il faut écrire \\.
```

Remarque : `\n` en fin de chaîne permet d'"assurer" l'affichage (sinon risque de non affichage en cas d'erreur) et facilite la lisibilité.

Comprendre un programme

Exemples de fonctions pré-définies

- La fonction de lecture

  `scanf(<format>, <liste>)`

- La fonction `scanf` est la fonction symétrique à `printf`
  elle offre pratiquement les mêmes conversions que `printf`, mais en sens inverse
  - `<format>` : format de lecture des données
  - `<liste>` : adresses des variables auxquelles les données seront attribuées

    Rq : l'adresse de la variable `var` est : `&var`

- C'est une instruction bloquante :
  le programme "attend" que l'utilisateur entre des valeurs, puis valide

À lire : `https://openclassrooms.com/courses/la-saisie-securisee-avec-scanf`

Exemples de fonctions pré-définies

Exemples d'utilisation de la fonction `scanf()`

```
int jour, mois, annee;
scanf("%d %d %d", &jour, &mois, &annee);
```

Lit trois entiers relatifs, séparés par des espaces, tabulations ou interlignes

Les valeurs sont attribuées respectivement aux variables `jour`, `mois`, `annee`

Si on entre : `06 10 2014`
alors : `jour=6`, `mois=10` et `annee=2014`

Remarque : Une suite de signes d'espacement est évaluée comme un seul espace

Comprendre un programme

Exemples de fonctions pré-définies

- La fonction de conversion d'une chaîne en entier

  `atoi(<chaine>)`

  - `<chaine>` : la chaîne de caractères que l'on désire convertir
  - Exemple : `n = atoi(''12'');`

- Usage typique : conversion d'un nombre entré au clavier

Exemple

```
int main (int argc, char *argv[]) {
     int n;
     n=atoi(argv[1]);
     ...
}
```

Comprendre un programme

```
/* nom : diviseur.c
   entrée : entier n > 0     sortie : affiche les diviseurs de n */
#include <stdio.h>                          /* Inclusion de fichiers      */
#include <stdlib.h>
void usage (char *s) {                      /* Déclaration d'une fonction */
        printf("Usage : %s <entier>",s);
        printf("<entier> positif\n");
}
int main (int argc, char *argv[]) {         /* Déclaration d'une fonction */
  int n, i;                                 /* Déclaration de variables   */
  if (argc < 2) {                           /* Suite d'instructions dans  */
     usage(argv[0]);                        /* des blocs { ... }          */
     exit(-1);                              /* chaque instruction se      */
  }                                         /* termine par un ;           */
  n= atoi(argv[1]);
  for (i=1 ; i <=n; i++)
      if (n%i == 0)
           printf(" %d ", i);
  printf("\n");
  exit(0);
}
```

Variables et types

Variables et types

Quelques généralités

- Objet de base en C
- Les variables sont typées et déclarées explicitement avant toute utilisation (permet d'éviter des erreurs via le compilateur qui contrôle)
  Exemple : `type nomVariable ;`
- Triplet (type, nomVariable, valeur)
- La valeur peut changer
- L'initialisation peut se faire lors de la déclaration
  Exemple : `int maVariable = 0;`
- Le nom des variables
  - Caractères de A à Z, a à z, 0 à 9 et _
  - Doit débuter par une lettre (convention : en minuscule)
  - Longueur quelconque
  - Un nom significatif est à préférer ! ! ! ! (relecture extrèmement plus facile)

Variables et types

Le type des variables

- Le type d'une variable est une contrainte de sécurité :
  - "contrôle" les opérations et les valeurs admises

- 2 types de types
  - **Types simples**
  - Types construits (étudiés plus tard)

Variables et types

Les types simples (mots réservés)

- `char` : 1 seul caractère (8bits) (entre ' ')
- `char[]` : tableau de caractères = chaîne de caractères (entre " ")
  Attention : la manipulation est à apprendre
- `int` : entier (16 ou 32 bits, soit 2 ou 4 octets, selon l'ordi)
  il existe aussi : `unsigned int`, `long int`, `short int`
- `float` : flottants (réels)
- `double` : flottants dit doubles (plus grande valeur possible)
- `void` : aucune valeur

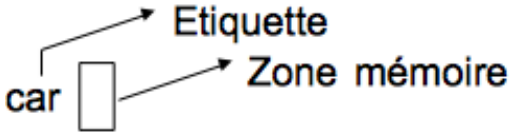
Variables et types

Les types simples

Exemples et notion de zone mémoire (un octet 8 bits)

- Déclarations :

```
char car;
```
$\Longrightarrow$

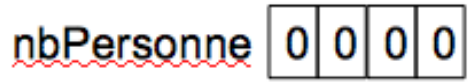



```
int i;
```
$\Longrightarrow$


```
double resultat;
```
$\Longrightarrow$

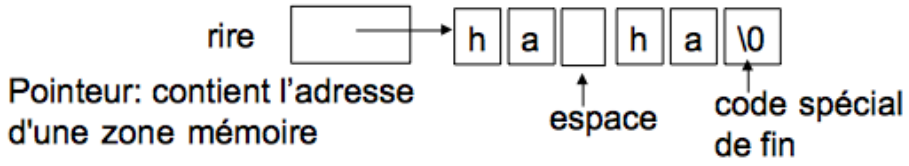
- Déclarations avec initialisation :

```
int nbPersonne = 0 ;
```
$\Longrightarrow$




```
char c = 'b';
```
$\Longrightarrow$


```
char rire[] = ''ha ha'';
```
$\Longrightarrow$



Variables et types

Des variables constantes !

Mot clé : `const`

On l'utilise lors d'une déclaration de variable initialisée pour indiquer au compilateur d'**interdire tout changement de valeur de cette variable**

Exemple : `const int joursParSemaine = 7;`

Variables et types

Les (pseudo-)constantes

- Définition en début de ligne juste après les inclusions (et en dehors de toute fonction)
- `#define NOM valeur`
  - `#define` est un nom réservé
  - `NOM` est en majuscule par convention

Exemple

```
#define PI 3.1415926
#define MESSAGE_3176 "Bienvenue sur emacs"
```

- Intérêts
  - Lisibilité du code
  - Facilite la maintenance, la traduction, etc.
- Fonctionnement : le pré-compilateur remplace `NOM` par `valeur`

Expressions

- Formées d'opérandes et d'opérateurs (44 prédéfinis)
  - Opérandes : variables, constantes, appels de fonctions
  - Opérateurs : nous en utiliserons 22
  - Les opérateurs sont associatifs à gauche

Variables et types

Les opérateurs arithmétiques

- `+` addition
- `-` soustraction
- `*` multiplication
- `/` division (division entière avec des entiers)
- `%` modulo (reste de la division entière)
- ordre des priorités : `*`, `/`, `%` prioritaires à `+`, `-`

Variables et types

Les opérateurs relationnels

- == test d'égalité

  Attention : ce n'est PAS =

  = est l'opérateur d'affectation de base et renvoie la valeur affectée !

- != test d'inégalité

- <, <= inférieur (ou égal)

- >, >= supérieur (ou égal)

Variables et types

Les opérateurs logiques

- && ET
- || OU
- ! NON
- Il n'y a pas de type booléen ! MAIS
  0 (zéro) est considéré comme FAUX
  'valeur non nulle' est considérée comme VRAI

Exemple

```
int maVariable;
maVariable = ...;
if (maVariable)          /* si variable != 0 */
...
```

Variables et types

Les opérateurs d'affectation

- Forme générale : `[variable] <operateur> [expression]`
- `[variable]` représente une variable existante
- L'évaluation de `[expression]` donne un résultat qui devient la nouvelle valeur de `[variable]`
- `<operateur>`
  - `=` affectation simple
  - `+=` affectation du résultat de `[variable] + [expression]`
    idem avec `-=`, `*=`, `%=`, `&=`, `|=`

Exemple

```
int i;
i=11*2;      /* i prend la valeur 22 */
i*=4;        /* i prend la valeur de i*4, ie 22*4, ie 88 */
```

Variables et types

Les opérateurs pour incrémenter et décrémenter

- **Sur des variables entières**
- `[variable]++` est équivalent à `[variable] += 1`
  est équivalent à `[variable] = [variable] + 1`
- `[variable]--` est équivalent à `[variable] -= 1`
  est équivalent à `[variable] = [variable] - 1`
- Pré et post fixé :

  si le ++ ou le -- est devant, alors l'opération est première

Exemple

```
int i,j,k;
i=4;
j= 10 + i++;    /* j vaut 14 et i vaut 5 */
k= 10 + ++i;    /* k vaut 16 et i vaut 6 */
```

- Remarque : Peut être utilisé pour d'autres variables (pour programmeur confirmé ou le devenant)

Les instructions

Les instructions

- instruction vide : ;
- instruction expression : `<expression>` ;
- instruction composée : un bloc = une suite d'instructions (vides, simples et/ou composées)

  un bloc est délimité par { au début et } à la fin
- Remarque : l'indentation du code aide à la lecture

Exemple

```
{                   /* début de bloc 1 */
 int i;             /* instruction simple */
     {              /* début de bloc 2 */
      i=14;         /* instruction simple */
     }              /* fin de bloc 2 */
  ;                 /* instruction vide */
}                   /* fin de bloc 1 */
```

Les instructions

Les instructions conditionnelles

Structure :

```
if (<expression)        /* evaluation Vrai (!=0) ou Faux (0)  */
      <instruction>     /* exécution si Vrai (c'est le "then")*/
else
      <instruction>     /* exécution si Faux                  */
<instruction suivante> /* executé dans tous les cas          */
```

La clause `else <instruction>` est optionnelle

Recommandations importantes

- bien présenter : indentation !
- travailler avec des blocs { }

Les instructions

Les instructions conditionnelles : Pourquoi indenter et utiliser { } ?

`if(<expression>) <instruction> else <instruction>`

`<instruction>` = instruction simple ou bloc

Exemple

```
if (C1 && C2 || C3)
<instruction simple 1>
<instruction simple 2>
else <instruction>
```

⇒ Provoque une erreur

Un autre exemple

```
if (C1 && C2 || C3)
<instruction>
else
<instruction simple 1>
<instruction simple 2>
```

⇒ ne provoque pas d'erreur mais peut en cacher une. . .

Les instructions

Les instructions conditionnelles : Emboîtements

Le `else` se rapporte au `if` le plus proche

Exemple

`if (C1) if (C2) l1 else l2` se "lit" :

```
if (C1)
   if (C2)
         l1            l2 est exécuté si C1 est vrai et C2 faux
   else
         l2
```

Un autre exemple

On peut introduire des accolades, `if (C1) {if (C2) l1} else l2` se "lit" :

```
if (C1){
   if (C2)
         l1            l2 est exécuté si C1 est faux
}else
         l2
```

Les instructions

Les instructions conditionnelles : Un exemple

Nombre de racines de $ax^2 + bx + c = 0$ et solutions

```
int main(int argc, char *argv[]){
   float a, b, c, delta, x1, x2;
   int nbRacines;
   a=atof(argv[1]); b=atof(argv[2]); /* Rq: atof(<chaine>):comme atoi() */
   c=atof(argv[3]);                  /* mais pour les flottants         */
   delta = b*b-4*a*c;
   if (delta>0){
       nbRacines = 2;
       x1=(-b+sqrt(delta))/(2*a);
       x2=(-b-sqrt(delta))/(2*a);
       printf("Nombre de racines : %d\n solutions :
              %f et %f\n",nbRacines,x1,x2);
   }else
       if (delta==0){
           nbRacines = 1;
           x1 = x2 = -b/(2*a);
       printf("Nombre de racines : %d\n solution :
              %f\n",nbRacines,x1);
       }
       else{
           nbRacines = 0;
           prinft("Nombre de racines : %d\n",0);
       }
   exit(0);
}
```

Les instructions

Les instructions conditionnelles : Une autre écriture

Forme condensée du “if” :

```
(condition) ? <expression1> : <expression2> ;
```

est équivalent à

```
if (condition)
   <expression1>
else
   <expression2>
```

Exemples

- Stocker dans `c` le minimum entre deux nombres `a` et `b` :
  `(a<b) ? c=a : c=b ;`
- Ajouter `'s'` en cas de pluriel :

```
int x;
x=...
printf("J'ai trouvé %d élément%c\n",x,(x>1) ? 's' : '');
```

Les instructions

Les instructions répétitives

Deux instructions répétitives "équivalente" : `while` et `for`

```
while(<expression>)
    <instruction>
```

Quatre parties

1. conditions préparatoires
2. bloc répété
3. changement d'"état"
4. test (d'entrée/de reprise/de sortie)

```
int i=1, n=100;
while(i<=n){
  if n%i == 0
    printf(''%d'',i);
  i++;
}
```

Les instructions

Les instructions répétitives

Deux instructions répétitives "équivalente" : `while` et `for`

```
for(<expression1>; <expression2> ; <expression3>)
   <instruction>
```

Quatre parties

1. conditions préparatoires
2. bloc répété
3. changement d'"état"
4. test (d'entrée/de reprise/de sortie)

```
int i, n=100;
for(i=1;i<=n;i++){
  if n%i == 0
    printf(''%d'',i);
}
```

Les instructions

Les instructions répétitives

Il existe une autre instructions répétitives exécutée au moins une fois !

```
do
    <instruction>
while (condition)
```

Exemple

```
char rep;
do{                /* ici un traitement au moins */
   printf("On continue? O/N\n");
}
while( (rep=getchar()) != 'N') ;
```

Remarque : `getchar()` permet de lire un caractère au clavier

Un petit bilan

- Un programme : suite d'instructions
- Fichier source $\rightarrow$ compilation $\rightarrow$ fichier objet $\rightarrow$ programme executable
- Des fonctions : `main()`, `atoi()`, `printf()`, `laVotre()`
- Réservation d'une zone mémoire avec une étiquette
- Typages des variables (taille zone, mémoire, controles de l'usage)
- Instructions et bloc d'instructions { }
- Opérateurs : relationnels, logiques, affectations, in/dé-crémentation
- Forme algorithmiques
  - Conditionnel : `si <condition> alors ... sinon... (if)`
  - Répétitifs :
    `tant que <condition> faire ... (while, for)`
    `Faire ... jusqu'à (do ... while)`

Les fonctions

Les fonctions

Structure d'une fonction

```
<type> <nom> <liste de paramètres>
<bloc de définition>
```

- `<type>` : type de la valeur retournée par la fonction (`int` par défaut)
- `<nom>` : nom (explicite) de la fonction en commençant par une minuscule
  la fonction `<nom>` est du type `<type retourné>`
- `<liste de paramètres>` :
  - entre (et)
  - pour chaque paramètre : `<type> <nomDeVariable>`
  - les paramètres sont séparés par une virgule `,`
- `<bloc de définition>` : liste d'instructions entre { et }

  si le type de la fonction **n**'est **pas** `void`, au moins une instruction :
  `return <valeur ou variable>;`

  ATTENTION `return` stoppe l'exécution de la fonction et renvoie la valeur indiquée (Rq : `return ;` ne renvoie rien (`void`))

Exemple :

```
float discriminant(float a, float b, float c){
    return b*b-4*a*c;
}
```

Les fonctions

Déclaration et définition d'une fonction

- La déclaration = prototype, signature
  - **sans** le bloc de définition, **avec** un point virgule ;
  - Après la déclaration : possibilité de l'utiliser
    = l'appeler, invoquer son nom dans le code

```
<type> <nom> <liste de paramètres> ;
```

- La définition : **avec** le bloc de définition

```
<type> <nom> <liste de paramètres>
<bloc de définition>
```

Les fonctions

L'appel de fonction : `variable = nomFonction(param1, param2, etc);`

- `variable` : facultatif (le retour n'est pas utilisé ou fonction de type `void`)
- `variable` est du type de la fonction `nomFonction`
- il y a autant de paramètres effectifs que de paramètres formels avec respect des types et de l'ordre

Rq : le programme appelant la fonction est la fonction qui contient l'appel

Exemple

```
float discriminant (float a, float b, float c);  /* déclaration */
float discriminant (float a, float b, float c){  /* définition  */
    return  b*b - 4*a*c;
}

void MaF( ){                                    /* programme appelant */
    float result, v=1 , fxp = 12;          /* déclarations variables*/
    result = discriminant (14, v , fxp);    /* appel de fonction */
}                                           /* puis affectation  */
```

Les fonctions

Remarque importante sur la déclaration

- La déclaration n'est pas obligatoire

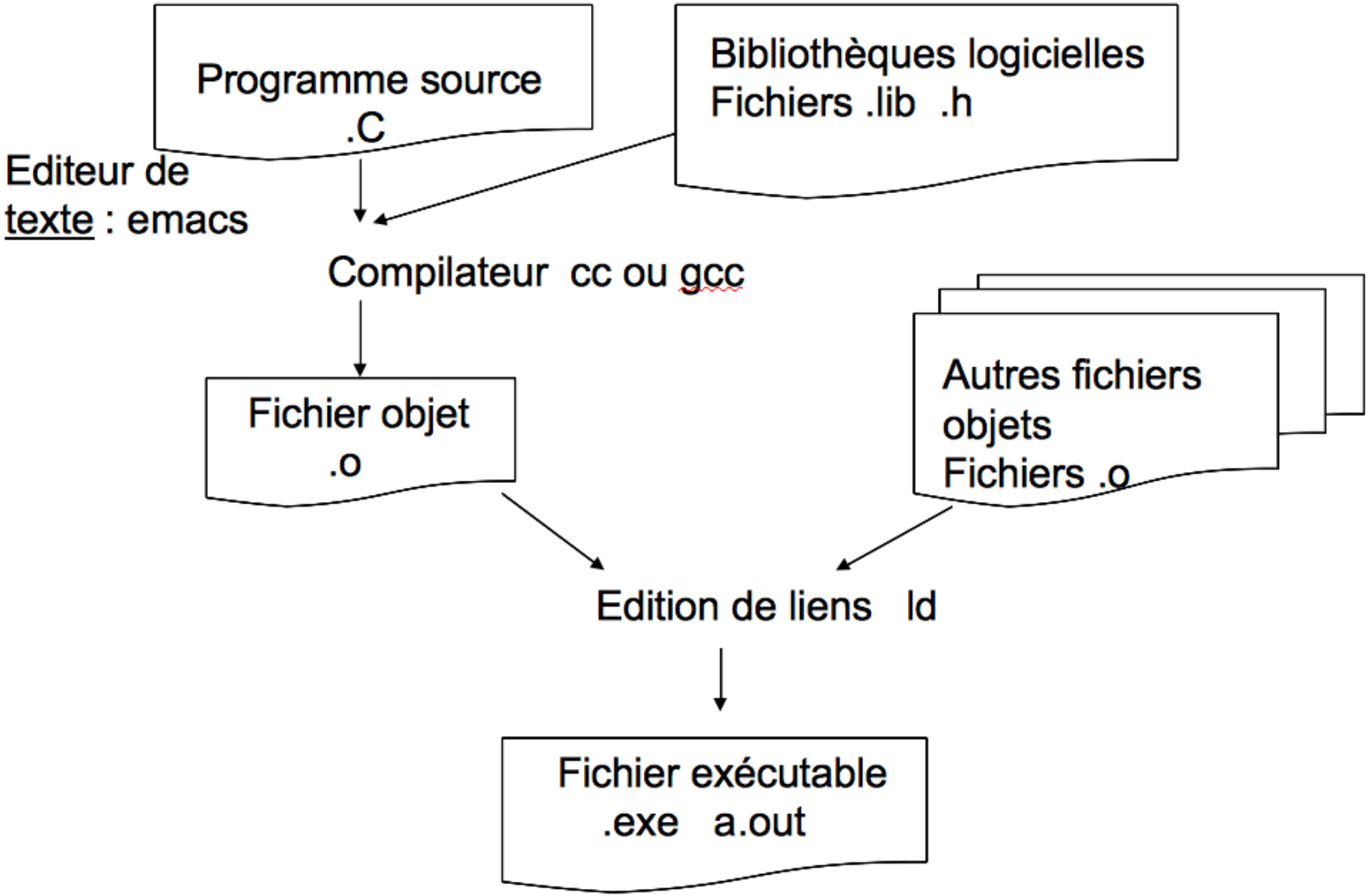
  MAIS parfois utile et parfois nécessaire

- **Utile** : le programme appelant (par exemple le main) est placé avant l'écriture de la fonction (question de composition du fichier .c)

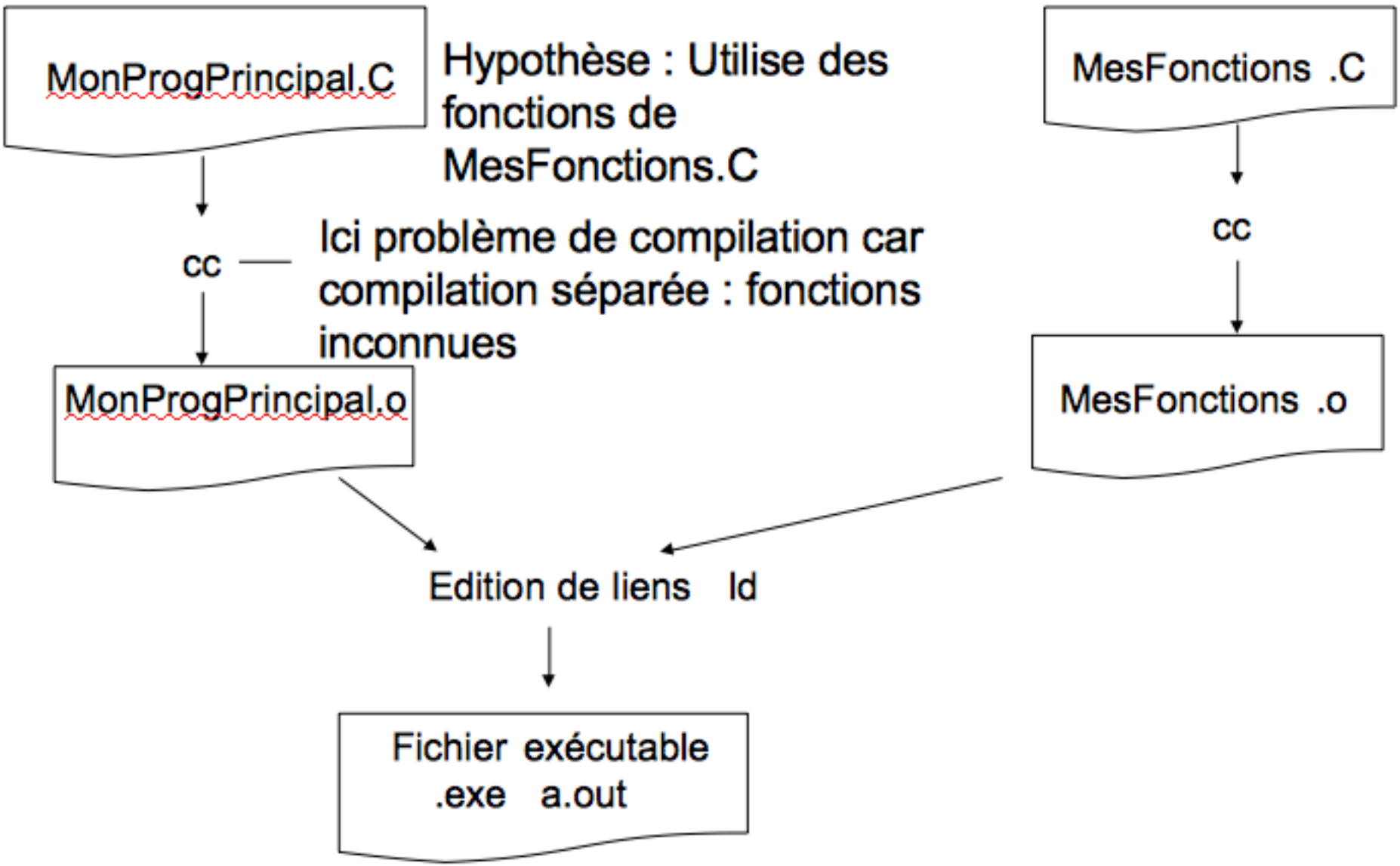
- **Nécessaire** : le programme appelant utilise une fonction d'un autre module (autre .c, .o)

  La compilation (qui est faite module par module) oblige à connaitre la déclaration de la fonction (type retourné, types paramètres)

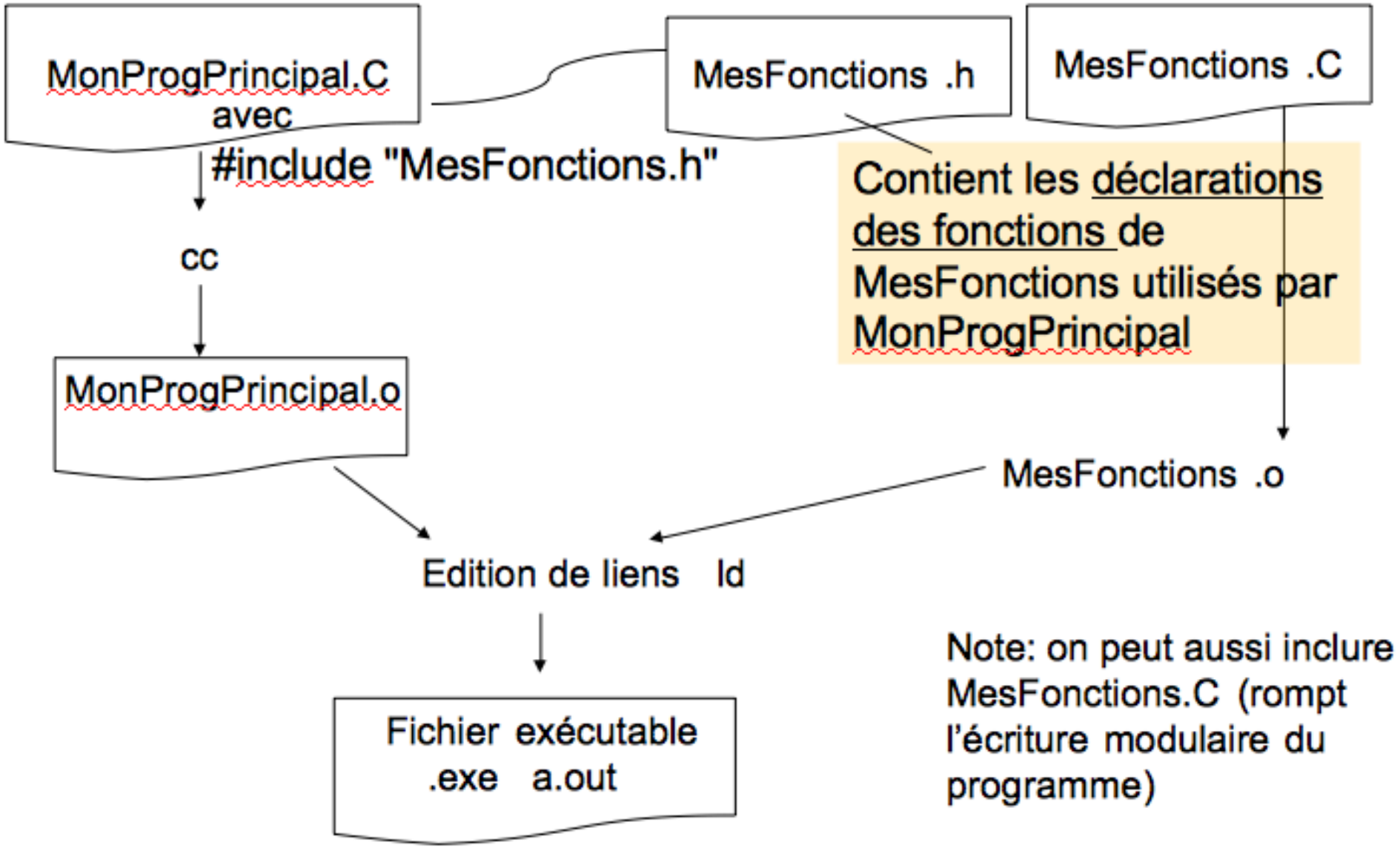
Schéma de production de logiciel C



Programme principal et module(s)



Programme principal et module(s)



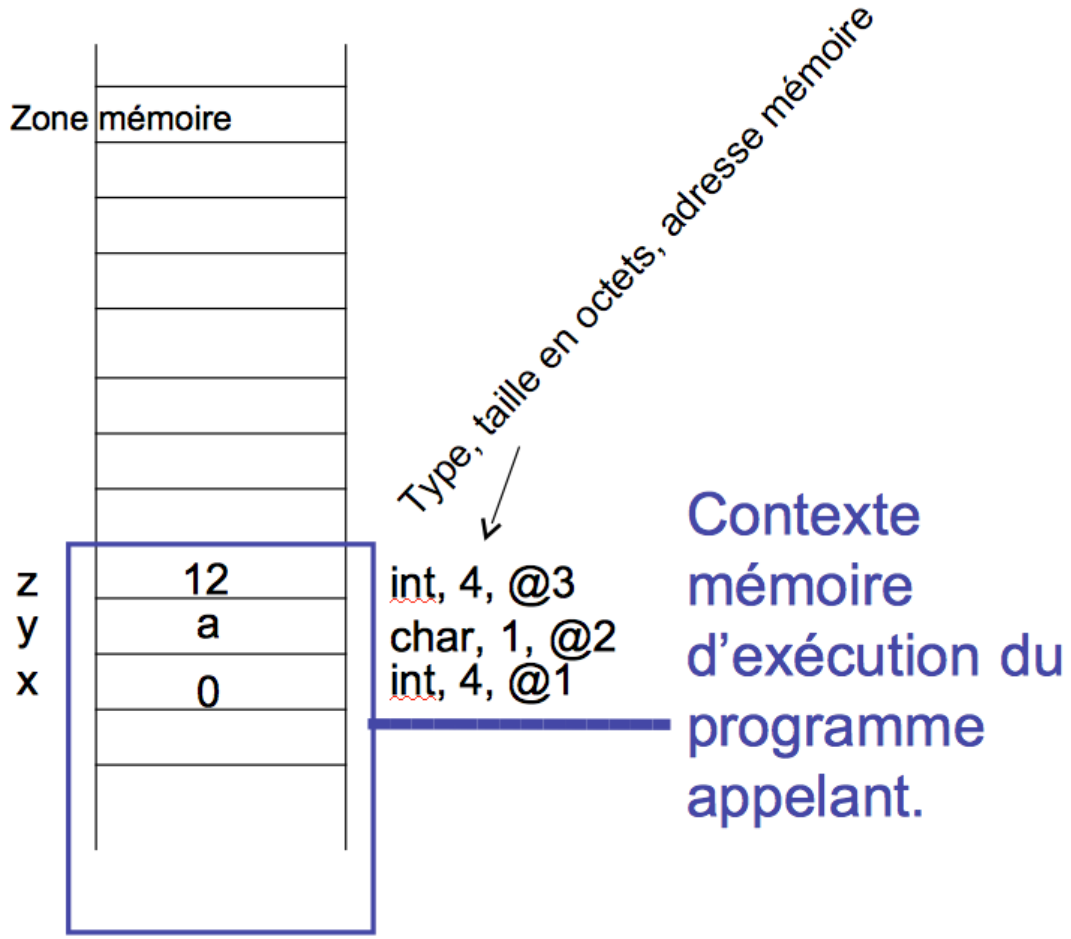
Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

Avant l'appel de `maFonction ()`



Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

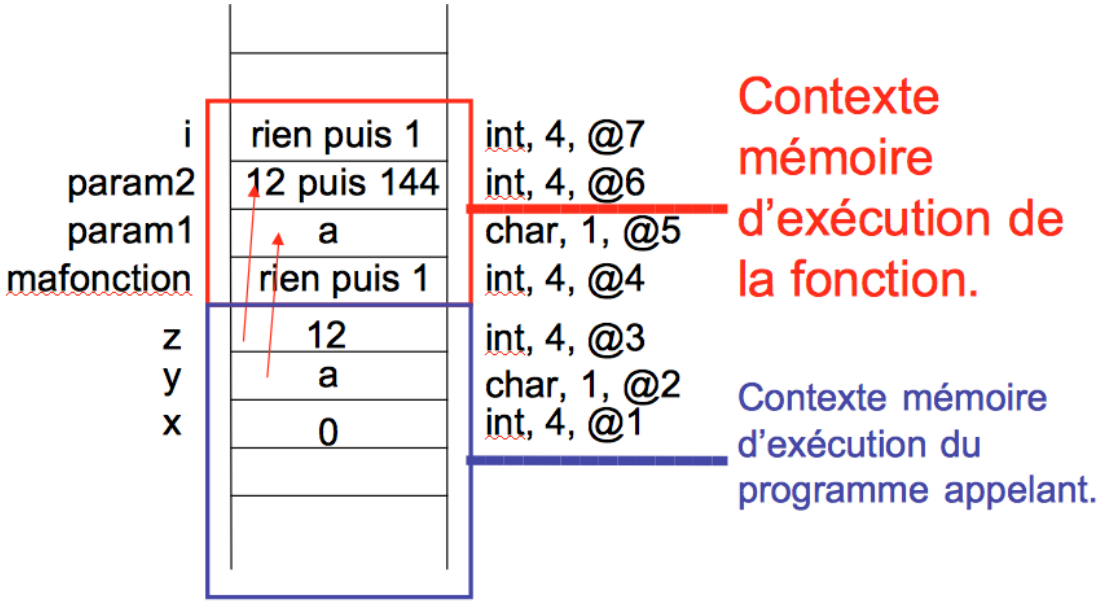
```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

Exécution de `x = maFonction (y, z);`

L'appel de la fonction $\Rightarrow$ changement de contexte d'exécution

1. Copie des valeurs données aux paramètres `param1` $\leftarrow$ `y` et `param1`$\leftarrow$ `z`
2. Exécution des instructions



Paramètres de fonction et zone mémoire

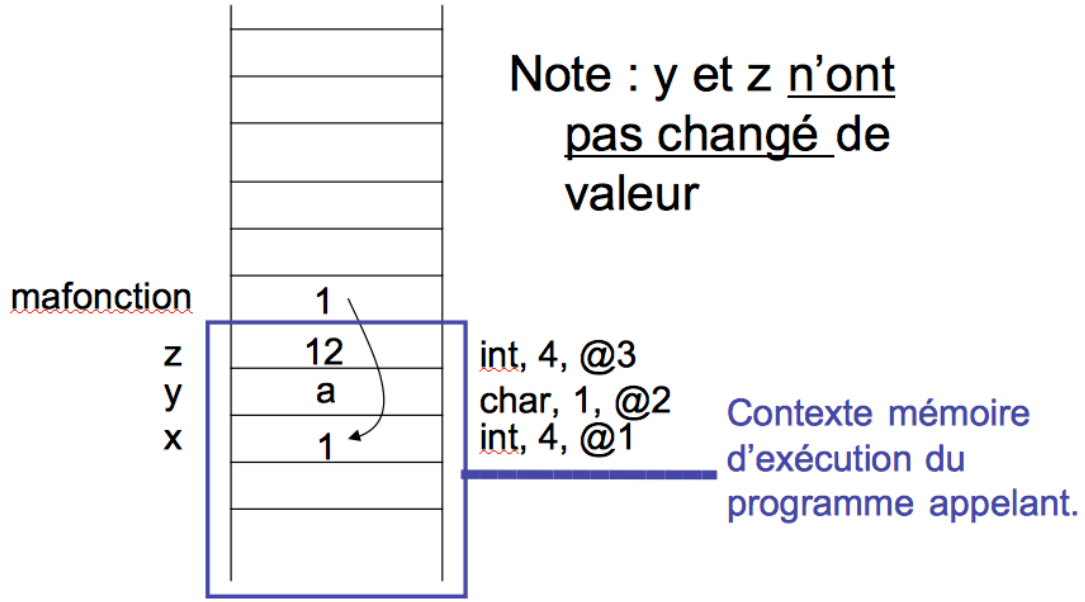
Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

après l'exécution de `x = maFonction (y, z);`

Retour au programme appelant
$\Rightarrow$ retour au précédent contexte d'exécution
avec copie du résultat de `x=maFonction(y,z)`



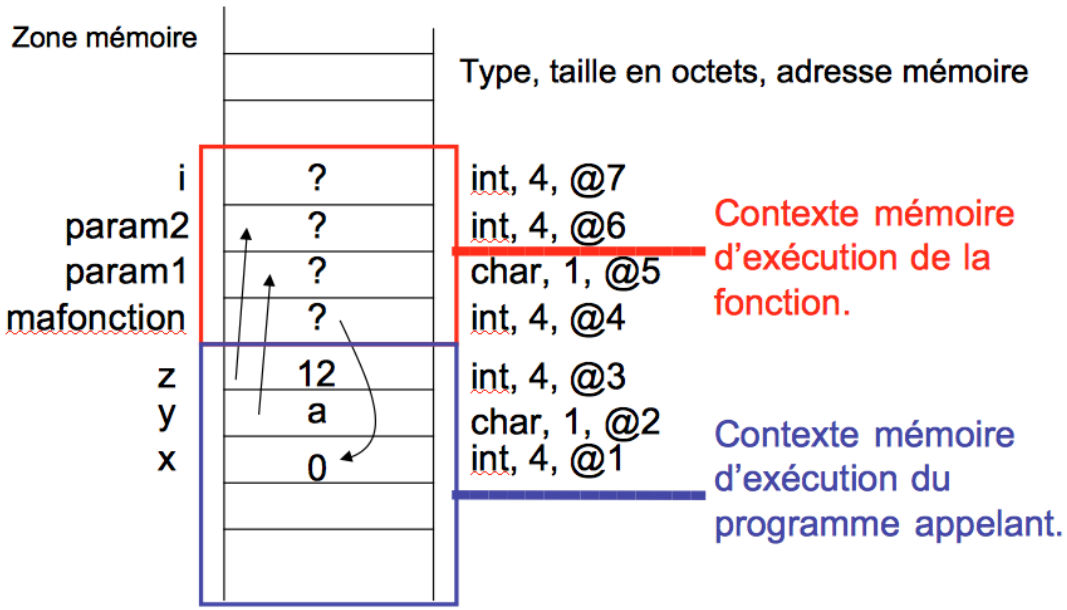
Paramètres de fonction et zone mémoire

Le passage de paramètres entre un programme appelant et une fonction $\Rightarrow$ Nécessite de s'intéresser à "ce qu'il se passe en mémoire"

```
int maFonction(char param1,int param2)
{
     int i=1;
     printf("%c", param1);
     param2= 144;
     return i;
}

main ( ){
    int x=0;
    char y = 'a';
    int z=12;
    x = maFonction (y,z);
}
```

Synthèse

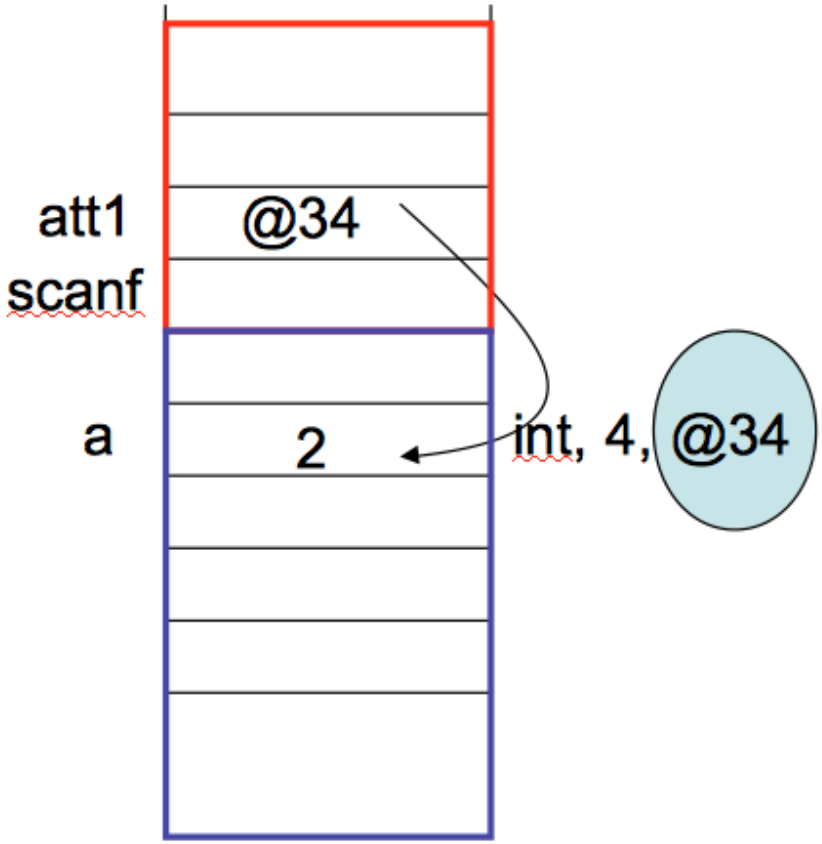


Paramètres de fonction et zone mémoire

Retour sur la fonction `scanf()`

Exemple d'utilisation :

```
int a; scanf (‘‘%d’’, &a );
```



Passage de paramètres par valeur

- **Recopie** de la valeur de la variable (ou expression) dans une variable locale à la fonction
- La variable locale est utilisée pour faire les calculs dans la fonction
- Aucune modification de la variable locale n'est répercuté dans la fonction appelante

Exemple

```
void test(int k){ /* k est la copie de la valeur
                     passée en paramètre          */
    k=k+3;        /* Modifie k,                   */
}                   mais pas la variable fournie par l'appelant */
int main(void){
    int i=2;
    test(i);        /* Le contenu de i est copié dans k.
                       i n'est pas modifié. Il vaut toujours 2. */
    test(2);        /* La valeur 2 est copiée dans k. */
    exit(0);
}
```

Passage de paramètres par référence/adresse

- Pour pouvoir répercuter la modification de la valeur dans la fonction appelante, il faut faire appel aux pointeurs !

$\Rightarrow$ Passage de paramètre par référence/adresse

- L'adresse de la variable à modifier est passée en paramètre : `&nom_var`
- Dans la fonction, pour modifier la valeur on utilise `*nom_variable`

Exemple

```
void test(int *pj){ /* test attend l'adresse d'un entier... */
    *pj=*pj+2;      /* ... pour en modifier sa valeur. */
}
int main(void){
    int i=3;
    test(&i);       /* On passe l'adresse de i en paramètre. */
                    /* Ici, i vaut 5. */
    exit(0);
}
```

N.B. : Les tableaux sont des pointeurs ! ! !

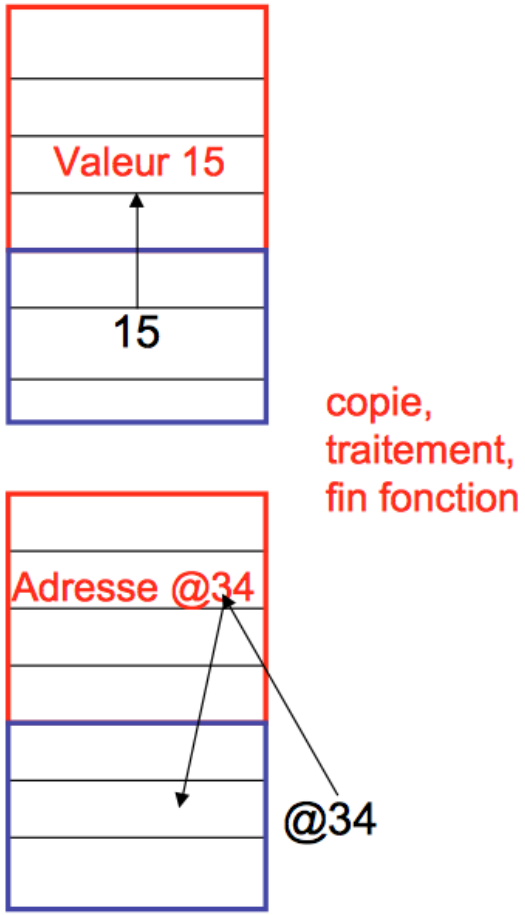
Paramètres de fonction et zone mémoire

Un petit bilan

Passage de paramètres

- soit par valeur
  - Copie de la valeur
  - Pas d'accès à la variable initale copiée
  - Modification possible de la valeur locale

- soit par adresse (avec le signe &)
  - Copie l'adresse
  - Accès (modification possible) à la variable initiale via l'adresse



Paramètres de fonction et zone mémoire

Retour sur la fonction `atoi()` (et `atof()`)

Rappel

```
int atoi(char* param);
```

fonction prédéfinie dans `<stdio.h>`

Conversion de la chaîne de caractères `param` en **entier**

Remarque :

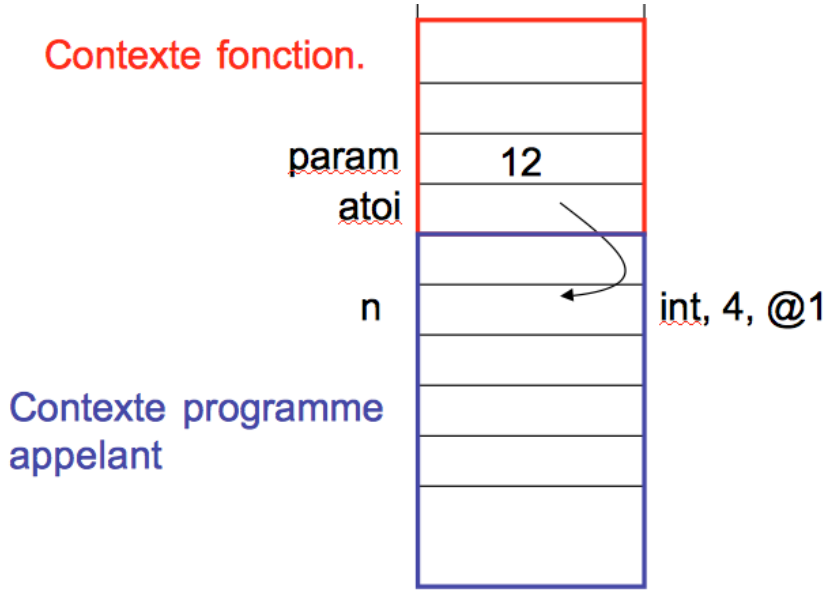
```
double atof(char* param);
```

Fonction prédéfinie dans `<stdio.h>`

Conversion de la chaîne de caractères `param` en **réel**

Exemple :

```
int n; n=atoi(''12'');
```



Paramètres de fonction et zone mémoire

Retour sur la fonction `printf()`

Rappel

`int printf(char* param1{,liste});`

fonction prédéfinie dans `<stdio.h>`

Affiche la chaîne de caractères `param1` en la formatant selon le(s) format(s) spécifié et les expressions calculées

Remarque :

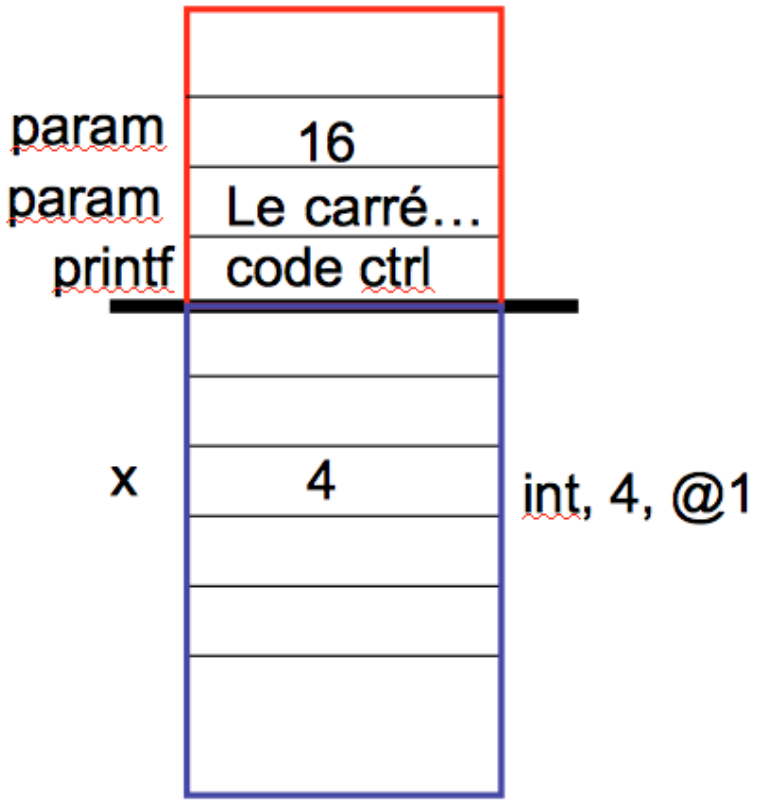
`printf()` retourne un entier

il correspond au nombre d'octets écrits

ou à la constante `EOF` (-1) en cas d'erreur

Exemple :

```
int x=4;
printf(‘‘Le carré de
    x est % d \n’’,x*x);
```



Paramètres de fonction et zone mémoire

Découverte de la fonction `getchar()`

```
char getchar();
```

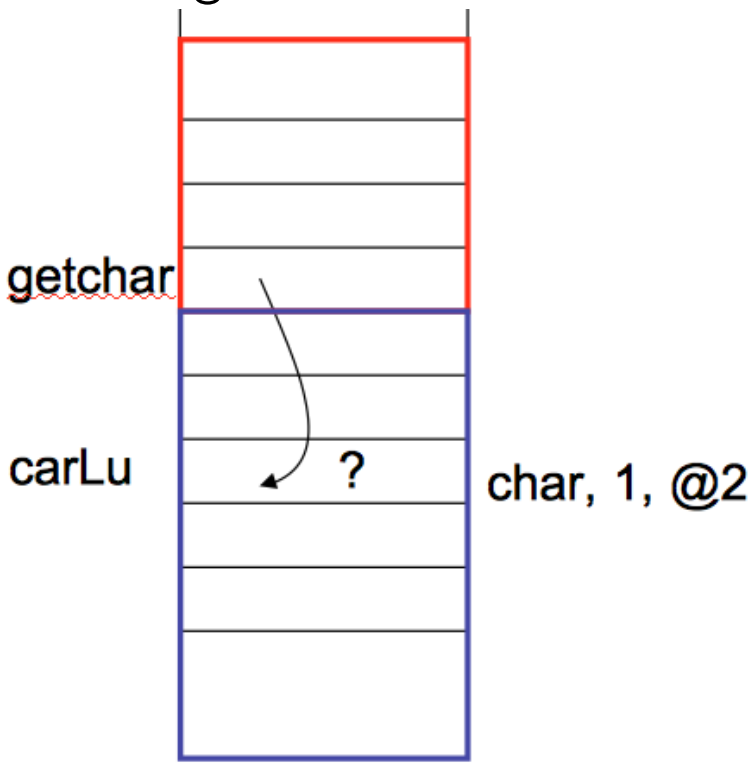
fonction prédéfinie dans `<stdio.h>`

Lit **un** seul caractère entré au clavier

C'est une fonction bloquante

Exemple :

```
char carLu;
carLu = getchar();
```



Paramètres de fonction et zone mémoire

Retour sur la fonction `scanf()`

À lire : `https://openclassrooms.com/courses/la-saisie-securisee-avec-scanf`

Rappel

```
int scanf(char* format, att1, att2, att3,...);
```

fonction prédéfinie dans `<stdio.h>`

Lit des données rentrées par l'utilisateur

Remarque : `scanf()` retourne un entier

il correspond au nombre de variables lues ou à la cste `EOF` en cas d'erreur

Exemple d'utilisation :

```
int a;
scanf (''%d'', a );   FAUX !!
scanf (''%d'', &a );  Correct ! mais pourquoi ?
```

&a correspond à l'adresse de la variable a

En fait, on veut que la valeur de a change ! !

Les tableaux

Notions de bases

Les tableaux - notions de bases

Définition

Représentation tabulaire des données de **même type** (liste, matrice)

Dimension au plus 2

Indicé par des entiers de 0 à `nbElements`-1

Les tableaux - notions de bases

Déclaration

```
<type> <nom>[<nbElements>];
```

Exemple

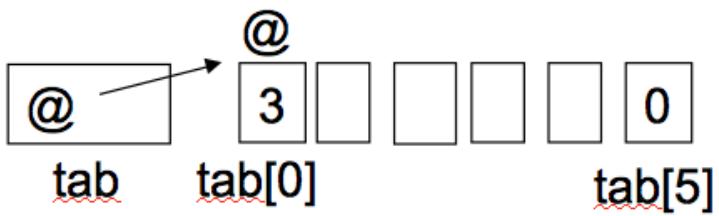
```
int tab[6];
```

Déclare la variable `tab` comme un tableau de 6 entiers

`tab` adresse de début du tableau

`tab[0]` valeur du premier entier `tab[0]=3;`

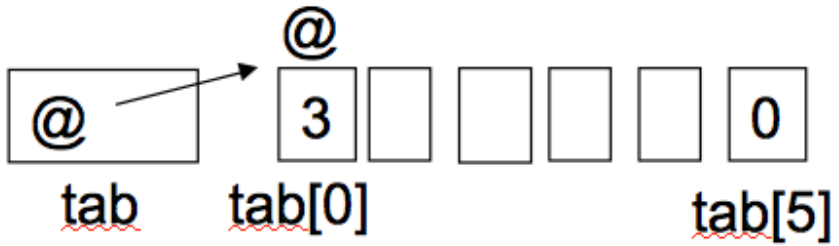
`tab[5]` valeur du dernier entier `tab[5]=0;`



Les tableaux - notions de bases

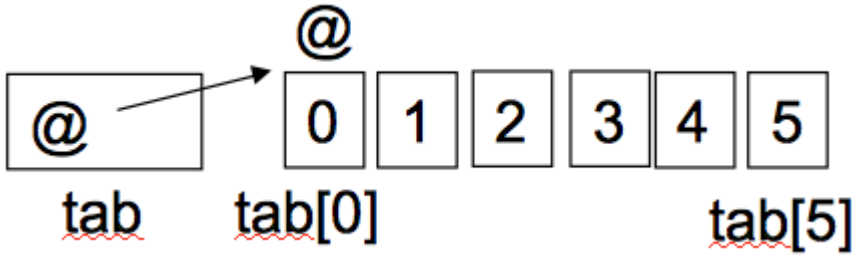
Exemples basiques d'utilisations des tableaux

```
int tab[6];      /* declaration d'un tableau
                    d'entier de taille 6   */
```




```
int i = 6, j ;
tab[0] = 3;
tab[5] = 0;
tab[i] = 14;   /* instruction possible mais a de grandes chances
                  de provoquer une erreur à l'exécution */
```

```
for (j=0 ; j < i ; j++)
     tab[j] = j ;
```



Les tableaux - notions de bases

Quelques propriétés importantes

- La taille du tableau DOIT être connue $\Rightarrow$ prévoir la taille maximale utile !
  On peut utiliser `#define` `#define TAILLE_MAX 300`

- **ATTENTION :** **pas** d'affectation entre tableaux ! (pas de ~~`tableau1 = tableau2`~~)

$\Rightarrow$ Il faut prévoir des boucles de recopie

- `double tab[TAILLE_MAX]; /* declare un tableau de TAILLE_MAX double */`

  `tab` est l'adresse du début du tableau

  `tab[i]` désigne l'élément (double) de rang `i`

  `tab[i]` est défini pour `0 <= i < TAILLE_MAX`

  `i` indique le décalage par rapport au début du tableau
  $\Leftrightarrow$ calculer un décalage de `i` double "au delà" de l'adresse de `tab`

Les tableaux - notions de bases

Remplissage d'un tableau

Lorsque l'on déclare un tableau, il est "vide" (aucune valeur n'a été affectée aux cases)

$\Rightarrow$ Il faut donc le remplir

Exemple

- un entier `N` , un tableau `tab` de `N` "cases" (vides)
- Traitement : affecter aux `N` cases de `tab` les sommes partielles $(0, 1, 3, 6, 10, 15, 21, \textit{etc.})$

```
#DEFINE N 144
int t[N];
int i;
t[0] = 0;
for (i=1 ; i < N ; i++)
     t[i] = t[i-1] + i ;
```

Exercice : ré-écrire la bout de code précédent à l'aide d'une boucle `while`

Les tableaux - notions de bases

Exemples de déclarations (1/2)

- `int vecteur[100];` tableau à 1 dimension, taille 100
- `float matrice[10][10];` tableau à 2 dimensions
- `#define MAX 100`

  `char prenom1[MAX];` chaîne (tableau) de caractères, taille `MAX`
- `char prenom2[] = "Jean";` chaîne de caractères initialisée à

'J'	'e'	'a'	'n'	'\0'

- `char prenom3[];` chaîne de caractères vide

  ATTENTION : ici ni taille, ni valeur

⇒ une réservation spécifique de mémoire sera nécessaire pour entrer des valeurs

Exemples de déclarations (2/2)

- On peut aussi définir un nouveau type `tab` ou `mat`

```
#define MAX 100
typedef int tab[MAX];
typedef int mat[MAX][MAX];
```

Puis déclarer une variable d'un de ces types :

```
tab vecteur;
mat matrice;
```

Une aparté sur la définition de types (`typedef`)

`typedef <déclaration>;`

↪ Permet de définit un nouveau type

Exemple :

```
typedef int kilometre; /* définit le type kilometre */
        kilometre distance = 4;
```

Les tableaux - notions de bases

Exemples d'algorithmes simples : La fonction `afficher`

Entrée : un entier `n`, un tableau d'entiers `t`
Sortie : affichage des `n` premiers éléments de `t`

ATTENTION : on suppose que `t` contient au moins `n` éléments

```
void afficher (int n, tab t){
    int i;
    for(i=0 ; i < n ; i++)
       printf("%d ",t[i]);
    printf("\n");
}
```

Les tableaux - notions de bases

Exemples d'algorithmes simples : La fonction `afficher_inverse`
Entrée : un entier `n`, un tableau d'entiers `t`
Sortie : affichage *inversé* des `n` premiers éléments de `t`

ATTENTION : on suppose que `t` contient au moins `n` éléments

```
void afficher_inverse (int n, tab t){
    int i;
    for(i=0 ; i < n ; i++)
       printf("%d ",t[(n-1)-i]);
    printf("\n");
}
```

OU

```
void afficher_inverse (int n, tab t){
    int i;
    for(i=n-1 ; i >=0  ; i--)
       printf("%d ",t[i]);
    printf("\n");
}
```

Les tableaux

Chaînes de caractères

Les tableaux - chaînes de caractères

Chaînes de caractères = tableau de caractères

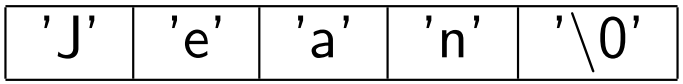
- se termine par le caractère nul '\0'

$\Leftrightarrow$ le premier caractère du code ASCII (dont la valeur est 0)

c'est un caractère de contrôle (non affichable) qui indique la fin d'une chaîne de caractère

$\Rightarrow$ Une chaîne composée de n éléments est en fait un tableau de $n+1$ éléments de type `char`

Rq : La chaîne débute à une adresse



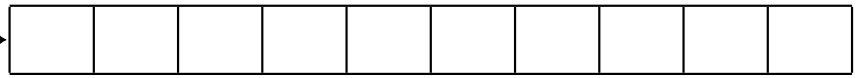
Rq : `char *argv[]` est un tableau de chaîne de caractères

Les tableaux - chaînes de caractères

Déclaration

- `char prenom[10];`

⇒ réserve 10 caractères : 9 explicitement utilisés + 1 pour le caractère de fin

prenom @→ 

Exemples d'initialisation de la chaîne

```
#include <stdio.h>
void main(){
  char chaine[10];
  chaine[0]= 'J';
  chaine[1]= 'e';
  chaine[2]= 'a';
  chaine[3]= 'n';
  chaine[4]= '\0';
}
```

```
#include <stdio.h>
void main(){
   char chaine[10]={'B','o','n','j','o','u','r','\0' };
}
```

- `char nom[]=''ZOLA'';`

⇒ Taille indiquée par le nombre de caractères de la chaîne (+1)

Place automatiquement le '\0'

nom @→ 

Les tableaux - chaînes de caractères

Attention à la gestion des chaînes

- Bien gérer les indices :
  début à 0 et fin à taille-1, incrémenter pour parcourir le tableau, etc.

- Travailler avec une zone mémoire réservée
  i.e., un nombre suffisant d'éléments

  Rq : `char maChaine[];` ou `char * maChaine;`
  ne reserve que le "pointeur vers", MAIS pas la zone pour les caractères

⇒ Le nombre d'éléments est ici inconnu !

- Insérer soi-même '\0' en fin de chaîne (si ce n'est pas fait automatiquement)

- **Pas d'affectation "globale" du type** ~~`maChaine1 = maChaine 2;`~~

Exemple d'un algorithme simple

Recopie de la chaîne source vers la chaîne cible

```
#include<stdlib.h>
int main(){
   char source[]="ZOLA";
   char dest[10];                  /* on travaille avec une zone mémoire réservée */
   int i=0;                        /* on gère les indices !! */
   while (source[i] != '\0'){
       dest[i]=source[i];
       i++;                        /* on incrémente pour parcourir le tableau */
   }
   dest[i]='\0';                   /* on insère le caractère de fin de chaine */
   exit(0);
}
```

Attention au piège !

Il faut recopier caractère par caractère ! (ou une fonction spéciale)
Sinon, c'est l'adresse que l'on recopie !

```
char rire[]="ha ha";
char *rireBis;            /* equivalent à char rireBis[] */
rireBis = rire;           /* copie de l'adresse de rire */
```

Les tableaux - chaînes de caractères

Fonctions prédéfinies

`string.h` contient des fonctions dédiées à la manipulation des chaines
`#include <string.h>` (Rappel : chemin par défaut `/usr/include`)

strcpy	stricmp	strpbrk	strcat	strlen	strrchr	strchr
strlwr	strrev	strcmp	strncat	isxdigit	strcmpi	strncmp
strset	strcpy	strncmpi	strstr	strcspn	strncpy	strtok
strdup	strnicmp	strupr	strerror	strnset		

str : string cmp : compare n : n premiers caractères len : lenght
i : ignore la casse cpy : copie cat : concaténation chr : char r : reverse

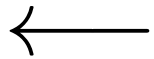
IMPORTANT

Ne pas apprendre la liste des fonctions et paramètres par cœur
MAIS savoir les chercher et les utiliser correctement

Les tableaux - chaînes de caractères

Recopie de chaîne

```
char *strcpy(char *cible, char *source);
```

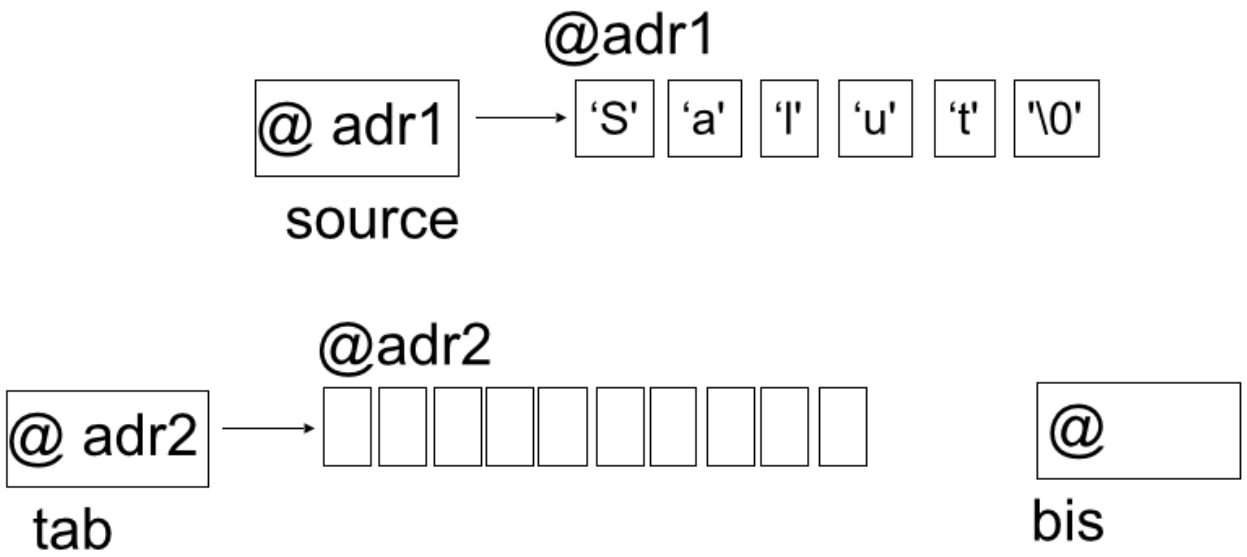


- Recopie de la chaîne de caractère qui commence à l'adresse de source dans la chaîne qui commence à l'adresse de cible
- **ATTENTION :** il faut avoir réservé la place mémoire (*i.e.*, il faut correctement déclarer les chaînes)
- Le retour de la fonction "pointe" (est l'adresse) de la chaîne cible
- Le '\0' est placé en fin de chaîne

Les tableaux - chaînes de caractères

Recopie de chaîne - Un exemple (1/3)

```
char *source = "Salut" ;
 /* déclaration et initialisation d'une chaîne de taille 6 */
char tab[10] ;
 /* déclaration d'une chaîne de taille 10 */
char bis[] ;
 /* déclaration d'une chaîne sans réservation de mémoire */
```



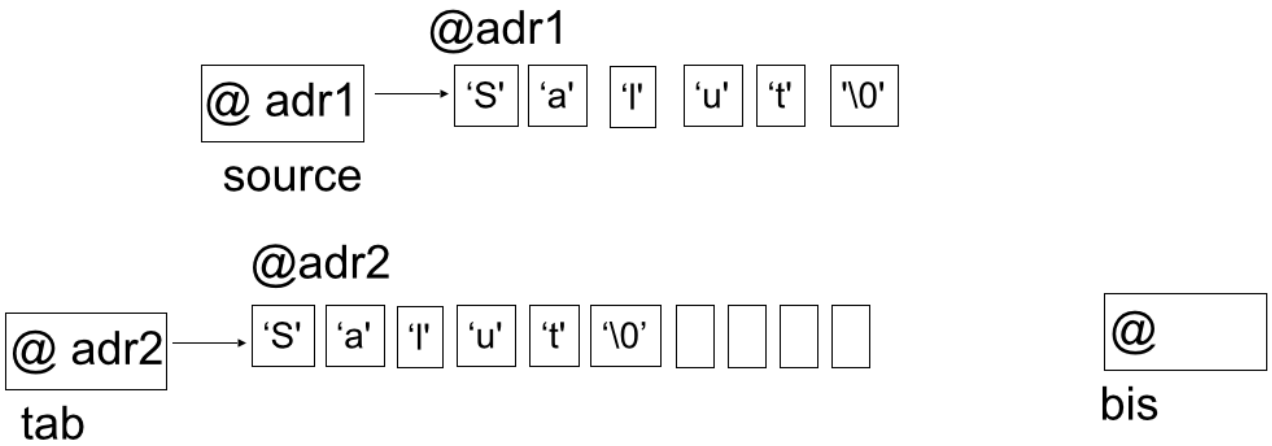
Recopie de chaîne - Un exemple (2/3)

```
char *source = "Salut", tab[10], bis[] ;
strcpy(tab,source);
```

Remarques

Ici, on veut copier la chaîne qu'il y a dans `source` dans la chaîne `tab`

On peut le faire puisque la taille de la chaîne est bien < 10
On n'utilise pas le retour de `strcpy()`



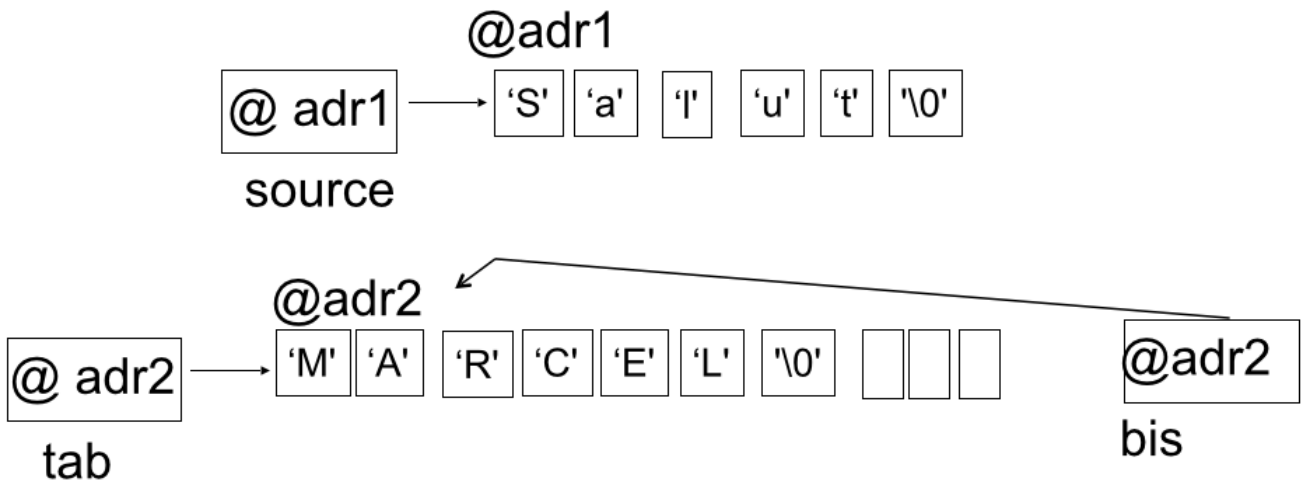
Recopie de chaîne - Un exemple (3/3)

```
char *source = "Salut", tab[10], bis[] ;
bis = strcpy(tab,"MARCEL");
```

Remarques

Ici, on veut copier la chaîne ‘‘MARCEL’’ dans la chaîne `tab`
ET récupérer l'adresse de `tab` dans bis

On peut le faire puisque la taille de la chaîne est bien < 10



Les tableaux - chaînes de caractères

Longueur de chaîne

`int strlen(char *chaine);`

↪ Retourne la longueur de la chaîne pointée par `chaine`

ATTENTION

`strlen()` ne compte pas le caractère de fin de chaîne '\0' ! !

Exemple

```
char *chaine = "Salut";
int taille;
taille = strlen(chaine);
printf("La taille de la chaîne \"%s\" est : %d\n",chaine,taille);
```

Affiche :

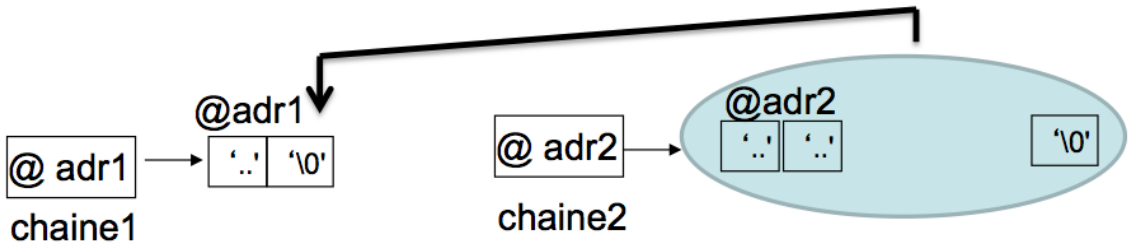
```
La taille de la chaîne ‘‘Salut’’ est : 5
```

Les tableaux - chaînes de caractères

Concatenation de chaînes

```
char *strcat(char *chaine1, char *chaine2);
```

- Recopie la chaîne pointée par `chaine2` **à la fin** de la chaîne pointée par `chaine1`



- Le résultat est pointé par `chaine1` et est retourné (pointeur sur la chaîne)

Remarques

1. La place réservée pointée par `chaine1` doit être suffisante !
   `strlen(chaine1) + strlen(chaine2)` < zone mémoire réservée pour `chaine1`
2. Le '\0' est (dé)placé en fin de chaîne

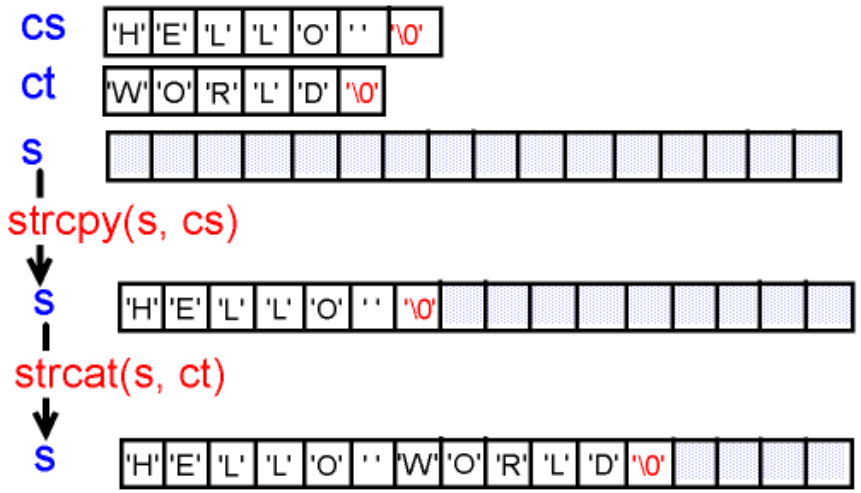
Concatenation de chaînes - Deux exemples

Exemple 1

```
char r[20], s1[20], s2[] = "nation";
strcpy(s1,"concaté");        /* copie de la chaîne concaté
                                dans S1 de taille > à 7 ! */
r=strcat(s1, s2);            /* concaténation de S1 et S2 */
printf("%s", r);             /* Affiche "concaténation" */
printf("%s", s1);            /* Affiche "concaténation" */
```

Exemple 2

```
char cs[] = "HELLO ";
char ct[] = "WORLD";
char  s[16];
strcpy(s,cs);
strcat(s,ct);
```



Les tableaux - chaînes de caractères

Comparaison de chaînes

```
int strcmp(char *chaine1, char *chaine2);
```

- Comparaison des chaînes pointées caractère par caractère
- Retourne un entier négatif, nul ou positif, selon le classement alphabétique des 2 chaînes pointées

Exemple

```
char *s1 = "abcd", *s2 = "abz";
if ( strcmp (S1, S2) == 0 ){
    printf("Les deux chaines sont identiques\n'');
else if ( strcmp (S1, S2) < 0 ){
    printf("%s est inférieure à %s\n", s1, s2);
}
```

Les tableaux - chaînes de caractères

Des fonctions utiles sur les chaînes (dans `stdio.h`)

On a déjà vu

- `int printf(char* param1{,liste});`

↪ pour afficher une chaîne

- `int scanf(char* format, att1, att2, att3,...);`

↪ pour lire une chaîne(s) formatée(s) **(tout "espace" est délimiteur)**

Une nouvelle fonction : `char *gets(char *s);`

- Lit une chaîne **terminée par** '`\n`' sur l'entrée standart
- Le résultat est pointé par `s` et est retourné (renvoie `NULL` en cas d'erreur)
- '`\n`' n'est pas recopié dans la chaîne et un '`\0`' est placé à la fin de la chaîne

Rq : `scanf("%s",s);` ne permet pas de lire des chaînes contenant des espaces (l'espace est pour `scanf` un séparateur), tandis qu'avec gets, seul le caractère '`\n`' sert de délimiteur

Les tableaux - chaînes de caractères

Exemple d'utilisation de `char *gets(char *s);`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
  char ligne[10];
  printf("Votre texte 1 ? ");
  gets(ligne);
  printf("Vous avez rentré le texte : %s\n",ligne);
  exit(0);
}
```

Résultat de l'exécution

```
>./monprog
Votre texte 1 ? Salut
Vous avez rentré le texte : Salut
>
```

ATTENTION à l'espace mémoire réservé pour la chaîne... Par exemple :

```
>./monprog
Votre texte 1 ? qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
Segmentation fault (core dumped)
```

Un petit bilan

- Tableau : regroupement de données de **même nature**

- `<type> <nom>[<nbElem>];`

↪ Déclaration d'un tableau de taille `<nbElem>`

- `<type> <nom>[<nbElem1>][<nbElem2>];`

↪ Déclaration un tableau à 2 entrées *ie* une matrice

- `<type> <nom>[];` ou `<type> *<nom>;`

↪ Réserve uniquement le pointeur vers le tableau (pas de réservation de zone mémoire)

- Les indices d'un tableau varient de 0 à `<nbElem>`-1

  `<nom>[0]` est la première case du tableau — `<nom>[<nbElem>-1]` est la dernière

- Les chaînes de caractères sont des tableaux de caractères
  `char chaine[<taille>];` ou `char *chaine;` ou `char chaine[];`

- Les chaînes se terminent par le caractère '\0'

- de nombreuses fonctions pour manipuler les chaînes sont pré-définies dans la bibliothèque `string.h`

Structure de données

Notions de bases

Structure de données - Notions de bases

Définition

Assemblage de données de types éventuellement distincts

Motivation : Regrouper des informations de types différents

Exemple

Un étudiant est donnu par :

- un nom
- un (ou des) prénom(s)
- une adresse
- une date de naissance
- l'année d'inscription
- ...

Structure de données - Notions de bases

Déclaration

`struct <nom>{ <suite de déclarations> } ;`

Exemple

```
struct entier_de_Gauss{
    int reelle;
    int imaginaire;
};
```

Définit **le nouveau type**

`struct entier_de_Gauss`

comme étant la liste de déclarations

`{int reelle; int imaginaire;}`

Structure de données - Notions de bases

Utilisation

- `struct entier_de_Gauss x;`

↪ Déclare une variable `x` de type `struct entier_de_Gauss`

⇒ il est préférable de **définir un nouveau type** via le mot-clé `typedef`

```
typedef struct entier_de_Gauss entier_de_Gauss;
```

⇒ Du coup : `entier_de_Gauss x;`

↪ Déclare une variable `x` de type `entier_de_Gauss`

- Accès aux champs de la structure : `<nom_variable>.<nom_champ>;`

```
x.reelle = 0;          x.imaginaire = 1;
```

- Accès aux champs lorsque l'on manipule des pointeurs/adresses : `<nom_pointeur> -> <nom_champ>;`.

Un exemple complet avec les entiers de Gauss

```
#include <stdlib.h>
#include <stdio.h>
struct entier_de_Gauss{
   int reelle;
   int imaginaire;
} ;
typedef struct entier_de_Gauss entier_de_Gauss;
void usage(char *nom){
   printf("usage : %s <partie reelle x1> <partie imaginaire x1> ", nom);
   printf("<partie reelle x2> <partie imaginaire x2>, tout en <entier>\n"
}
entier_de_Gauss multiplier(entier_de_Gauss x, entier_de_Gauss y){
   entier_de_Gauss z;
   z.reelle = x.reelle*y.reelle - x.imaginaire*y.imaginaire;
   z.imaginaire = x.reelle*y.imaginaire + x.imaginaire*y.reelle;
   return z;
}
int main(int argc,char * argv[]){
   entier_de_Gauss x1, x2, res;
   if (argc < 5){
      usage(argv[0]);
      exit (-1);
   }
   x1.reelle = atoi(argv[1]);
   x1.imaginaire = atoi(argv[2]);
   x2.reelle = atoi(argv[3]);
   x2.imaginaire = atoi(argv[4]);
   res = multiplier(x1,x2);
   printf("(%d+%di) x (%d+%di) = %d+%di\n",x1.reelle, x1.imaginaire, x2.reelle, x2.imaginaire,
           res.reelle,res.imaginaire);
   exit(0);
}
```

Structure de données - Notions de bases

Un autre exemple

```
struct date {
   int an;
   short mois, jour;
} ;
typedef struct date date;
struct personne {
   char nom[20], prenom[20];
   date naissance;
} ;
typedef struct personne etudiant;
```

Il existe une **écriture simplifiée** :

```
typedef struct  {
   int an;
   short mois, jour;
} date ;
typedef struct {
   char nom[20], prenom[20];
   date naissance;
}  etudiant;
```

Pour accéder aux différents champs d'une variable de type étudiant :

```
etudiant etu; /* declaration d'une variable de type etudiant */
etu.nom = "Dupond";
etu.prenom = "Marcel";
etu.naissance.an = 1995;
etu.naissance.mois = 3;
etu.naissance.jour = 21;
```

Manipulation de fichiers

Lecture/Ecriture

Manipulation de fichiers - Lecture/Ecriture

Ce que l'on va voir

- Le type "fichier" : `FILE *`
- Ouverture et fermeture de fichier
- Différentes méthodes de lecture/écriture

Manipulation de fichiers - Lecture/Ecriture

Le type "fichier" : `FILE *`

- **Déclaration :** `FILE * file;`

↪ réserve le pointeur vers une variable de type `FILE`

⇒ Il va falloir réserver la zone mémoire en "ouvrant" le fichier

- Cette structure se trouve dans la bibliothèque `stdio.h`

Rq : `EOF` est le marqueur de fin de fichier *(End Of File)*

Manipulation de fichiers - Lecture/Ecriture

Ouverture/Fermeture d'un fichier

Procédure à suivre

- **Ouverture de fichier** avec `fopen();` (renvoie un pointeur sur le fichier)
- **Vérification** de l'ouverture (est-ce que fichier existe ?)

$\Rightarrow$ Si le pointeur vaut `NULL`, l'ouverture a échoué (afficher un message d'erreur)

$\Rightarrow$ Si le pointeur n'est pas `NULL`, on peut lire et/ou écrire dans le fichier

- Quand on a finit, on **ferme** le fichier avec `fclose();`

Manipulation de fichiers - Lecture/Ecriture

Ouverture/fermeture d'un fichier

```
FILE* fopen(char* nom_fich, char * mode);
```

↪ Renvoie un pointeur vers le fichier en cas de succès, `NULL` sinon

- `nom_fich` est le chemin vers le fichier
- où `mode` est une chaîne de caractère :

mode	lecture	écriture	création	vider	position	description
r	X				début	**lecture seule, le fichier doit exister**
r+	X	X			début	**lecture et écriture**
w		X	X	X	début	**écriture seule**
w+	X	X	X	X	début	**lecture et écriture, avec suppression du contenu**
a		X	X		fin	**ajout à la fin**
a+	X	X	X		fin	**ajout en lecture/écriture à la fin**

```
int fclose(FILE* fichier);
```

↪ renvoie un entier

- 0 : si la fermeture a marché
- `EOF` : si la fermeture a échoué

Manipulation de fichiers - Lecture/Ecriture

Ouverture/Fermeture d'un fichier — Un exemple

```
int main(){
    FILE* fichier = NULL;
    fichier = fopen("test.txt", "r+");
    if (fichier == NULL){ /* si le fichier n'existe pas */
        printf("Vous tentez d'accéder à un fichier inexistant\n");
        exit(-1);
    }
    /* sinon on peut lire et écrire dans le fichier */
    ..
    ..

    fclose(fichier);

    exit(0);
}
```

Rq : Si `test.txt` existe, le pointeur `fichier` devient un pointeur sur `test.txt`

Manipulation de fichiers - Lecture/Ecriture

Écriture dans un fichier

- d'un seul caractère `int fputc(int caractere, FILE* fichier);`

↪ renvoie le caractère écrit si succès, `EOF` sinon

Rq : `caractere` est un entier, mais revient à utiliser `char` (vous pouvez écrire `'A'`)

- d'une chaîne : `int fputs(char* chaine, FILE* fichier);`

↪ renvoie un entier négatif si succès, `EOF` sinon

- `chaine` est la chaîne à écrire

- d'une chaîne formatée :

  `int fprintf(FILE * fichier, char* chaine_formatee{,liste});`

↪ retourne le nombre de caractère écrits

Rq : s'utilise comme `printf`, excepté le 1er paramètre qui est un pointeur de FILE

Manipulation de fichiers - Lecture/Ecriture

Lecture dans un fichier

- d'un seul caractère `int fgetc(FILE* fichier);`

↪ renvoie le caractère lu si succès, `EOF` sinon

- d'une chaîne : `char* fgets(char* chaine, int nb_car, FILE* fichier);`

↪ renvoie le pointeur vers la chaîne lue, `NULL` sinon

- `nb_car` est le nombre de caractères à lire

- d'une chaîne formatée :
  `int fscanf(FILE * fichier, char* format{,liste});`

↪ retourne le nombre d'affectation(s) effectuée(s), `EOF` en cas d'erreur

Rq : s'utilise comme `scanf`, excepté le 1er paramètre qui est un pointeur de FILE

Manipulation de fichiers - Lecture/Ecriture

Lecture dans un fichier — Un exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
   FILE* fichier = NULL;
   char chaine[TAILLE_MAX];
   fichier = fopen("test.txt", "r");
   if(fichier==NULL){
     printf("Vous tentez d'ouvrir un fichier inexistant\n");
     exit(-1);
   }
     /* On lit le fichier tant qu'on ne reçoit pas d'erreur (NULL) */
   while (fgets(chaine, TAILLE_MAX, fichier) != NULL){
     printf("%s",chaine); /* On affiche la chaîne qu'on vient de lire */
   }

   fclose(fichier);
   exit(0);
}
```

Lecture dans un fichier — Un autre exemple

On suppose que le fichier `test.txt` contient 3 nombres séparés par un espace (ex : 15 20 30)

```
#include<stdio.h>
#include<stdlib.h>

int main(){
    FILE* fichier = NULL;
    int score[3];
    fichier = fopen("test.txt", "r");
    if(fichier==NULL){
       printf("Vous tentez d'ouvrir un fichier inexistant\n");
       exit(-1);
    }

    fscanf(fichier, "%d %d %d", &score[0], &score[1], &score[2]);
    printf("Les meilleurs scores sont : %d, %d et %d",
                        score[0], score[1], score[2]);

    fclose(fichier);
    exit(0);
}
```

Manipulation de fichiers - Lecture/Ecriture

Écriture dans un fichier — Un exemple

```
#include<stdio.h>
#include<stdlib.h>

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
       printf("Impossible d'ouvrir le fichier\n");
       exit(-1);
    }

    fputs("Salut !\nComment allez-vous ?", fichier);
    fclose(fichier);
    exit(0);
}
```

Écriture dans un fichier — Un autre exemple

```
#include<stdio.h>
#include<stdlib.h>
#define TAILLE_MAX 1000

int main(){
    FILE* fichier = NULL;
    int age ;
    fichier = fopen("test.txt", "w");
    if(fichier==NULL){
       printf("Impossible d'ouvrir le fichier\n");
       exit(-1);
    }
    /* on demande votre age */
    printf("Quel age avez-vous ?\n");
    scanf("%d", &age);

    /* On l'écrit dans le fichier */
    fprintf(fichier, "Le Monsieur qui utilise le programme, il a %d ans", age);

    fclose(fichier);
    exit(0);
}
```

Manipulation de fichiers - Lecture/Ecriture

Quelques mots sur `printf/fprintf` et `scanf/fscanf`

- `scanf` et `printf` utilisent le flot standard (`stdin`, `stdout`)
- `fscanf` et `fprintf` permettent préciser le flot
  (c'est pourquoi on peut les utiliser sur un `FILE *`)

$\Rightarrow$ `scanf(``%d'',&x);` $\Longleftrightarrow$ `fscanf(stdin,``%d'',&x);`

$\Rightarrow$ `printf(``salut !'');` $\Longleftrightarrow$ `fprintf(stdout,``salut!'');`

- Utile pour différencier les affichages d'erreurs des affichages "normaux"

$\Rightarrow$ On peut afficher nos messages d'erreurs sur `stderr` :
`fprintf(stderr,``ceci est un message d'erreur\n'');`

Programmation modulaire

Programmation modulaire

Retour sur la production de programmes

1. Pré-processeur : `.c` $\rightarrow$ un `.c` pour chaque fichier C
2. Compilateur : `.c` $\rightarrow$ un `.o` (langage machine) pour chaque fichier `.c`
3. Edition de liens : *combinaison des* `.o` $\rightarrow$ executable (langage machine)

Programmation modulaire

Pré-processeur

- `#include` : inclusion textuelle de fichiers
  - Bibliothèque C (ex : `stdio.h`, `math.h`,...)
  - Fichier perso de déclarations de fonctions

- `#define` : remplacement textuel
  - ex : `#define PI 3.1415926`

- Option de `gcc` : `-E` pour visualiser le texte C modifié après passage du pré-processeur

Programmation modulaire

Compilateur

- Traduction en langage machine
- Contrôle des arguments et des types des valeurs de retour pour toutes les fonctions
- Il ne connaît pas le code des fonctions seulement déclarées (option `-c`)

Édition des liens

- Crée l'exécutable à partir des fichiers objets
- "Résolution" des noms de fonctions et de variables indéfinis
- Arrêt dès qu'il y a un défaut

Programmation modulaire

Compilation des modules

- Compilation séparée de chaque module indépendamment

- Permet
  - lisibilité
  - partage du travail
  - maintenance
  - réutilisabilité

Compilation des modules — Un exemple

1. Fichier `pluriel.c`

```
char pluriel_simple (int n) {
   if (n == 1)
      return ' ';
   else
      return 's';
}
```

2. Compilation : `gcc -c pluriel.c` → `pluriel.o`

3. Utilisation dans un programme `prog.c`
   - Déclarer la fonction : `char pluriel_simple(int n);`
   - Lier sa définition : `gcc prog.c pluriel.o -o prog`

Programmation modulaire

Découpage d'un programme en modules

Un module

- Propose un service via une interface de programmation
- Masque l'implantation (la réalisation)
- Un module = 3 fichiers : `.h`, `.c`, `.o`

- Le fichier **interface de programmation** : `.h`

↪ Déclarations des fonctions du module

- Le fichier **réalisation** : `.c`

↪ Définitions des fonctions et fonctions cachées (non présentes dans le `.h`)

- Le fichier **objet** : `.o`

↪ Code machine des fonctions du module

Programmation modulaire

Utiliser un module en C

Soit un programme `prog.c` voulant utiliser un module module :

- Recopier `module.h` et `module.o` dans le même répertoire que `prog.c`
- Dans `prog.c` : `#include "module.h"`
- Compilation : `gcc prog.c module.o -o prog`

Remarque : `module.h` et `module.o` peuvent être dans un autre répertoire, il faut juste penser à bien indiquer les chemins correctement

Créer un module — Un exemple

Pour créer le module `pluriel`

- fichier d'interface : `pluriel.h`

```
char pluriel_simple (int n);
```

- fichier source : `pluriel.c`

```
char pluriel_simple (int n){
   if (n==1)
      return ' ';
   else   return 's';
}
```

- fichier objet : `pluriel.o`, obtenu par `gcc -c pluriel.c`

Programmation modulaire

Automatisation de la compilation : `Makefile`

- Programmes de taille réaliste :
  - plusieurs modules
  - re-compilations partielles
  - détermination une fois pour toute des dépendances
- Utilisation de l'utilitaire `make` via un fichier `Makefile`

Programmation modulaire

Structure d'un fichier `Makefile`

Un `Makefile` se compose différentes sections

- **déclarations de variables** sous la forme : `<nom> = <valeur>`
- **cible** : un nom d'exécutable ou de fichier objet
- **dépendances** : les éléments ou le code source nécessaires pour créer une cible
- **règles** : les commandes nécessaires pour créer la cible

```
<déclarations de variables>

<cible> : <dépendances>
      <TABULATION><regle1>
      ...
      <TABULATION><regle n>
```

Programmation modulaire

Exemple d'un fichier `Makefile`

Soit `prog.c` qui utilise les modules `mod1` et `mod2`

```
CC= gcc
OPTIONS = -Wall

prog : mod1.o mod2.o prog.o
	$(CC) $(OPTIONS) mod1.o mod2.o prog.o -o prog
prog.o : prog.c prog.h
	$(CC) $(OPTIONS) -c prog.c
mod1.o : mod1.c mod1.h
	$(CC) $(OPTIONS) -c mod1.c
mod2.o : mod2.c mod2.h
	$(CC) $(OPTIONS) -c mod2.c
clean :
	rm -f *.o *~ prog
```

Programmation modulaire

Utilitaire `make`

Exploitation du fichier makefile : utilitaire `make`
Syntaxe : `make <option> <cible>`

```
>make
gcc -Wall -c mod1.c
gcc -Wall -c mod2.c
gcc -Wall -c prog.c
gcc -Wall mod1.o mod2.o prog.o -o prog
>make clean
rm -f *.o *~ prog
>make mod1.o
gcc -Wall -c mod1.c
>make
gcc -Wall -c mod2.c /* si seulement mod2 a été changé (date fichier)
gcc -Wall mod1.o mod2.o prog.o -o prog
>make
'prog' is up to date
>
```

Portée des variables et des fonctions

Portée des variables et des fonctions

Portée des déclarations dans une fonction

À l'intérieur d'une fonction, on **"voit" (= a accès à)**

- les **fonctions** déclarées **avant**
- les **types** et les **variables** déclarés
  - **avant** et **dans la fonction**
  - **avant dans le fichier source** et **hors d'un bloc** (ex : hors d'une fonction)

À l'intérieur d'une fonction, on **ne voit pas**

- les **variables masquées** (par une déclaration locale de même nom)
- les **variables** déclarées **dans un bloc** (ex : dans une autre fonction)

Portée des variables et des fonctions

Portée d'une fonction

Une fonction est **utilisable**

- dans **son module** de définition : après la définition ou après la déclaration
- dans un **autre module** : après la déclaration

Rq : Une fonction définie `static` **n'est pas accessible** depuis un **autre module**

```
static int maFonctionLocale(){ ... }
```

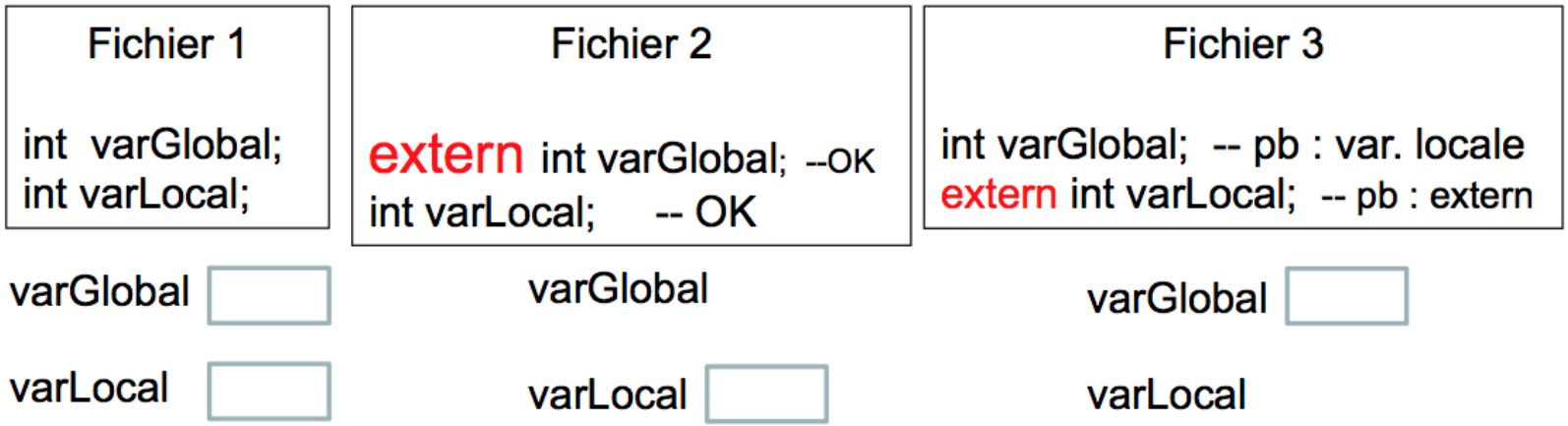
Cette fonction ne sera utilisable QUE dans son module

Portée des variables et des fonctions

Portée des variables

Une variable **définie en dehors d'un bloc (ou fonction)** est **accessible**

- dans **son module** de définition : après la déclaration
- dans un **autre module** : après une déclaration `extern`
  (déclaration sans réserve de zone mémoire)



Portée des variables et des fonctions

Portée des types

Un nouveau type (défini par `typedef`) a une portée **limitée à son module**

Pour pouvoir l'utiliser ailleurs

$\hookrightarrow$ Il faut créer et inclure un fichier de définition de types

Portée des variables et des fonctions

Quelques mots sur `static` et `extern`

Attributs `static` et `extern` dans une programmation par modules

- `static` : pour une variable et une fonction

↪ Implique que la définition reste locale au module

- `extern` : pour une variable

↪ déclaration sans réserve de zone mémoire

⇒ variable dite "globale" qui doit être déclarée sans `static` dans un autre module

- `extern` : pour une fonction (implicite devant la déclaration)