

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 :
Serveur discord : Réactivité/Interactions ++
Mail : `emilie.morvant@univ-st-etienne.fr`

Objectifs du cours

- Découvrir le langage de programmation impérative C
 - ▶ après Python en L1 MISPIC
- Maîtriser 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 - lien sur claroline)

Introduction

Notion d'algorithme

Qu'est ce qu'un algorithme ?

↪ C'est une suite d'actions simples 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 d'un problème qu'on peut rencontrer...

```
Avancer de 55 pas en ligne droite
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 ?!
(par exemple si le mur est à une distance de 10 pas)

⇒ **ERREUR ! On fonce droit dans le mur...**

Solution

IL FAUT POUVOIR TESTER À CHAQUE PAS

Notion d'algorithme

Solution

Utilisation d'une instruction répétitive : boucle "while"

Exemple

```
nombre_de_pas ← 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 ← 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 : boucle “do...while”

Exemple

```
nombre_de_pas ← 0
repéter
  Avancer de 1 pas en ligne droite
  nombre_de_pas ← 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 deux 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

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.

Les composants d'un ordinateur : disque dur, mémoire, processeur, etc.

↪ C'est le **processeur** qui fait les **opérations** (calculs, tests)

Un programme est une 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 → ajoute 1 à une case mémoire désignée par R3

De l'algorithme au programme

L'ordinateur, ou plutôt le processeur, "travaille" en langage machine

Langage machine = succession de chiffres binaire - **BIT**

0 : le courant ne passe pas

1 : le courant passe

- le stockage physique des bits dépend des technologies :
différentiels de tension (condensateurs), moments magnétiques,
cuvettes ou surfaces planes, émission de photons, etc
- Composition de bit : octet (=byte) = 8 bits
- Codage en base 2 :

$$0001 = 0 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 1$$

$$0010 = 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 0 \times 2^0 = 2$$

$$0011 = 0 \times 2^3 + 0 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 3$$

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

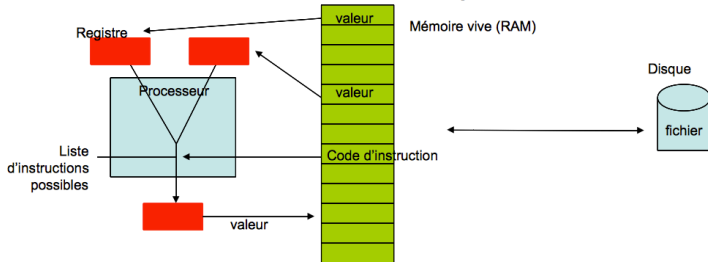
De l'algorithme au programme

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

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

De l'algorithme au programme

- 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

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

- ⇒ Trop élémentaire et donc trop long à programmer pour l'humain
- ⇒ Nécessité de construire un langage de programmation pour avoir
 - des programmes à un niveau d'abstraction plus élevé
 - une meilleure expressivité et généralité
 - moins de risques d'erreurs
- ⇒ Besoin de passer du langage de programmation à une exécution
- ⇒ Un programme doit être successivement
 - saisi et enregistré (édition) → fichier texte
 - traduit (compilation / interprétation) → code binaire
 - "relié" (édition de liens) → fichier exécutable
 - exécuté, testé, mis en service (débugage)

De l'algorithme au programme

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

Compiler un programme

Un programme dans un langage **est traduit** en langage machine

- Écrire un programme dans un ou des fichiers : **fichier(s) source(s)**
- Soumettre au compilateur les fichiers
- 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 émet des “warning” lorsqu’il détecte des incohérences dans le code

Un programme correct doit compiler sans avertissement

(nécessaire, mais pas suffisant...)

De l'algorithme au programme

Quelques remarques importantes

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

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 des instructions l'une après l'autre
c'est-à-dire traduction et exécution des instructions
- Plus lent que la compilation
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

Exemples de langages de programmation

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

Machines virtuelles (VM)

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

De l'algorithme au programme

Lors de la compilation

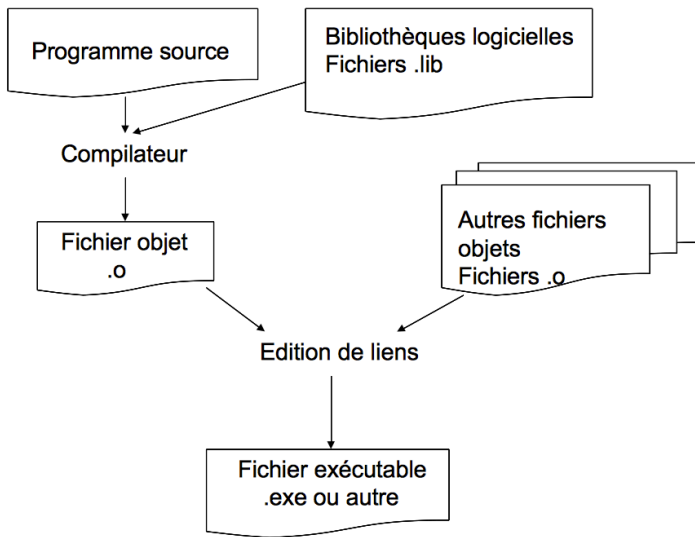
Chaque fichier source est compilé
et produit un fichier "objet"

Edition de liens

Fusion des fichiers objets pour créer le programme
binaire exécutable

- **Concerne les langages compilés** (*comme le C*)
- Permet d'écrire de gros programmes (dans plusieurs fichiers)
- Permet la réutilisation de modules

De l'algorithme au programme



De l'algorithme au programme

Paradigme de programmation - Définition Wikipédia

Un paradigme de programmation est une façon d'approcher la programmation informatique et de traiter les solutions aux problèmes et leur formulation dans un langage de programmation approprié. Il s'oppose à la méthodologie, qui est une manière d'organiser la solution des problèmes spécifiques du génie logiciel.

Exemples de paradigmes

- **Programmation impérative**

Un programme est une suite explicite d'instructions à exécuter
ex. C (L2), Java (RDV en L3), C++, Ada, etc.

- **Programmation fonctionnelle**

Un programme est un ensemble de fonctions
L'exécution est une évaluation de fonction—ex. CAML (L1), LIPS

- **Programmation logique**

Un programme est un ensemble de théorèmes
L'exécution en est une preuve—ex. Prolog (RDV en L3)

De l'algorithme au programme

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 algos 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
gcc est le compilateur du projet GNU
- 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 : 1972
- Auteurs : Kernighan et Ritchie, Bell Labs
- But : proposer un langage impératif compilé, à la fois de haut niveau et “proche de la machine” (rapidité d’exécution)
- Conçu pour être le langage de programmation d’Unix, le 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 presque 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
- (-) (ds 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 "nouveauautés" (exception, ramasse-miettes...)

Historique du langage C et caractéristiques

Langage impératif et de haut niveau

- Programmation structurée
 - pas d'instruction de type "GO TO", 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

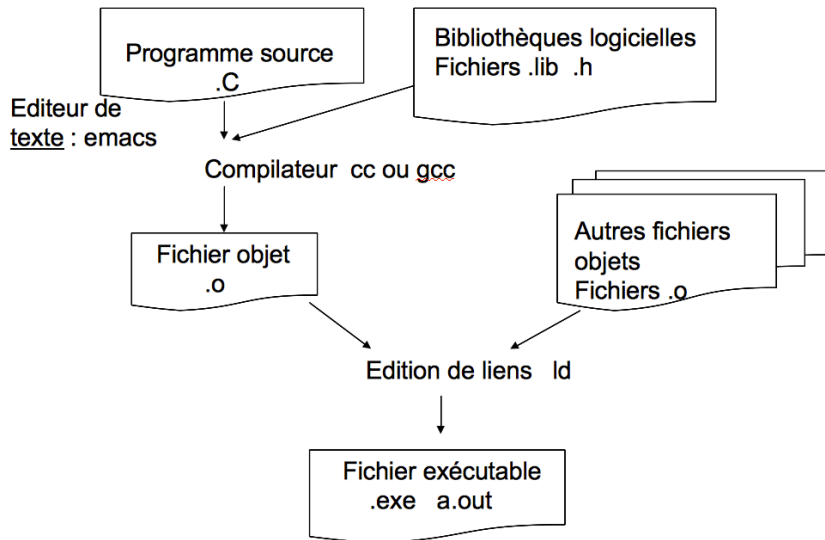
- Conçu pour être facilement traduit en langage machine
- Gestion de la mémoire "à la main"
- Attention : pas de gestion des exceptions

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 **cc** ou **gcc**) vérifie la syntaxe du code source et le traduit en code **objet** qui sera compris par le processeur
- Le programme obtenu en code objet peut 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 pour compiler et exécuter

```
> 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
>
```

NB : si besoin, vérifier que l'utilisateur a bien les droits en exécution sur le fichier exécutable

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”

Exemple de compilation de “diviseur.c”

```
% gcc -Wall -ansi diviseur.c -o diviseur  
diviseur.c (+stdio.h + stdlib.h) → diviseur
```

Les options de compilation

- `-o nomfichier` : nom du fichier de sortie/exécutible
(par défaut : “a.out”)
- `-Wall` : pour avoir tous les avertissements
- `-ansi` : pour compiler du C standard (norme ANSI C)
- 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) → 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

- Premières lignes de code
- En début de ligne : `#include`

- Bibliothèques “système” :

```
#include <nomBiblio>
```

utilise un dossier connu du compilateur, en général : `/usr/include`

Exemples de biblios : `stdio`, `stdlib`, `ctype`, `string`, `time`, `math`, 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 d'une fonction :

```
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 (c-à-d “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 divis
#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` : nb d'arguments sur la ligne de commande (**c**=count)
NB : Le nom de l'exécutable compte pour 1
`argv` : tableau 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> : chaîne de caractères, contenant 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");  
printf("Pour afficher %, il faut écrire %%%.\n");
```

Affiche :

Le carré de x est 25.

Merci.

Pour afficher \\, il faut écrire \\\

Pour afficher %, il faut écrire %%

IMPORTANT : `\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/fr/courses/](https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16993-la-saisie-de-texte-securisee)

[19980-apprenez-a-programmer-en-c/16993-la-saisie-de-texte-securisee](https://openclassrooms.com/fr/courses/19980-apprenez-a-programmer-en-c/16993-la-saisie-de-texte-securisee)

Comprendre un programme

Exemples de fonctions pré-définies

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

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

Lit 3 entiers relatifs, séparés par des espaces, tabulations ou interlignes

Les valeurs sont affectées respectivement aux variables `jour`, `mois`, `an`

Si on entre : 06 10 2014

alors : `jour=6`, `mois=10` et `an=2014`

NB : Une suite de signes d'espacement est évaluée comme un 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 (évite 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 (mais pas trop long)
 - ▶ Choisir un nom significatif !!!! (relecture extrêmement 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[]` : 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

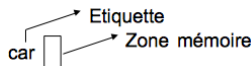
Variables et types

Les types simples

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

- Déclarations :

`char car;`



`int i;`

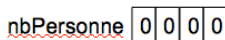


`double resultat;`



- Déclarations avec initialisation :

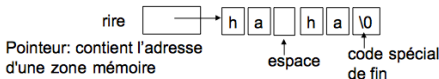
`int nbPersonne = 0 ;`



`char c = 'b';`



`char rire[] = 'ha ha';`



Variables et types

Des variables constantes !

Mot clé : `const`

S'utilise lors d'une déclaration de variable à laquelle on affecte une valeur pour indiquer au compilateur d'interdire tout changement de valeur de cette variable

Exemple : `const int joursParSemaine = 7;`

Variables et types

Les macros (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 qui 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] <opérateur> [expression]`
- `[variable]` représente une variable existante
- L'évaluation de `[expression]` donne un résultat qui devient la nouvelle valeur de `[variable]`
- `<opérateur>`
 - ▶ `=` 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 */
```

- Rq : Peut être utilisé pour d'autres variables (pour programmeur confirmé)

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
- *IMPORTANT : Contrairement à python, l'indentation n'est pas obligatoire, mais elle est nécessaire pour faciliter l'organisation, la lecture du code et le débogage*

Exemple de blocs

```
{           /* 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> /* exécuté 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
  else
    l2
```

l2 est exécuté si C1 est vrai et C2 faux

Un autre exemple : `if (C1) {if (C2) l1} else l2` se "lit" :

```
if (C1){
  if (C2)
    l1
}else
  l2
```

l2 est exécuté si C1 est faux

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;
            printf("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 “équivalentes” : 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 “équivalentes” : 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 instruction répétitive **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

Une parenthèse importante : l'aléatoire en C

IMPORTANT

L'ordinateur ne sait pas générer l'aléatoire, il ne fait que des calculs...

Syntaxe pour générer un entier dans $[a, b]$ en faisant donc des calculs

```
int alea;  
srand(time(NULL))           /* il faut inclure la biblio time.h */  
alea = (rand() % (b - a + 1)) + a;
```

- `rand()` envoie un entier pseudo-aléatoire dans $[0, \text{RAND_MAX}]$
 - renvoie **toujours** la même suite d'entiers
 - a donc besoin d'être initialisée par une "graine" (seed) pour ne pas commencer à chaque exécution du programme par le même élément
- `srand()` initialise la "graine"
 - pour ne pas avoir tout le temps la même initialisation et donc la même suite, on initialise à l'heure de la machine en secondes : `time(NULL)`
 - doit être exécutée qu'**UNE SEULE FOIS** au début du programme

Un petit bilan

- Un programme = suite d'instructions
- Fichier source → compilation → fichier objet → exécutable
- 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
 - ▶ Conditionnelle : si <condition> alors ... sinon... (if)
 - ▶ Répétitives :
 - tant que <condition> faire ... (while, for)
 - Faire ... jusqu'à (do ... while)
- Le (pseudo-)aléatoire en C : `srand()`, `rand()`, `time(NULL)`

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
- `<nom>` : nom (explicite) de la fonction (commence 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 (`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 : `var = nomFonction(param1, param2, etc);`

- `var` : facultatif (le retour n'est pas utilisé ou fonction de type `void`)
- `var` 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

NB : 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

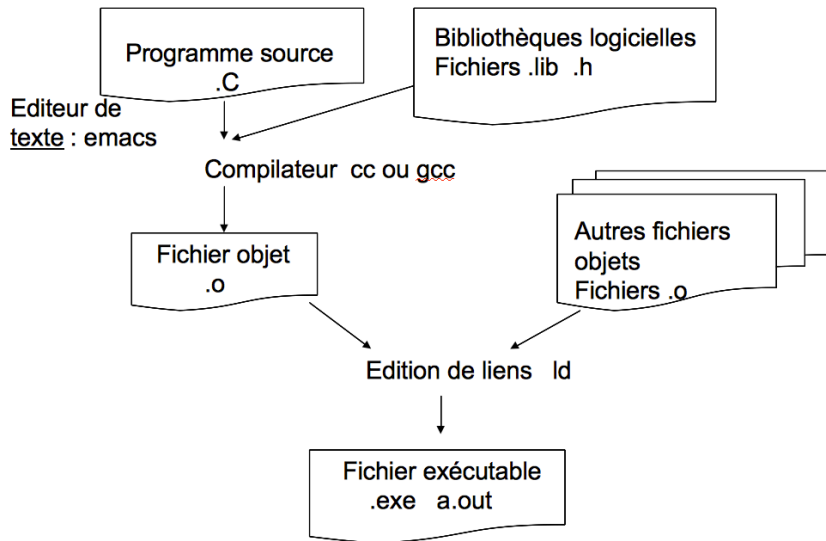
- La déclaration n'est pas obligatoire

MAIS parfois utile et parfois nécessaire

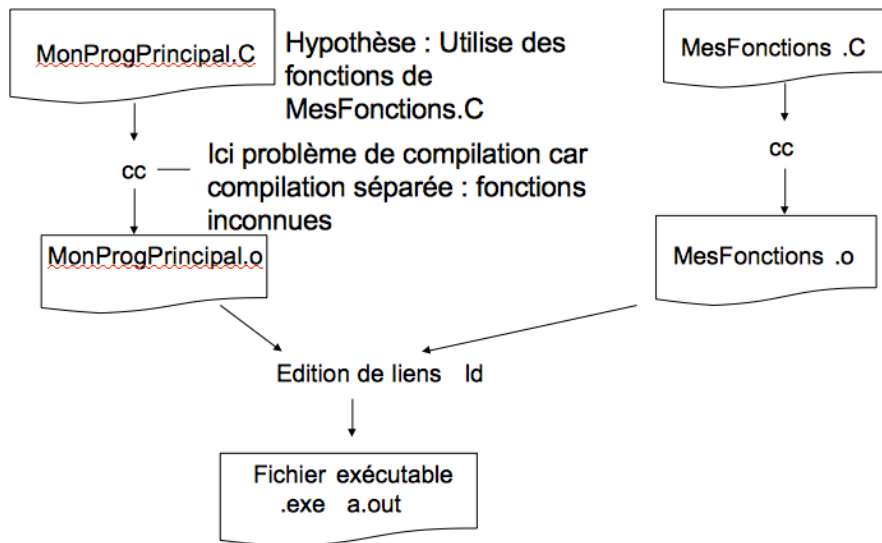
- **Utile** : Le programme appelant (par exemple le main) est placé avant l'écriture de la fonction
- **Nécessaire** : Le programme appelant utilise une fonction d'un autre module (un autre .c, .o)

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

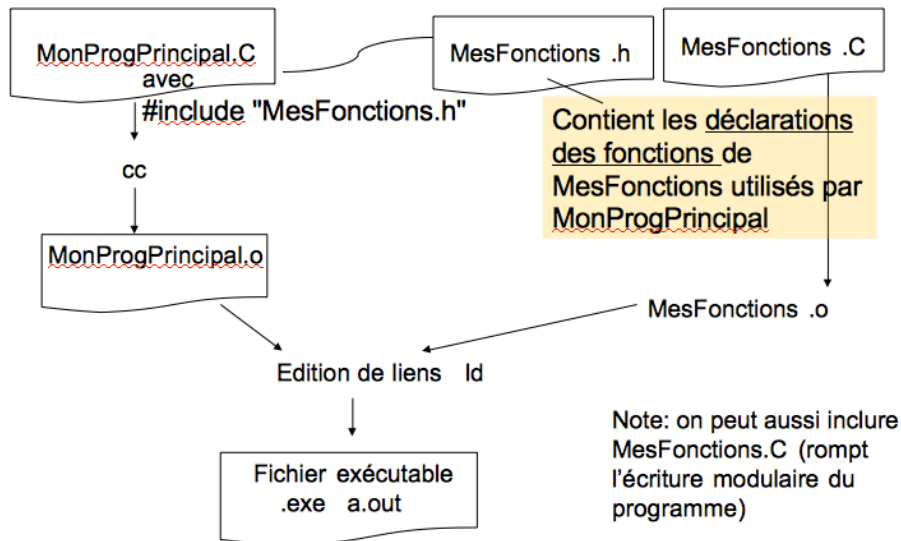
Schéma de production de logiciel C



Programme principal et module(s)



Programme principal et module(s)

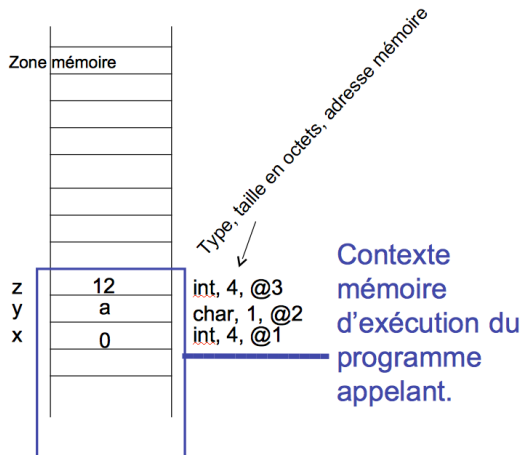


Paramètres de fonction et zone mémoire

Avant l'appel de `maFonction ()`

```
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);
}
```



Paramètres de fonction et zone mémoire

Exécution de `x = maFonction (y, z);`

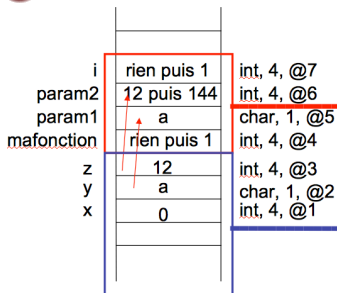
L'appel de la fonction \Rightarrow changement de contexte d'exécution

```
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);
}
```

1 Copie des valeurs données aux paramètres $\text{param1} \leftarrow y$ et $\text{param1} \leftarrow z$

2 Exécution des instructions



Contexte mémoire d'exécution de la fonction.

Contexte mémoire d'exécution du programme appelant.

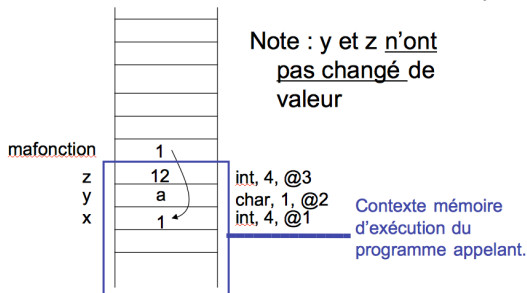
Paramètres de fonction et zone mémoire

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

```
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);
}
```

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

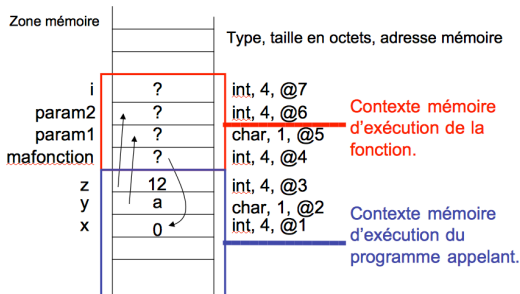


Paramètres de fonction et zone mémoire

Synthèse

```
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);
}
```

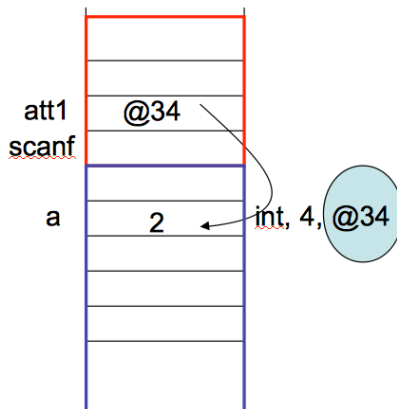


Paramètres de fonction et zone mémoire

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 !

⇒ 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);
}
```

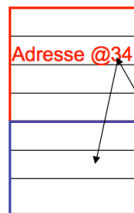
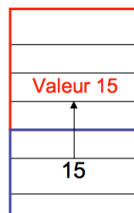
N.B. : Les tableaux sont des pointeurs !!!

Paramètres de fonction et zone mémoire

Un petit bilan IMPORTANT

Passage de paramètres

- soit par valeur
 - ▶ Copie de la valeur
 - ▶ Pas d'accès à la variable initiale 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



copie,
traitement,
fin fonction

Paramètres de fonction et zone mémoire

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

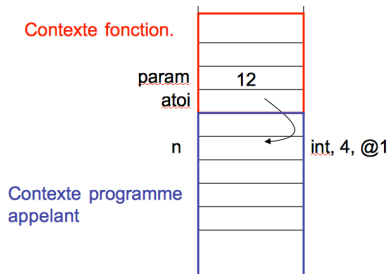
Rappel

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

fonction prédéfinie dans <stdio.h>
Conversion de la chaîne de caractères param en entier

Exemple :

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



Pour les réels

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

Fonction prédéfinie dans <stdio.h>
Conversion de la chaîne de caractères param en réel

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(s) et les expressions calculées

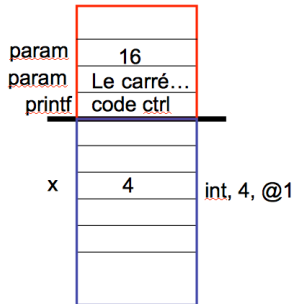
Remarque :

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();
```

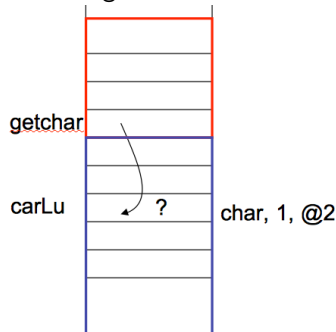
fonction prédéfinie dans `<stdio.h>`

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 entrées au clavier par l'utilisateur

Remarque : scanf() retourne un entier qui 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

Un tableau en C c'est

une représentation tabulaire de données de **même type** (liste, matrice)

de dimension au plus 2

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

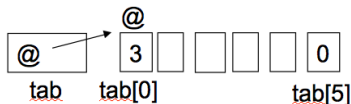
Les tableaux - notions de bases

Déclaration

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

Exemple

<code>int tab[6];</code>	déclare la variable <code>tab</code> comme un tableau de 6 int
<code>tab</code>	adresse de début du tableau
<code>tab[0]</code>	valeur du premier entier <code>tab[0]=3;</code>
<code>tab[5]</code>	valeur du dernier entier <code>tab[5]=0;</code>



Les tableaux - notions de bases

Exemples basiques d'utilisations des tableaux

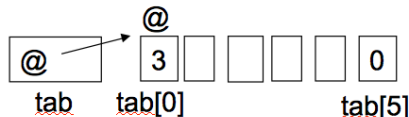
```
int tab[6];      /* déclaration d'un tableau  
                  d'entiers de taille 6    */
```

```
int i = 6, j ;
```

```
tab[0] = 3;
```

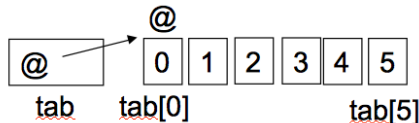
```
tab[5] = 0;
```

```
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 `tab1 = tab2`)

\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

Lors de la déclaration d'un tableau, il est "vide", c-à-d qu'aucune valeur n'a été affectée aux cases \Rightarrow Il faut donc le remplir !

Exemple de remplissage d'un tableau

- 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, 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 stocker des valeurs

Les tableaux - notions de bases

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 définir un nouveau type

Exemple :

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

Les tableaux - notions de bases

Exemples d'algorithmes simples : 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 : `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'`
- ⇒ 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
- ⇒ 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

'j'	'e'	'a'	'n'	'\0'
-----	-----	-----	-----	------

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

Les tableaux - chaînes de caractères

Déclaration

- `char prenom[10];`

↪ réserve 10 caractères : 9 utilisés +1 pr 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 @→

'Z'	'O'	'L'	'A'	'\0'
-----	-----	-----	-----	------

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
ATTENTION : `char maChaine[]`; OU `char * maChaine`;
ne réserve que le "pointeur vers", MAIS pas la zone pour les caractères
↪ Le nombre d'éléments est ici inconnu!
- Insérer le `'\0'` en fin de chaîne si besoin
- **Pas d'affectation "globale" du type** ~~`maChaine1 = maChaine 2;`~~

Les tableaux - chaînes de caractères

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 chaînes
#include <string.h> (Rappel : chemin par défaut /usr/include)

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

str : string cmp : compare n : n premiers caractères len : length
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);
```

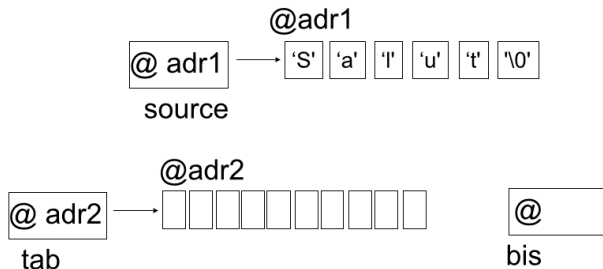


- Recopie de la chaîne de caractère qui commence à l'adresse de **source** dans la chaîne qui commence à l'adresse de **cible**
- **ATTENTION** : il faut avoir réservé la place mémoire (i.e., il faut correctement déclarer les chaînes)
- Le retour de la fonction "pointe" (est l'adresse) de la chaîne **cible**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



Les tableaux - chaînes de caractères

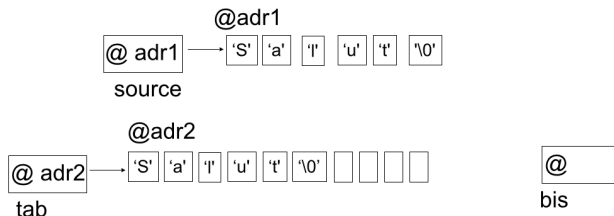
Recopie de chaîne - Un exemple (2/3)

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

Remarques

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()`



Les tableaux - chaînes de caractères

Recopie de chaîne - Un exemple (3/3)

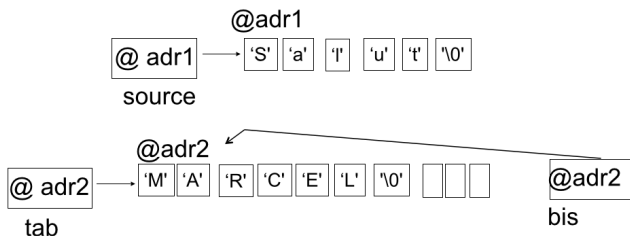
```
char *source = "Salut", tab[10], bis[] ;  
bis = strcpy(tab,"MARCEL");
```

Remarques

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 :

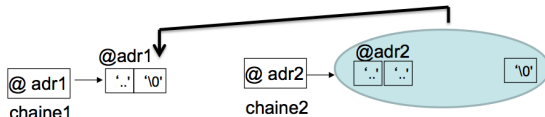
La taille de la chaîne ``Salut'' est : 5

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)

Remarques

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

Les tableaux - chaînes de caractères

Concatenation de chaînes - Deux exemples

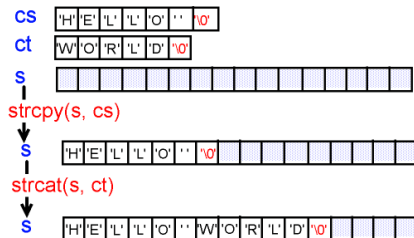
Exemple 1

```
char r[20], s1[20], s2[] = "nation";  
strcpy(s1,"concaté");  
  
r=strcat(s1, s2);  
printf("%s", r);  
printf("%s", s1);
```

/* copie de la chaîne concaté
dans S1 de taille > à 7 ! */
/* concaténation de S1 et S2 */
/* Affiche "concaténation" */
/* 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 (NULL en cas d'erreur)
- `'\n'` n'est pas recopié, un `'\0'` est placé à la fin de la chaîne

NB : `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

Un petit bilan

- Tableau : regroupement de données de même type
- `<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 : il n'y a 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'`
- `string.h` : bibliothèque contenant de nombreuses fonctions de manipulation de chaînes

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 connu par :

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

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`
⇒ **CONSEIL : 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>;`

Structure de données - Notions de bases

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> \n", 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;
```

Écriture simplifiée :

```
typedef struct {
    int an;
    short mois, jour;
} date ;

typedef struct {
    char nom[20], prenom[20];
    date naissance;
} etudiant;
```

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

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

Manipulation de fichiers

Lecture/Écriture

Ce que l'on va voir

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

Manipulation de fichiers - Lecture/Écriture

Le type “fichier” : `FILE *`

- **Déclaration** : `FILE * file;`
 - réserve le pointeur vers une variable de type `FILE`
 - Il faut réserver la zone mémoire en “ouvrant” le fichier
 - Cette structure se trouve dans la bibliothèque `stdio.h`
- NB** : `EOF` est le marqueur de fin de fichier (*End Of File*)

Ouverture/Fermeture d'un fichier

Procédure à suivre

- 1. Ouverture de fichier** avec `fopen()`; (renvoie un pointeur sur le fichier)
- 2. Vérification de l'ouverture** (est-ce que fichier existe?)
 - ⇒ Si le pointeur est `NULL`, l'ouverture a échoué (afficher un message)
 - ⇒ Si le pointeur n'est pas `NULL`, on peut lire et/ou écrire dans le fichier
- 3. Quand on a finit, on ferme le fichier** avec `fclose()`;

Manipulation de fichiers - Lecture/Écriture

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
<code>r</code>	X				début	lecture seule, le fichier doit exister
<code>r+</code>	X	X			début	lecture/écriture
<code>w</code>		X	X	X	début	écriture seule
<code>w+</code>	X	X	X	X	début	lecture/écriture, suppression du contenu
<code>a</code>		X	X		fin	ajout à la fin
<code>a+</code>	X	X	X		fin	ajout en lecture/écriture à la fin

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

↪ renvoie un entier

- ▶ 0 : si la fermeture a fonctionné
- ▶ EOF : si la fermeture a échoué

Manipulation de fichiers - Lecture/Écriture

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/Écriture

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 int, 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_formatée{,liste});
```

↪ retourne le nombre de caractère écrits

Rq : comme printf, excepté le 1^{er} paramètre qui est un pointeur de FILE

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 : comme `scanf`, excepté le 1^{er} paramètre qui est un pointeur de FILE

Manipulation de fichiers - Lecture/Écriture

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);
}
```


Manipulation de fichiers - Lecture/Écriture

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);
}
```

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);
}
```

Manipulation de fichiers - Lecture/Écriture

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/Écriture

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 *)
- ⇒ `scanf(``%d'', &x);` ⇔ `fscanf(stdin, ``%d'', &x);`
- ⇒ `printf(``salut !'');` ⇔ `fprintf(stdout, ``salut!'');`
- Utile pour différencier les affichages d'erreurs des affichages "normaux"
- ⇒ On peut afficher nos messages d'erreurs sur stderr :
- `fprintf(stderr, ``ceci est un message d'erreur\n'');`

Programmation modulaire

Retour sur la production de programmes

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

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 par le pré-processeur

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, de variables indéfinis
- Arrêt dès qu'il y a un défaut

Compilation des modules

- Compilation indépendante de chaque module
- 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 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 dossier 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 dossier, il faut juste penser à bien indiquer les chemins correctement

Programmation modulaire

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`

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 -rf *.o *~ prog
```

Programmation modulaire

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^I/* 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(){ ... }
```

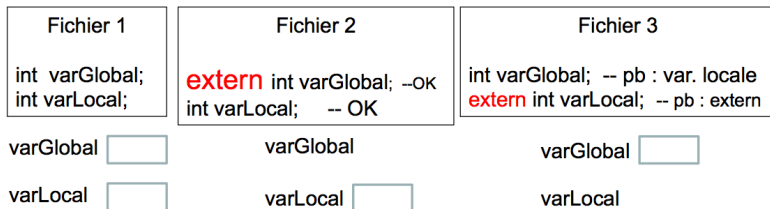
Cette fonction ne sera utilisable **QUE** dans son module

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

↪ 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 globale qui doit être déclarée sans `static` dans un autre module

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