

Programmation en langage C

PG109 | PROG1 – Programmation Impérative

Philippe SWARTVAGHER

ph-sw.fr



À propos de ce cours

Organisation

- ▶ 9 séances d'EI ($9 \times 2 \times 1\text{h}20$)
- ▶ 6 séances de TP ($6 \times 2 \times 1\text{h}20$)

Évaluation

- ▶ 2 derniers TPs notés
- ▶ Partiel en décembre

Attention aux modalités d'évaluation !

- ▶ Les TPs ne sont pas rattrapables !

Supports

- ▶ Cette présentation
- ▶ Vos notes personnelles

Matériel requis

- ▶ Un ordinateur, un terminal, un éditeur de texte, un compilateur
- ▶ Du papier et un crayon

Objectifs

- ▶ Savoir écrire un programme en langage C
- ▶ Comprendre les principes généraux de la gestion de la mémoire d'un programme écrit en C
- ▶ Comprendre les principales étapes de la compilation d'un programme écrit en C

Quelques ressources

Ce cours est basé, entre autres, sur :

- ▶ Claude Delannoy, *Programmer en langage C*, Éditions Eyrolles, 2009
- ▶ Le cours de G. Mercier
(/net/npers/mercier/Cours/PG109/Cours_PG109_2025_2026_Eleves.pdf)
- ▶ Le cours de F. Morandat (**avec des exercices**) :
<https://www.labri.fr/perso/fmoranda/pg101/>
- ▶ Les cours de G. Mercier, F. Pellegrini et R. Giraud
- ▶ Quelques morceaux de *CS 61 : Systems Programming and Machine Organization*, Eddie Kohler,
<https://cs61.seas.harvard.edu/site/2018/>

Pour installer les outils nécessaires pour faire du C sur sa machine :

- ▶ <https://www.labri.fr/perso/fmoranda/cathome/>
- ▶ <https://thor.enseirb-matmeca.fr/ruby/docs/teaching/vmlinux>

Autres ressources :

- ▶ Dennis Ritchie et Brian Kernighan, *The C Programming Language*
- ▶ Beej's Guide to C Programming, <https://beej.us/guide/bgc/>
(jusqu'au chapitre 24)

Un petit sondage

- ▶ Qui a déjà programmé ?

Un petit sondage

- ▶ Qui a déjà programmé ?
- ▶ Avec quels langages ?

Un petit sondage

- ▶ Qui a déjà programmé ?
- ▶ Avec quels langages ?
- ▶ Qui a déjà programmé en C ?

Un petit sondage

- ▶ Qui a déjà programmé ?
- ▶ Avec quels langages ?
- ▶ Qui a déjà programmé en C ?
- ▶ Qui estime maîtriser le C ?

Le langage C

Un langage de programmation...

- ▶ bas-niveau
- ▶ impératif
- ▶ compilé
- ▶ Première version en 1972
- ▶ Toujours très populaire
- ▶ Utilisé dans de nombreux domaines : systèmes d'exploitation, simulation numérique, embarqué, ...

Un langage bas-niveau

Langage bas-niveau

- ▶ Offre peu d'abstractions du matériel (jeu d'instructions du processeur, gestion de la mémoire, ...)
- ▶ « Plus proche de la machine »
- ▶ Apprentissage plus difficile
- ▶ Plus rapide
- ▶ Permet de vraiment maîtriser ce que va faire le processeur et comment il fonctionne
- ▶ Exemples : assembleur, C, C++

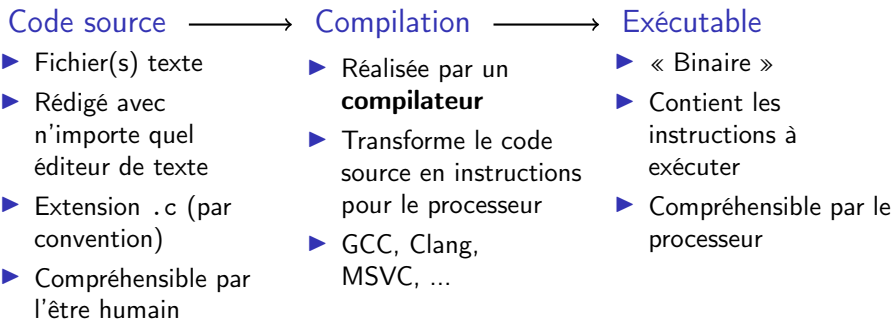
Langage haut-niveau

- ▶ Abstraction importante du matériel (ex. : gestion cachée de la mémoire)
- ▶ Écriture plus rapide de programmes
- ▶ Apprentissage plus facile
- ▶ Pénalité en terme de performances
- ▶ Exemples : Java, Python, C#, ...

Un langage impératif

- ▶ **Paradigme** de programmation
 - ▶ Façon de programmer
- ▶ Expression de *comment* résoudre le problème
- ▶ À l'aide d'une **séquence d'instructions** :
 - ▶ Affectations
 - ▶ Conditions
 - ▶ Boucles
 - ▶ ...
- ▶ Instructions exécutées par le processeur

Un langage compilé



Votre premier programme

Dans le fichier `first_code.c` :

```
#include <stdio.h>    Permet d'utiliser printf

int main()           Définit la fonction main
{
    printf("Hello world!\n");    Appelle la fonction printf

    return 0;        La fonction main renvoie la valeur 0
}
```

- ▶ Fonction `main` : point d'entrée d'un programme C
- ▶ Fonction `printf` : permet d'afficher du texte à l'écran
- ▶ Chaînes de caractères entre guillemets : `"Hello world!\n"`
- ▶ `\n` dans une chaîne de caractère produit un retour à la ligne

Ne copiez/collez pas depuis ce PDF !

- ▶ Toute l'indentation va être cassée
- ▶ Certains caractères bizarres que le compilateur ne comprendra pas vont apparaître
- ▶ Vous allez passer plus de temps à comprendre d'où vient le problème que si vous aviez recopié à la main
- ▶ Recopier à la main va vous aider à retenir

Votre premier programme

Dans le fichier `first_code.c` :

```
#include <stdio.h>    Permet d'utiliser printf

int main()           Définit la fonction main
{
    printf("Hello world!\n");    Appelle la fonction printf

    return 0;        La fonction main renvoie la valeur 0
}
```

Compilation (depuis le terminal) :

```
cc -std=c99 -Wall -Werror first_code.c -o first_code
```

Exécution du programme obtenu (depuis le terminal) :

```
./first_code
```

À vous de jouer !

Options du compilateur

Option	Signification
-std=c99	Utilise le standard C99
-Wall	Affiche tous les avertissements
-Werror	Considère tous les avertissements comme des erreurs (stoppe la compilation)
-o nom_executable	Précise le nom de l'exécutable produit (par défaut : a.out)
Et beaucoup d'autres...	

Toujours utiliser -Wall -Werror en TP !

Instruction

Tout code C décrit une séquence d'instructions :

- ▶ Instruction simple : termine par ;
 - ▶ Exemple : déclaration / affectation de variable
- ▶ Bloc d'instructions : délimitées par { }
- ▶ Instruction de branchement
 - ▶ Exemple : conditions
- ▶ Instruction d'itération
 - ▶ Exemple : boucles
- ▶ ...

Expression

Une expression s'évalue en une **valeur** qui possède un **type** :

- ▶ une constante : 42 (de type entier)
- ▶ un calcul arithmétique : $3.5 + 2.8$ (de type flottant)
- ▶ la valeur retournée par une fonction : `foo(3, 4)`
- ▶ une variable : `nombre_billes` (de type entier)
- ▶ ...

Valeur qui peut (par exemple) être affectée à une variable à l'aide d'une instruction

Quelques remarques sur le code

Écriture de code C

Le C est un langage *sensible à la casse* (fait la différence entre les majuscules et les minuscules) :

- ▶ `printf` \neq `PRINTF` \neq `Printf`
- ▶ `nombre_billes` \neq `Nombre_Billes`

Commentaires

```
/* Ceci est un commentaire */

/* Les commentaires commençant par '/*'
   peuvent être sur plusieurs lignes,
   mais doivent toujours finir par */

// Ceci est un commentaire sur une unique ligne

printf("Bonjour "); // affiche 'Bonjour '

printf(/* texte : */ "le monde !\n");
```

Le bon et le mauvais commentaire

- ▶ Un commentaire facilite la lecture et la compréhension du code
- ▶ Il ne décrit pas le code
- ▶ À l'extrême : un code bien écrit se passe de commentaires¹

```
int i = 0; // compteur

for (i = 1; i < nb_billes-2; i++)
{
    /* Selon nos règles, les premières et
       dernières billes ne comptent pas */

    // Affichage du score :
    printf(...);
}
```

Quels sont les commentaires superflus ?

1. Robert C. Martin, *Clean Code : A Handbook of Agile Software Craftsmanship*
(à lire avec un esprit critique)

Espaces dans le code

Le C n'est pas sensible aux espaces (alignement, tabulations, retours à la ligne, ...), **mais ça améliore la lisibilité du code.**

Ces trois codes compilent et seront compilés de la même manière :

```
int main()
{
    printf("Hello\n");

    return 0;
}
```

```
int main() {
    printf("Hello\n");
    return 0; }
```

```
int main() { printf("Hello\n"); return 0; }
```

Style de code

- ▶ Quand faire des retours à la ligne, comment aligner le code, règles de nommage, longueur de lignes, anglais ou autre, ...
- ▶ Le C n'impose rien, donc à vous de choisir
- ▶ Privilégiez la lisibilité (quite à en écrire un peu plus)
- ▶ **Soyez constants et cohérents :**
 - ▶ Choisissez un style et appliquez-le à l'ensemble du projet
 - ▶ Mettez-vous d'accord au début de chaque projet en groupes
 - ▶ Suivez le style des projets existants

Vous vous remercieriez dans 6 mois !

Exemples de styles de codes

(liste non exhaustive)

```
int uneVar = 0;
if(uneVar >= 3) {
    printf("Oui\n");
}
else{ printf("Non\n");
}
```

```
int uneVar = 0;

if(uneVar >= 3) {
    printf("Oui\n");
}
else {
    printf("Non\n");
}
```

```
int une_var = 0;

if (une_var>=3)
{
    printf("Oui\n");
}
else
{
    printf("Non\n");
}
```

```
int une_variable = 0;

if (une_variable >= 3)
{
    printf("Oui\n");
}
else
{
    printf("Non\n");
}
```

Quelles différences ? Lequel est le plus lisible ?

Utilisation de LLM

- ▶ Écriture de code... Pourquoi ne pas demander à un LLM ?
 - ▶ *Large Language Model*
 - ▶ ChatGPT, GitHub Copilot, ...

Utilisation interdite !

- ▶ L'objectif est que vous compreniez le code que vous produisez, pas que vous recopiez sans comprendre les réponses de ChatGPT
- ▶ Pour bien comprendre ce que propose ChatGPT, il faut déjà maîtriser soi-même le C
- ▶ À votre niveau d'apprentissage du C, utiliser un LLM est comme demander à votre voisin de faire vos devoirs à votre place

Variables

Variable

- ▶ Identifie une zone mémoire
- ▶ Possède :
 - ▶ un type : nombre entier, nombre flottant, adresse mémoire, ...
 - ▶ un nom : [a-zA-Z_] [a-zA-Z0-9_]* (pas de caractères spéciaux, accentués, ...)
- ▶ Portée :
 - ▶ où la variable est accessible dans le code source
 - ▶ au sein du bloc d'instructions (jusqu'au prochain `}`)

Déclaration :

```
type identifiant;
```

Affectation :

```
identifiant = expression;
```

Le `=` n'est pas l'égalité mathématique !
En C, il peut se lire « reçoit ».

Déclaration et affectation simultanées :

```
type identifiant = expression;
```

Affichage de variables

```
#include <stdio.h>

int main()
{
    int longueur = 3;
    int largeur = 5;      int : type pour les entiers
    int aire_rectangle = longueur * largeur;

    printf("%d cm x  %d cm = %d cm^2\n",
           longueur, largeur, aire_rectangle);

    return 0;            %d sera remplacé par le contenu de
                           la variable correspondante
}
```

À vous de jouer !

Types primitifs de variables

Types entiers

- ▶ Pour stocker les nombres entiers
- ▶ `char`, `short`, `int`, `long`, et d'autres
- ▶ Taille de ces types différente, différents intervalles représentables
- ▶ Version `unsigned` pour des entiers non-signés (positifs)
- ▶ `char` aussi utilisé pour stocker les caractères
(cf partie sur les chaînes de caractères)

Types flottants

- ▶ Pour stocker les nombres décimaux (« à virgule flottante »)
- ▶ `float`, `double`
- ▶ Taille de ces types différente, différents intervalles représentables, précisions différentes

Représentation des entiers en mémoire

Entiers non signés

Décomposition en base binaire : $13 = 1 + 4 + 8 = 2^0 + 2^2 + 2^3$

1 octet = 8 bits =

0	0	0	0	1	1	0	1
b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0

Avec n bits, représentation des entiers entre $[0, 2^n - 1]$

Entiers signés

Complément à 2 : miroir sur tous les bits et + 1 ($X + (-X) = 0$)

Bit de signe →	Représentation binaire							Non signé	Signé
0	0	0	0	1	1	0	1	13	13
1	1	1	1	0	0	1	0	242	-14
1	1	1	1	0	0	1	1	243	-13

Avec n bits, représentation des entiers signés entre $[-2^{n-1}, 2^{n-1} - 1]$

Représentation des entiers dans le code

Base décimale

- ▶ Notation en base 10
- ▶ Suite de chiffres décimaux : [0-9]
- ▶ Exemple : 42

Base octale

- ▶ Notation en base 8
- ▶ Suite de chiffres octals : [0-7]
- ▶ Utilisation du préfixe 0 (zéro)
- ▶ Exemple : 042 ($= 4 \times 8^1 + 2 \times 8^0 = 34$)

Base hexadécimale

- ▶ Notation en base 16
- ▶ Suite de chiffres hexadécimaux : [0-9A-F]
- ▶ Utilisation du préfixe 0x ou 0X (zéro x)
- ▶ Exemple : 0x42 ($= 4 \times 16^1 + 2 \times 16^0 = 66$)

Types primitifs de variables

Type	Bits	Intervalle	Code printf
char	8	$[-128, 127]$	%hhd
unsigned char		$[0, 255]$	%hhu
short (int)	16	$[-32768, 32767]$	%hd
unsigned short (int)		$[0, 65535]$	%hu
int	32	$[-2147483648, 2147483647]$	%d
unsigned (int)		$[0, 4294967295]$	%u
long (int)	64	$[-9223372036854775808, 9223372036854775807]$	%ld
unsigned long (int)		$[0, 18446744073709551616]$	%lu
float	32	$[-3.4e38, 3.4e38] \varepsilon = 1.2e-38$	%f
double	64	$[-1.8e308, 1.8e308] \varepsilon = 2.2e-308$	%lf

Types primitifs de variables

Type	Bits	Intervalle	Code printf
char	8	$[-128, 127]$	%hhd
unsigned char		$[0, 255]$	%hhu
short (int)	16	$[-32768, 32767]$	%hd
unsigned short (int)		$[0, 65535]$	%hu
int	32	<p>Dépend de l'architecture !</p> <p><code>sizeof(type)</code> pour connaître le nombre d'octets occupés par le type</p>	%d
unsigned (int)			%u
long (int)	64	$[-9223372036854775808, 9223372036854775807]$	%ld
unsigned long (int)		$[0, 18446744073709551616]$	%lu
float	32	$[-3.4e38, 3.4e38] \epsilon = 1.2e-38$	%f
double	64	$[-1.8e308, 1.8e308] \epsilon = 2.2e-308$	%lf

Constantes

Ajout du mot-clé `const` lors de la déclaration :

```
const int x = 3;
```

- ▶ Le compilateur empêche la variable d'être modifiée par la suite
- ▶ Permet au compilateur de réaliser des optimisations
- ▶ Permet de se protéger des erreurs d'inattention du développeur
- ▶ `const` est un *qualificateur de type*

Opérations arithmétiques

Principales opérations arithmétiques

Symbole	Opération
+	Addition
-	Soustraction
*	Multiplication
/	Division
%	Modulo

Exemple

```
int a = 3;  
int b = 4;  
int c = a * b;
```

Conversion de types

Toute expression étant typée, il faut convertir en cas de types hétérogènes entre deux opérandes.

- ▶ Par défaut : conversion implicite du type le plus petit vers le type le plus grand.
- ▶ Les constantes entières sont de type `int`.
- ▶ Les constantes flottantes sont de type `double`.
- ▶ Pour les affectations : conversion du type du membre droit vers le type du membre gauche. Si le type de destination est plus petit que le type du membre droit :
 - ▶ Résultat indéfini si un des types est flottant
 - ▶ Troncature pour les types entiers

Exemple

```
char a = 7;  
int b = 2;  
float c = a / b; // conversion de a en int, a/b = 7/2  
           = 3 , conversion de 3 en flottant
```

Conversion de types explicite

Il est possible de forcer la conversion d'un type (« cast ») :

Exemple

```
char a = 7;
int b = 2;
float c = (float) a / b; // conversion de a en float,
                        // conversion de b en float, a/b = 7/2 = 3.5
```

Question

Que va afficher le code suivant ? Pourquoi ? Indice : $245895 \times 478565 > 2^{31} - 1$

```
int var0 = 245895;
int var1 = 478565;

long val2 = (long) var0 * var1;
long val3 = (long) (var0 * var1);

printf("val2 = %ld\nval3 = %ld\n", val2, val3);
```

Affectations combinées

Dans le cas où le contenu de la variable est modifié par l'opération :

$$\text{var} = \text{var op expr}; \Leftrightarrow \text{var op} = \text{expr};$$

Exemple

```
x += 3;  
y -= 2;  
z *= 4;
```


Incrémentations et décréments

Dans le cas où on incrémente ou décrémente :

`var += 1; ⇔ var++;`

`var -= 1; ⇔ var--;`

Selon la position de l'opérateur, la valeur est lue avant ou après l'opération :

`var++;` ⇒ Post-incrémentation

`++var;` ⇒ Pré-incrémentation

Exemples

Code	Incrémentation	Valeurs finales
<code>i = 3; a = i++;</code>	Post-incrémentation	a = 3 et i = 4
<code>i = 3; a = ++i;</code>	Pré-incrémentation	a = 4 et i = 4
<code>i = 3; a = i--;</code>	Post-décrémentation	a = 3 et i = 2
<code>i = 3; a = --i;</code>	Pré-décrémentation	a = 2 et i = 2

Opérateurs bit-à-bit

- ▶ `&` (ET) , `|` (OU) , `^` (OU exclusif : xor) bit-à-bit
- ▶ `~` complément à un (inversion de bits)
- ▶ `var << n` décalage de `n` bits vers la gauche (multiplication par 2^n)
- ▶ `var >> n` décalage de `n` bits vers la droite (division entière par 2^n)

Exemple : unsigned char

a	0	0	0	0	1	1	0	1	13
b	0	0	0	0	0	1	1	0	6

a & b	0	0	0	0	0	1	0	0	4
a b	0	0	0	0	1	1	1	1	15
a ^ b	0	0	0	0	1	0	1	1	11
~a	1	1	1	1	0	0	1	0	242
a >> 1	0	0	0	0	0	1	1	0	6
a << 3	0	1	1	0	1	0	0	0	104
3 >> a	0	0	0	0	0	0	0	0	0

Opérateurs bit-à-bit

Que va afficher le code suivant ?

```
#include <stdio.h>

int main()
{
    char a = 1;
    char b = 2;
    printf("a & b = %hhd\n", a & b);
    printf("a | b = %hhd\n", a | b);
    printf("a ^ b = %hhd\n", a ^ b);
    b = 4;
    printf("b << 2 = %hhd\n", b<<2);
    printf("b >> 2 = %hhd\n", b>>2);
    printf("2 >> b = %hhd\n", 2>>b);
    a = 255;
    printf("~a = %hhd\n", ~a);
    return 0;
}
```

Conditions

Si

Si expression est vraie
ALORS bloc d'instructions

```
if (expression)
{
    instruction1;
    instruction2;
}
```

Si ... Sinon

Si expression est vraie
ALORS bloc d'instructions
SINON autre bloc d'instructions

```
if (expression)
{
    instruction1;
    instruction2;
}
else
{
    instruction3;
}
```

Si ... Sinon Si ... Sinon

```
if (expression1)
{
    instruction1;
    instruction2;
}
else if (expression2)
{
    instruction3;
}
else
{
    instruction4;
}
```

Possibilité d'avoir plusieurs else if

Condition

Toute expression évaluée à 0 est fausse.

Toute expression évaluée à autre chose que 0 est vraie.

```
int nb_billes = 3;

if (nb_billes)
{
    printf("Une ou plusieurs billes\n");
}
else
{
    printf("Pas de bille\n");
}
```


Expressions booléennes

Opérateurs de comparaisons

Symbole	Signification
==	Égal
!=	Différent
<	Inférieur
<=	Inférieur ou égal
>	Supérieur
>=	Supérieur ou égal

Opérateurs booléens

Symbole	Signification
&&	Et
	Ou
!	Négation

Priorités des opérateurs

! > && > ||

Exemple :

$\neg A \vee B \wedge C \Leftrightarrow (\neg A) \vee (B \wedge C)$

Exemple

```
int nb_billes = 3;
int a_moi = 1;

if (a_moi && nb_billes)
{
    if (nb_billes > 1)
    {
        printf("J'ai plusieurs billes\n");
    }
    else
    {
        printf("J'ai une bille\n");
    }
}
else
{
    printf("Je n'ai pas de bille :(\n");
}
```

À vous de jouer !

Attention !

`==` \neq `=`

`!=` \neq `=!`

`&&` \neq `&`

`||` \neq `|`

Quelles sont les différences ?

Égalité de flottants

Représentation des nombres flottants

- ▶ Principal problème : représenter un nombre infini de nombres décimaux avec un nombre fini de bits
- ▶ Norme IEEE 754
- ▶ Précision limitée (eg $\varepsilon = 1.2e - 38$ pour float)

Test de l'égalité de flottants (question de partiel)

- ▶ Conséquence de la représentation : les erreurs de précision se propagent lors d'opérations sur les flottants
- ▶ Comparaison avec les fonctions fabs (double) ou fabsf (float) :

```
if (fabsf(b - a) < 1e-10)
{
    printf("a == b\n");
}
else
{
    printf("a != b\n");
}
```

switch

```
switch (expression)
{
    case 0:
        instructions;
        break;
    case 1:
        instructions;
        break;
    default:
        instructions;
        break;
}
```

- ▶ Surtout utilisé pour les ensembles finis de possibilités (*cf* énumérations, par exemple)
- ▶ Peut tester uniquement des valeurs de nombres entiers

switch - Exemple

```
int nb_billes = 3;

switch (nb_billes)
{
    case 0:
        printf("Je n'ai pas de bille :(\n");
        break;
    case 1:
        printf("J'ai une bille\n");
        break;
    default:
        printf("J'ai plusieurs billes\n");
        break;
}
```

À vous de jouer !

Conditions ternaires

Expressions de la forme :

```
expression ? expression_si_vrai : expression_si_faux
```

Exemple d'utilisation :

```
int nb_billes = droit_billes ? 3 : 0;
printf(
    nb_billes > 0 ?
        "J'ai une ou plusieurs billes\n" :
        "Je n'ai pas de billes\n"
);
```

Possibilités de chaîner les ternaires (mais attention à la lisibilité!) :

```
printf(
    nb_billes == 0 ?
        "Je n'ai pas de billes :(\n" :
    nb_billes == 1 ? "J'ai une bille\n" :
        "J'ai plusieurs billes\n"
);
```

Boucles

Tant que

TANT QUE expression est vraie
ALORS bloc d'instructions

```
while (expression)
{
    instruction1;
    instruction2;
}
```

```
int nb_billes = 3;
int i = 0;

while (i < nb_billes)
{
    printf("J'ai la bille numéro %d\n", i+1);
    i++;
}
```

Tant que – variante

EXÉCUTE bloc d'instructions
TANT QUE expression est vraie

```
do
{
    instruction1;
    instruction2;
} while (expression);
```

Exécute au moins une fois le bloc d'instructions.

Attention au point-virgule !

Quelle est la différence avec la version précédente ?

Boucles infinies

Lorsque la condition de la boucle est toujours vraie :

- ▶ Le programme est bloqué dans la boucle.
- ▶ Ce n'est (généralement) pas le comportement désiré.
- ▶ Oubli de changer la valeur d'une variable utilisée dans la condition, mauvais algorithme, ...
- ▶ Ctrl+C pour arrêter le programme

La boucle for

```
for (inst_init; expr_vraie; inst_a_chaque_iter)
{
    instruction1;
    instruction2;
}
```

Quasiment toujours utilisée avec un compteur :

```
int nb_billes = 3;
int i;

for (i = 0; i < nb_billes; i++)
{
    printf("J'ai la bille numéro %d\n", i+1);
}
```

- ▶ Quelle est la portée de la variable `i` ?
- ▶ Quelle est la valeur de `i` à la sortie de la boucle ?

À vous de jouer !

Exercice : nombre de bits à 1

Consigne

Écrivez un programme qui compte le nombre de bits à 1 dans un nombre.

Conseils

- ▶ `sizeof(type)` pour connaître le nombre d'*octets* du type
- ▶ Boucle pour itérer sur chaque bit
- ▶ Opérateur bit-à-bit pour savoir si le bit est à 1 ou pas

À vous de jouer !

Instruction break

Permet de sortir de la boucle ou du **case** le plus interne.

```
for (i = 0; i < 3; i++)
{
    int reponse = obtenir_reponse();

    if (reponse < 0)
    {
        break;
    }

    traiter_reponse(reponse);
}
```

Non-recommandé (dans les boucles) : fait du code « spaghetti », plus difficile à comprendre.

Instruction continue

Permet d'interrompre l'itération courante de la boucle et aller à l'itération suivante.

```
for (i = 0; i < 3; i++)  
{  
    int reponse = obtenir_reponse();  
  
    if (reponse < 0)  
    {  
        continue;  
    }  
  
    traiter_reponse(reponse);  
}
```

Quelle est la différence de comportement par rapport à l'exemple précédent ?

Non-recommandé : fait du code « spaghetti », plus difficile à comprendre.

Exercice : overflow sur les entiers

Consigne

Écrivez un programme qui détermine l'intervalle des nombres entiers représentables avec le type `char`.

Faites une version avec `break` et une version sans.

Conseils

- ▶ Partez d'un `char` avec la valeur 0, incrémentez de 1 jusqu'à ce que la valeur devienne négative
- ▶ Stockez les valeurs maximales et minimales obtenues

À vous de jouer !

Fonctions

Fonctions

- ▶ Découpage du programme en sous-programmes : *fonctions*
- ▶ Permet de *factoriser* le code
 - ▶ Évite le copier-coller
 - ▶ Simplifie le code
- ▶ Peut être vue comme une fonction mathématique :
 - ▶ à partir de paramètres...
 - ▶ ... la fonction exécute des instructions...
 - ▶ ... et renvoie une valeur
- ▶ Il y a aussi des fonctions sans paramètres et/ou sans valeur de retour : uniquement une séquence d'instructions.

Définitions de fonctions

Une fonction possède :

- ▶ un type de retour
- ▶ un nom
- ▶ une liste de valeurs typées comme paramètres
- ▶ un corps qui contient les instructions à exécuter

Syntaxe :

```
type_retour nom_fonction(type1 param1, type2 param2)
{
    instruction;
    instruction;
}
```

- ▶ Les fonctions doivent être définies avant leur utilisation (pour l'instant).
- ▶ Il est impossible de définir une fonction *dans* une fonction (sans utiliser d'extension GNU).

Paramètres de fonctions

- ▶ Les paramètres d'une fonction s'utilisent comme des variables définies au début de la fonction.

- ▶ Si pas de paramètres :

`type_retour foo()`

ou (mieux) :

`type_retour foo(void)`

- ▶ À partir de C23, `foo()` \Rightarrow `foo(void)`
- ▶ Paramètres passés *par copie*
 - ▶ Les modifications faites aux paramètres dans la fonction ne sont pas répercutées dans les variables de la fonction appelante.

Valeur de retour

- ▶ L'instruction `return expression`; indique que l'exécution de la fonction est terminée et qu'elle renvoie (*retourne*) la valeur de `expression`.
 - ▶ On ne peut retourner qu'une valeur.
 - ▶ `return` peut être placé n'importe où dans la fonction, plusieurs fois
- ▶ `void` est à utiliser pour indiquer qu'il n'y a pas de valeur de retour
 - ▶ `return`; est alors facultatif

```
int max(int a, int b)
{
    int m = a;
    if (b > a)
    {
        m = b;
    }
    return m;
}
```

```
void dire_bonjour(void)
{
    printf("Bonjour !\n");
}
```

Question : pourquoi la fonction `main` renvoie une valeur ?

Appel de fonctions

```
int max(int a, int b)
{
    // ...
}

void dire_bonjour(void)
{
    // ...
}

int main()
{
    int x = 5;
    int y = 7;

    dire_bonjour();

    int m = max(x, y);
    printf("Le maximum est %d\n", m);

    return 0;
}
```

Exercice : le retour du nombre de bits

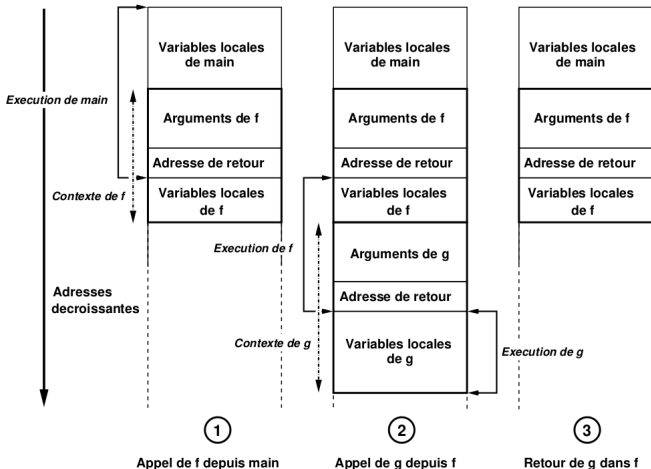
Consigne

Reprenez l'exercice du nombre de bits, mais créez une fonction `int nb_bits_char(char x)` qui renvoie le nombre de bits à 1 dans `x`.

À vous de jouer !

Le mécanisme derrière un appel de fonction

- Les contextes des fonctions s'empilent et se dépilent au fur et à mesure des entrées et sorties dans les fonctions, dans une zone mémoire appelée **pile**



Attention ! Dépend de l'architecture du processeur.

Ce schéma décrit le fonctionnement de l'architecture x86-32 bits.

Exercice : Fibonacci

Consigne

Écrivez une fonction (récursive) qui renvoie le $n^{\text{ème}}$ terme de la suite de Fibonacci, définie par :

$$F(n) = \begin{cases} 0 & \text{si } n = 0 \\ 1 & \text{si } n = 1 \\ F(n-1) + F(n-2) & \text{sinon} \end{cases}$$

Conseil

- La fonction sera récursive : ne pas oublier la condition d'arrêt !

Question : Peut-on calculer tous les termes ? Pourquoi ?

À vous de jouer !

Prototypes

Ce code fonctionne-t-il ? Pourquoi ?

```
int est_pair(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return est_impair(n-1);
}

int est_impair(int n)
{
    if (n == 0)
    {
        return 0;
    }
    return est_pair(n-1);
}
```

Prototypes

```
int est_pair(int n);  
int est_impair(int n);
```

Déclaration de *prototypes*
avant l'utilisation des fonctions.

```
int est_pair(int n)  
{  
    if (n == 0)  
    {  
        return 1;  
    }  
    return est_impair(n-1);  
}
```

```
int est_impair(int n)  
{  
    if (n == 0)  
    {  
        return 0;  
    }  
    return est_pair(n-1);  
}
```

Disgression

```
int est_pair(int n);
int est_impair(int n);

int est_pair(int n)
{
    if (n == 0)
    {
        return 1;
    }
    return est_impair(n-1);
}

int est_impair(int n)
{
    if (n == 0)
    {
        return 0;
    }
    return est_pair(n-1);
}
```

Questions

- ▶ Ce code est-il efficace ?
- ▶ Proposez d'autres façons de tester si un nombre est pair

Fonctions du langage C

- ▶ Le langage C fournit un ensemble de fonctions dans la *bibliothèque standard*.
- ▶ Peut nécessiter d'ajouter des lignes `#include <...h>` au début du fichier source (on verra plus tard pourquoi)
- ▶ Informations sur les fonctions (prototypes, fonctionnement, signification des paramètres, valeurs de retour possible, exemples, `#include` nécessaires, ...) dans les pages de manuel (*manpages*) :
 - ▶ `man 3 <fonction>`
Exemple : `man 3 printf`
 - ▶ Peut-être que vous avez besoin d'installer les paquets `manpages` et `manpages-dev` (pour les avoir en français : `manpages-fr` et `manpages-fr-dev` ; pour Debian, adaptez à votre distribution)

Pointeurs

Pointeurs, références, adresses, variables, mémoire...

Variables

- ▶ Les variables peuvent être vues comme des étiquettes sur des cases mémoires.
- ▶ Les cases mémoires stockent la valeur de la variable.
- ▶ Ces étiquettes sont propres au programme, voire à la fonction.

Adresse

- ▶ Chaque case mémoire possède une adresse.
- ▶ Chaque case mémoire possède 1 octet : impossible d'avoir l'adresse d'un bit en particulier.
- ▶ L'adresse de la case étiquetée par une variable s'obtient avec l'opérateur &.
- ▶ Affichage dans `printf` avec `%p`.

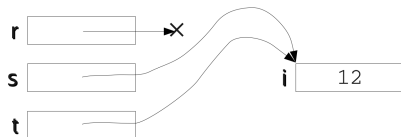
Pointeur

- ▶ Les adresses sont stockées dans des variables de type pointeurs :
`type* variable`
- ▶ Toujours la même taille

Pointeurs

Initialisations

- ▶ `int* r = NULL;`
- ▶ `int i = 12; int* s = &i;`
- ▶ `int* t = s;`



`type* var :`

- ▶ se lit « `var` référence / pointe sur une variable de type `type` » ;
- ▶ contient l'adresse de la première case mémoire stockant des données de type `type`.

Pointeurs

Utilisation

- Pour accéder au contenu de l'adresse stockée (*déréférencement*) : on utilise l'opérateur `*`.
- En cas de tentative d'accès à une adresse interdite : erreur de segmentation (*segmentation fault* / *segfault*) et le programme est interrompu.

Exemple

On souhaite multiplier par 2 la variable avec la plus petite valeur :

```
int a = 5;
int b = 17;
int* min_ref = NULL;
min_ref = (a < b) ? &a : &b;
(*min_ref) *= 2;
printf(
    "La plus petite valeur doublée est %d\n",
    *min_ref
);
```

Pointeurs de pointeurs

```
int a;  
int* p;  
int** q;  
  
a = 5;  
p = &a;  
q = &p;  
**q = 7; /* Maintenant a vaut 7 */
```



Pointeurs comme paramètres de fonctions

Pourquoi le code suivant ne peut pas fonctionner ?

```
void swap(int a, int b)
{
    int temp = a;
    a = b;
    b = temp;
}

int main()
{
    int a = 12;
    int b = 7;

    printf("a = %d, b = %d\n", a, b);
    swap(a, b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Pointeurs comme paramètres de fonctions

Pourquoi le code suivant fonctionne? (question de partiel)

```
void swap(int* a, int* b)
{
    int temp = *a;
    *a = *b;
    *b = temp;
}

int main()
{
    int a = 12;
    int b = 7;

    printf("a = %d, b = %d\n", a, b);
    swap(&a, &b);
    printf("a = %d, b = %d\n", a, b);

    return 0;
}
```

Une fonction ne peut pas renvoyer l'adresse d'une variable locale

Quel est le problème de ce code ?

```
void toto(int a, int b, int c, int d, int e)
{
    // ...
}

int* foo(int i, int j)
{
    int a = j * i ;
    int* b = &a;
    return b;
}

int main()
{
    int* ptr = foo(3, 2);
    toto(10, 11, 12, 13, 14);
    printf("%d\n", *ptr);

    return 0;
}
```

```

#include <stdio.h>

int* addr_max(int* a, int* b)
{
    int* m = a;
    if (*b > *a)
    {
        m = b;
    }
    return m;
}

int main()
{
    int i = 3;
    int j = 7;

    int* p_max = addr_max(&i, &j);
    printf("i    = %d (%p)\n", i, &i);
    printf("j    = %d (%p)\n", j, &j);
    printf("max  = %d (%p)\n", *p_max, p_max);

    return 0;
}

```

- ▶ Que fait ce code ?
- ▶ Pourquoi / comment fonctionne-t-il ?

Zones mémoires

Toute zone mémoire stockant une variable peut-être décrite par :

- ▶ son adresse de début ;
- ▶ sa taille (dépend du type de variable).



Pointeur générique

Dans le cas où on n'est pas intéressé par le type, qu'on a juste besoin de l'adresse :

`void*` est un type pour stocker n'importe quelle adresse

Question

Pourquoi ne peut-on pas déréférencer un pointeur générique `void*` ?

Tableaux

Tableaux

Séquence d'éléments

- ▶ Longueur définie à l'initialisation
- ▶ Indices commencent à 0
- ▶ Données de type homogène, stockées de façon contiguë en mémoire
- ▶ **Impossible de connaître la longueur d'un tableau existant**
(question de partiel) (possible seulement pour certains types de tableaux)
- ▶ Bornes non vérifiées (possibilité d'écrire en-dehors du tableau)

Tableaux

Déclaration

```
int t[4];  
int u[4] = {4, 5, 3, 2};  
int v[] = {4, 5, 3, 2};
```

La taille fournie à l'initialisation doit toujours être une constante
(cas variable vu plus tard)

Accès à une case

```
u[2] = 1;  
printf("u[2] = %d\n", u[2]);
```

Que contient tout le tableau maintenant ?

Exercice : calcul du maximum

Consigne

Compléter le code suivant pour que `maximum` contienne le maximum du tableau `nombres`.

```
int nombres[9] = {5, 4, 2, 1, 9, 4, 3, 8, 3};  
int maximum;  
  
// ...  
  
printf("Le maximum est : %d\n", maximum);
```

À vous de jouer !

Tableau, pointeur...

Un tableau est un pointeur !

```
int t[7];  
int* p = t;
```

- ▶ `t` contient en réalité l'adresse mémoire de `t[0]`
- ▶ `t` est en réalité de type `int*`
- ▶ `p` contient l'adresse `&t[0]`
- ▶ `t[i]` est équivalent à `*(t + i*sizeof(int))`

Exemple

Que va afficher le code suivant ?

```
int t[3] = {1, 4, 2};  
int* u;  
  
u = t;  
u[1] = 0;  
  
printf("t[1] = %d u[1] = %d\n", t[1], u[1]);
```

Exemple

```
int t[5] = {1, 4, 2, 8, 9};  
int* u;  
  
u = &t[2];  
u[1] = 0;
```

Quelle case de `t` contient la valeur 0 ?

Généralisation : arithmétique des pointeurs

```
int t[7];  
int* p = t;
```

$p+i \Leftrightarrow \&p[i]$ et $*(p+i) \Leftrightarrow p[i]$

D'où l'importance du type des pointeurs :

+i sur un pointeur signifie $+i * \text{sizeof}(\text{type})$

Conséquence : pas d'arithmétique possible sur un pointeur générique `void*` (utilisez un `char*` en cas de besoin).

Retour à l'exemple

Quelle case de `t` contient 0 ?

```
int t[5] = {1, 4, 2, 8, 9};  
int* u;  
  
u = t + 2  
*(u+1) = 0;
```

Pointeur générique et arithmétique

Exemple

```
void my_memcpy(void* dest, void* src, size_t len)
{
    char* d = dest;
    char* s = src;
    for (int i = 0; i < len; i++)
    {
        *(d+i) = *(s+i);
    }
}
```

Questions

- ▶ Que fait cette fonction ?
- ▶ Pourquoi ne peut-on pas directement utiliser `src` et `dest` ?
- ▶ Comment écrire ce code sans se servir de `i` dans le corps de la boucle ? Indice : penser à l'incréméntation.
- ▶ Comment écrire ce code sans compteur `i` ?

Tableaux et fonctions

- ▶ Un tableau est un pointeur : on peut passer ce pointeur en paramètre à une fonction.
- ▶ On ne peut retourner qu'un pointeur vers un tableau existant avant l'appel à la fonction.

Exemple

```
int* bar(int n, int t1[], int t2[])
{
    if (sum(n, t1) > sum(n, t2))
        return t1;
    else
        return t2;
}
```

Exercice

Consigne

1. Faites une fonction `print_array` qui affiche le contenu de toutes les cases d'un tableau d'entiers.
2. Faites une fonction `max_array` qui renvoie la valeur maximum contenue dans un tableau d'entiers.
3. Faites une fonction `array_with_max` qui, parmi deux tableaux de même taille passés en paramètres, renvoie le tableau contenant la plus grande valeur maximale.
4. Créez deux tableaux d'entiers, affichez-les et affichez celui avec la plus grande valeur maximale.

Question

- Quel est le paramètre que toutes ces fonctions vont avoir en commun ? Pourquoi ?

À vous de jouer !

Chaînes de caractères

Représentation des caractères

- ▶ Comme toujours : ensemble de bits, tout est dans l'interprétation qu'on en fait
- ▶ Pour les caractères alpha-numériques, une des normes est l'ASCII
 - ▶ Par exemple, le code pour le caractère A est 65
 - ▶ `man ascii`
- ▶ Type utilisé pour stocker *un* caractère : `char`
- ▶ Les caractères s'expriment entre simples guillemets : `char c = 'A';`
- ▶ Interprétation comme un caractère dans `printf` : `%c`
 - ▶ Rappel : `%hhd` est pour interpréter comme un entier

Questions

- ▶ Quelle est la différence entre `char c = 65;` et `char c = 'A';` ?
- ▶ Que va-t-il s'afficher si j'utilise `%c` ou `%hhd` sur les deux valeurs ?
- ▶ Mêmes questions avec `char c = 57;` et `char c = '9';`.

Quelques caractères particuliers

Caractère	Signification
'\0'	Caractère nul
'\n'	Nouvelle ligne
'\t'	Tabulation
'\\'	Anti-slash
'\''	Guillemet simple

- ▶ Il s'agit bien d'un unique caractère à chaque fois !
- ▶ \ est appelé « caractère d'échappement ».

Exercice

Consigne

1. Écrivez une fonction `my_isalpha` qui indique si un caractère est une lettre (comme la fonction `isalpha` avec `#include <ctype.h>`).
2. Écrivez une fonction `my_tolower` qui renvoie la lettre minuscule correspondant à la lettre passée en paramètre (comme la fonction `tolower` avec `#include <ctype.h>`).
 - Que faire lorsque le paramètre n'est pas une lettre ?

À vous de jouer !

Chaînes de caractères

- ▶ Pas de type « chaîne de caractères » en C.
- ▶ Une chaîne de caractères est un tableau de caractères (`char`) dont le dernier caractère est le caractère nul (`'\0'`).
 - ▶ Ne pas oublier de réserver la place du caractère nul !
- ▶ Affichage avec `printf` : `%s`
- ▶ À quoi sert le dernier caractère nul ?

Chaînes de caractères

Une chaîne de caractères est un tableau de caractères (`char`) dont le dernier caractère est le caractère nul (`'\0'`) :

```
char t[] = {'B', 'o', 'n', 'j', 'o', 'u', 'r', '\n', '\0'};
```

La notation avec des guillemets doubles ("`foo`") est plus pratique (`\0` est automatiquement ajouté) :

```
char t[] = "Bonjour\n";
```

On peut aussi avoir des pointeurs vers des chaînes de caractères, **mais elles sont alors constantes** :

```
char* s = "Bonjour !";
```

Stockage en mémoire des chaînes de caractères

- ▶ Une déclaration de la forme `char t[] = "foo"` alloue de la mémoire pour un tableau **sur la pile**
 - ▶ C'est la zone mémoire propre à la fonction, on peut lire, écrire, modifier cette mémoire
- ▶ Une déclaration de la forme `char* s = "bar"` stocke dans la variable `s` l'adresse de `"bar"` stockée **dans une zone en lecture seule** de la mémoire du programme

Chaînes de caractères

Exemple

```
char t[] = "Bonjour\n" ; /* Tableau modifiable en mémoire */
char * s = "Salut!";      /* Pointeur sur une constante */
t[1] = 'Z';               /* Légal car t est modifiable */
s[1] = 'Z';               /* Segfault car zone constante */
s = t;                    /* Légal car s est une variable */
s[1] = 'A';               /* Légal car t[1] est modifiable */
```

Fonctions de manipulation de chaînes

Nécessaire d'ajouter au début du fichier (on verra plus tard pourquoi) :

```
#include <string.h>
```

- ▶ `strlen` : renvoie le nombre de caractères dans une chaîne (sans le caractère terminal)
 - ▶ Quel est l'algorithme de cette fonction ?
- ▶ `strcpy` : copie d'une chaîne de caractères vers un tableau de destination
- ▶ `strcat` : ajoute une chaîne de caractères à la fin d'une autre
- ▶ `strchr` : recherche la première occurrence d'un caractère dans une chaîne
- ▶ `strcmp` : compare deux chaînes de caractères
- ▶ et beaucoup d'autres : `man string`

Comment passer une chaîne de caractères comme paramètre à une fonction ?

Exercice

Consigne

Codez votre fonction `my_strlen` qui renvoie le nombre de caractères dans une chaîne de caractères.

À vous de jouer !

Retour sur la fonction `main`

Son prototype (presque) complet est le suivant :

```
int main(int argc, char* argv[]);
```

- ▶ Elle renvoie un entier.
 - ▶ Pourquoi ? (on en a déjà parlé)
 - ▶ Elle prend deux arguments :
 - ▶ `argc` : le nombre d'éléments dans `argv`
 - ▶ `argv` : un tableau de pointeurs de chaînes de caractères, correspondant aux arguments du programme.
- La première chaîne (`argv[0]`) est toujours le nom du programme.**

Exemple

```
./programme argument1 argument2 argument3
```

```
int main(int argc, char* argv[])
{
    printf("Les arguments du programme sont :\n");
    for (int i = 1; i < argc; i++)
    {
        printf("%s\n", argv[i]);
    }
    return 0;
}
```

Exercice

Consigne

Reprenez votre fonction `my_strlen` et écrivez un programme qui affiche les arguments passés au programme et la longueur des chaînes correspondantes.

À vous de jouer !

Conversion des chaînes de caractères en nombres

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

```
int  atoi(char* str);  
long atol(char* str);  
long long atoll(char* str);
```

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

Exemple

```
char* str = "12";  
int nb = atoi(str);  
printf("%s = %d\n", str, nb);
```

Énumérations

Énumérations

- ▶ Sous-ensemble du type `int` auquel est associé un nombre fini de valeurs symboliques.
- ▶ Évite d'avoir à se souvenir quelle signification a telle valeur numérique.
- ▶ Permet de connaître l'ensemble entier des valeurs possibles.
- ▶ Type de variable défini par le développeur.
- ▶ Déclaration à l'extérieur de toute fonction.

```
enum feu_e {  
    FEU_VERT ,  
    FEU_ORANGE ,  
    FEU_ROUGE  
};
```

Utilisation

```
void affiche_feu(enum feu_e feu)
{
    switch (feu)
    {
        case FEU_VERT:
            printf("Vert ! Go !\n");
            break;
        case FEU_ORANGE:
            printf("Orange !\n");
            break;
        case FEU_ROUGE:
            printf("Rouge !\n");
            break;
        default:
            printf("Feu invalide\n");
            break;
    }
}
```

```
enum feu_e feu = FEU_ROUGE;
affiche_feu(feu);
affiche_feu(FEU_ORANGE);
```

Énumérations comme des entiers

Les énumérations sont des entiers : par défaut le premier élément vaut zéro.

```
enum feu_e {
    FEU_VERT,
    FEU_ORANGE,
    FEU_ROUGE,
    _NB_FEUX
};

void affiche_feu(enum feu_e feu)
{
    char* feux_str[] = {"vert", "orange", "rouge"};
    if (feu >= _NB_FEUX)
    {
        printf("bug !\n");
    }
    else
    {
        printf("%s\n", feux_str[feu]);
    }
}
```

Énumérations comme des entiers

- ▶ Numérotation consécutive des constantes.
- ▶ On peut préciser explicitement leur valeur :

```
enum nombre_e {  
    UN = 1,  
    DEUX,           // 2  
    QUATRE = 4,  
    CINQ,           // 5  
    SEPT = 7  
};
```

Structures

Structures

- ▶ Type défini par le développeur
- ▶ Type de variable qui regroupe plusieurs variables, potentiellement de types différents
- ▶ Peuvent être retournées / copiées
- ▶ La taille doit être calculable par `sizeof()` :
 - ▶ Donc calculable à la compilation
 - ▶ Pas de structures récursives

Définition

À l'extérieur de toute fonction :

```
struct personne_s {  
    char* prenom;  
    int age;  
};
```


Structures

Définition

À l'extérieur de toute fonction :

```
struct personne_s {  
    char* prenom;  
    int age;  
};
```

Initialisation

```
struct personne_s bobby;  
bobby.prenom = "Bobby";  
bobby.age = 42;  
struct personne_s john = {"John", 53};  
struct personne_s jeanny = {  
    .age = 24,  
    .prenom = "Jeanny"  
};  
struct personne_s jeanny_twin = jeanny;
```

Structures

Utilisation

```
jeanny_twin.prenom = "Katy";  
printf(  
    "%s a %d ans\n",  
    jeanny.prenom,  
    jeanny.age  
);
```

Exercice

Consigne

Écrivez une fonction qui à partir d'un nombre de minutes renvoie une structure contenant la décomposition en heures et minutes.

- ▶ Le nombre de minutes à décomposer sera un paramètre passé au programme.

Question bonus

Comment peut-on autrement obtenir cette décomposition en heures et minutes sans utiliser de structure ?

À vous de jouer !

Taille des structures

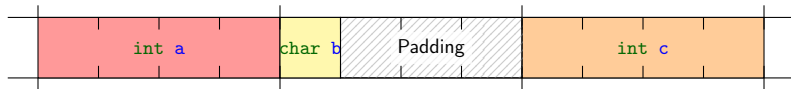
```
struct foo_s {  
    int a;  
    char b;  
    int c;  
};  
  
int main()  
{  
    struct foo_s foo;  
  
    printf("int %lu\n", sizeof(foo.a));  
    printf("char %lu\n", sizeof(foo.b));  
    printf("struct foo_s %lu\n", sizeof(foo));  
  
    return 0;  
}
```

Que va afficher le code suivant ?

Taille des structures

```
struct foo_s {  
    int a;  
    char b;  
    int c;  
};
```

Représentation en mémoire :



- ▶ Meilleures performances lorsque les adresses sont alignées sur un certain multiple
- ▶ Ajout d'octets de bourrage (*padding*) pour aligner les adresses des différents membres des structures

Pointeurs et structures

- Structures récursives : utilisation d'un pointeur

```
struct int_list_item_s {  
    int value;  
    struct int_list_item_s* next;  
};
```

Pourquoi peut-on faire ça ?

- Accéder aux membres d'un pointeur sur une structure :

$(*x).y \Leftrightarrow x->y$

```
void append(struct int_list_item_s* item,  
            struct int_list_item_s* other)  
{  
    item->next = other;  
}
```

Différence entre | et ||

Que va-t-il se passer à l'exécution du code suivant ? Pourquoi ?

```
struct foo_s {
    int i;
};

int main()
{
    int a = 12;
    struct foo_s* b = NULL;
    if (a | b->i) // | devrait être remplacé par ||
    {
        printf("a ou b->i ou les deux sont vrais\n");
    }
    else
    {
        printf("a et b->i sont faux\n");
    }

    return 0;
}
```

Allocation dynamique de mémoire

Allocation dynamique

Deux cas peuvent se présenter :

- ▶ On ne connaît pas à la compilation la quantité de mémoire dont on va avoir besoin, information connue seulement lors de l'exécution
- ▶ On a besoin d'une zone mémoire qui survivra à la fin de la fonction

⇒ allocation de la mémoire avec `malloc`

À ne pas faire

À ne pas faire !

On pourrait être tenté de faire :

```
int n = 1000; // le résultat d'un calcul à l'exécution  
int tab[n];
```

À ne pas faire !

- ▶ Ça va compiler...
- ▶ Et même fonctionner pour les petits `n`
- ▶ Mais :
 - ▶ Que se passe-t-il si `n` est trop grand ?
 - ▶ Ne répond pas au second cas : `tab` ne survit pas après la fonction

À ne pas faire !

Fonction malloc

Réserve un certain nombre d'octets **dans le tas** : zone mémoire spécifique lors de l'exécution du programme

- ▶ En paramètre le nombre d'octets à allouer (pensez à `sizeof`)
- ▶ En retour : pointeur sur le début de la zone allouée (non initialisée), ou `NULL` en cas d'erreur
 - ▶ Quelle erreur peut-il se produire?
- ▶ Nécessite `#include <stdlib.h>`

Exemple

```
int n = 1024;
int* tableau = malloc(n*sizeof(int));
if (tableau == NULL)
{
    printf("Erreur lors de l'allocation\n");
    // Gérer l'erreur ...
}

for (int i = 0; i < n; i++)
{
    tableau[i] = i;
}
```

Libération de mémoire avec `free`

Toute zone mémoire allouée dynamiquement doit être libérée !

```
free( tableau );
```

- ▶ Nécessite `#include <stdlib.h>`
- ▶ Une zone mémoire ne peut être libérée qu'une seule fois
- ▶ `free(NULL)` ne fait rien

Autres fonctions pour l'allocation dynamique

Allocation avec initialisation

```
void *calloc(size_t n, size_t size);
```

Alloue la mémoire pour stocker de façon contiguë `n` éléments de taille `size`, et remplit toute la zone mémoire allouée avec des zéros.

Changer la taille d'une zone mémoire allouée dynamiquement

```
void *realloc(void *ptr, size_t size);
```

Redimensionne à `size` octets la zone mémoire pointée par `ptr` allouée dynamiquement avant.

Renvoie un pointeur vers la zone mémoire redimensionnée, qui peut être différent de la zone initiale.

Autres fonctions utiles

(Pas spécifiques à l'allocation dynamique)

Nécessitent `#include <string.h>`

Initialiser une zone mémoire

```
void *memset(void* p, int c, size_t n);
```

Remplit les `n` premiers octets de la zone mémoire pointée par `p` avec l'octet `c`.

Copier une zone mémoire

```
void *memcpy(void* dest, void* src, size_t n);
```

Copie `n` octets depuis la zone mémoire `src` vers la zone mémoire `dest`.

Résumé des zones mémoire d'un programme

► Pile (*stack*)

Variables locales des fonctions, empilement des appels de fonctions

► Tas (*heap*)

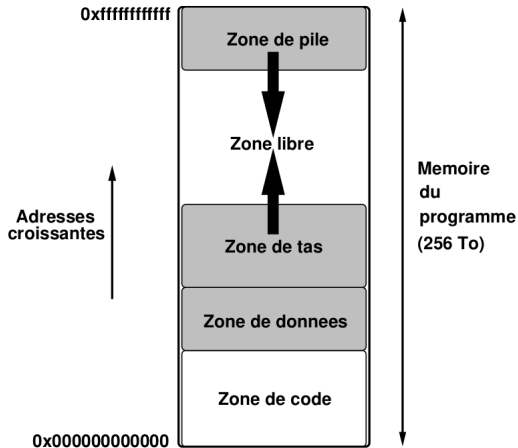
Mémoire allouée avec `malloc`

► Données

Variables globales et/ou statiques (*cf* plus loin)

► Code

Instructions du programmes, chaînes de caractères constantes, ...
En lecture seule



Visualiser la mémoire du programme

- ▶ Faites des dessins
- ▶ <https://pythontutor.com/c.html>

Question

Quelle est la différence entre

```
struct foo_s {  
    int tab[10];  
    int n; // vaudra 10  
};
```

et

```
struct foo_s {  
    int* tab; // tableau de taille n  
    int n;  
};
```

?

Préprocesseur

Préprocesseur

- ▶ Étape juste avant de vraiment compiler le programme : agit sur le code source sans en comprendre la sémantique
- ▶ `cc -E` ou `cpp` pour voir le résultat produit par le préprocesseur
- ▶ Instructions qui commencent par `#`, n'importe où dans le code
 - ▶ Portée limitée au fichier source
 - ▶ Pas une construction du langage C : pas de ; à la fin
 - ▶ Quelle instruction de préprocesseur a-t-on déjà utilisé ?
- ▶ Utile pour inclure des fichiers, définir des macros et faire de la compilation conditionnelle
- ▶ Par convention, tout ce qui est défini avec le préprocesseur est complètement en majuscules

Macros - Constantes

- ▶ Remplacement textuel
- ▶ **Attention aux priorités !**

```
#include <stdio.h>

#define FOO 42
#define BAR (FOO + 3)
#define BAZ FOO + 3
#define TEXT "du texte en préprocesseur"

int main(int argc, char* argv[])
{
    printf("%d\n", FOO);
    printf("%d\n", BAR * 2);
    printf("%d\n", BAZ * 2);

    printf("Voilà "TEXT"\n");

    return 0;
}
```

Que va-t-il s'afficher ?

Macros - Constantes pas vraiment constantes

- ▶ `__LINE__` : numéro de ligne
- ▶ `__func__` : nom de la fonction (depuis C99)
- ▶ `__FILE__` : nom du fichier source
- ▶ `__DATE__` : date de compilation

```
1 #include <stdio.h>
2
3 int main()
4 {
5     printf(
6         "[%s:%d %s()] Compilé le %s\n",
7         __FILE__,
8         __LINE__,
9         __func__,
10        __DATE__
11    );
12
13    return 0;
14 }
```

```
[fichier.c:8 main()] Compilé le Oct 13 2023
```

Macros - "Fonctions"

```
#define MAX(__x, __y) \  
    ((__x) > (__y) ? (__x) : (__y))
```

- ▶ \ permet de continuer la définition de la macro sur la ligne suivante
- ▶ Pourquoi autant de parenthèses ? Pensez à `#define MULT(x, y) x*y`

```
int m = MAX(foo(), bar());  
  
int f = foo(), b = bar();  
int mm = MAX(f, b);
```

- ▶ Combien y a-t-il d'appels à `foo` et `bar` dans les deux cas ? Dans quel cas cela peut-il poser problème ?

Macros - "Fonctions"

- ▶ ## permet de concaténer un terme et un paramètre

```
#define FUNCTIONPLUS(type)      \  
type plus_##type(type a, type b)  \  
{                                  \  
    return (a + b);              \  
}  
  
FUNCTIONPLUS(int)    // Crée la fonction plus_int()  
FUNCTIONPLUS(float) // Crée la fonction plus_float()
```

#ifdef

- ▶ Permet d'inclure du code si une macro est définie ou exclure sinon
- ▶ Format :
 - ▶ `#ifdef ... #endif`
 - ▶ `#ifdef ... #else ... #endif`
- ▶ Si une macro n'est pas définie : `#ifndef`

```
#ifdef TYPE_DOUBLE
#define FLOTTANT double
#else
#define FLOTTANT float
#endif
```

```
// ...
```

```
FLOTTANT nombre = 1.8;
```


#if

- ▶ Comme `#ifdef`, mais évalue l'expression logique mêlant macros et constantes numériques

```
#define VERSION 2.0
#define TAILLE_INT 8

#if ((VERSION >= 1.3) || (TAILLE_INT == 8))
#define ENTIER long
#endif
```

`#if` `defined`

- Le mot-clé `defined` joue le même rôle que `#ifdef`, mais dans une expression `#if`

`#ifdef` \Leftrightarrow `#if` `defined`
`#ifndef` \Leftrightarrow `#if` `! defined`

```
#define VERSION 2.0
#define DEBUG

#if ((VERSION >= 1.3) && (defined DEBUG))
...
#endif
```

Définition des macros par la ligne de commande

On peut définir les macros :

- ▶ Dans les fichiers sources
- ▶ Lors de la compilation avec les paramètres `-Dnom` ou `-Dnom=valeur` passé au compilateur

```
#ifndef FLOTTANT
#define FLOTTANT float
#endif

// ...

FLOTTANT nombre = 1.8;
```

```
cc main.c
cc -DFLOTTANT=double main.c
```

Directive de préprocesseur `#include`

- ▶ Remplace la ligne par le contenu du fichier spécifié
- ▶ Deux formes :
 - ▶ `#include <fichier>` : inclut le fichier en le cherchant dans les répertoires connus du compilateur pour contenir des fichiers d'en-tête. Surtout utilisé pour inclure des en-têtes système.
 - ▶ `#include "fichier"` : comme `#include <fichier>`, mais commence d'abord par chercher dans le dossier du fichier source. Surtout utilisé pour inclure des en-têtes du programme.
- ▶ Deux façons d'ajouter des répertoires contenant les fichiers à inclure :
 - ▶ Option `-I../include/` au compilateur
 - ▶ Variable d'environnement `C_INCLUDE_PATH= "../include/"`

RTFM : <https://gcc.gnu.org/onlinedocs/cpp/Search-Path.html>

On n'inclut jamais de fichier `.c` !

cf la prochaine section pour comment gérer plusieurs fichiers `.c`

Doubles inclusions

grandparent.h :

```
struct foo_s {  
    int number;  
};
```

parent.h :

```
#include "grandparent.h"
```

enfant.c :

```
#include "grandparent.h"  
#include "parent.h"
```

Que se passe-t-il à la compilation de `enfant.c` ?

Protection contre les doubles inclusions

grandparent.h :

```
#ifndef __GRANDPARENT_H
#define __GRANDPARENT_H
struct foo_s {
    int number;
};
#endif // !__GRANDPARENT_H
```

parent.h :

```
#ifndef __PARENT_H
#define __PARENT_H
#include "grandparent.h"
#endif // !__PARENT_H
```

enfant.c :

```
#include "grandparent.h"
#include "parent.h"
```

Utiliser ce mécanisme de garde-fou pour tous les fichiers d'en-tête !

Compilation séparée

Compilation séparée

Pour les programmes plus conséquents, on organise le code en **plusieurs fichiers sources** :

- ▶ Plus simple à maintenir
- ▶ Meilleure organisation
- ▶ Peut rendre la compilation plus rapide
- ▶ Les fonctions qui vont ensemble sont dans le même fichier

Exemple de découpage

main.c :

```
int main()
{
    struct pers p;
    p.nom = "Camille";
    parler(&p, "Bonjour !");

    return 0;
}
```

personne.c :

```
#include <stdio.h>

struct pers {
    char* nom;
};

void parler(struct pers* p,
            char* msg)
{
    printf(
        "%s dit '%s'\n",
        p->nom,
        msg
    );
}
```

Compilation

```
cc main.c personne.c
```

Est-ce que ça compile ?

Fichiers d'en-tête

On place dans un **fichier d'en-tête** l'interface publique qu'offre le fichier `.c` correspondant :

- ▶ Prototypes de fonctions
- ▶ Types : structures, énumérations, ...
- ▶ Définition de constantes et de macros

Les éléments propres au fichier `.c` (pas utilisés par d'autres fichiers `.c`) ne vont **pas** dans les fichiers d'en-tête !

Forme de programmation *par contrat* : en gardant le même fichier d'en-tête, on doit pouvoir changer le fichier `.c` par un autre fichier `.c` qui implémente toutes les fonctions déclarées dans le fichier d'en-tête.

Par convention, les fichiers d'en-tête ont l'extension `.h` (pour `header`).

Exemple de découpage

main.c :

```
#include "personne.h"

int main()
{
    struct pers p;
    p.nom = "Camille";
    parler(&p, "Bonjour !");

    return 0;
}
```

personne.c :

```
#include <stdio.h>
#include "personne.h"

void parler(struct pers* p,
            char* msg)
{
    printf(
        "%s dit '%s'\n",
        p->nom, msg);
}
```

personne.h :

```
#ifndef __PERSONNE_H
#define __PERSONNE_H

struct pers {
    char* nom;
};

void parler(struct pers* p,
            char* msg);

#endif // !__PERSONNE_H
```

Compilation

```
cc main.c personne.c
```

Pourquoi ça compile ?

Autre découpage

main.c :

```
#include "personne.h"

int main()
{
    struct pers* p = creer_pers("
        Camille");
    parler(p, "Bonjour !");
    liberer_pers(p);

    return 0;
}
```

personne.h :

```
#ifndef __PERSONNE_H
#define __PERSONNE_H

struct pers;

struct pers* creer_pers(char* nom)
;
void liberer_pers(struct pers* p);
void parler(struct pers* p, char*
    msg);

#endif // !__PERSONNE_H
```

personne.c :

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

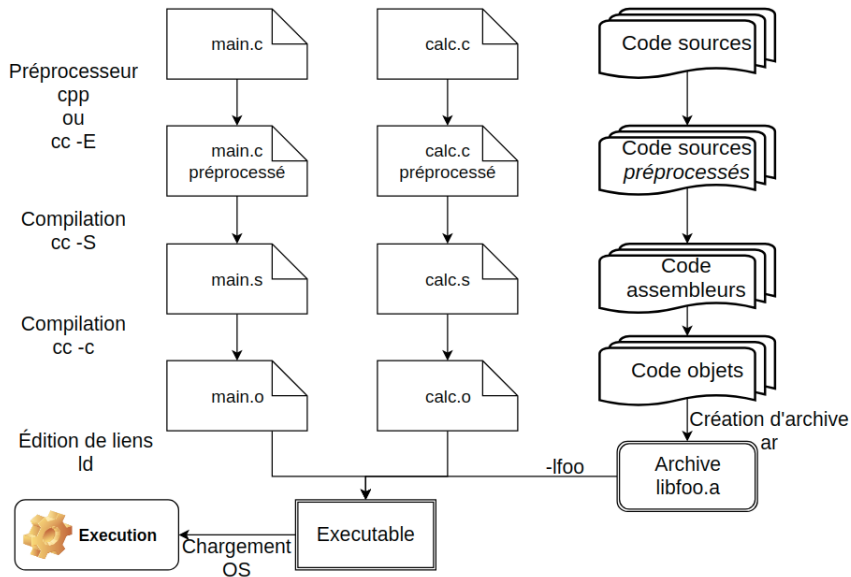
struct pers {
    char* nom;
};

struct pers* creer_pers(char* nom)
{
    struct pers* p = malloc(sizeof(
        struct pers));
    p->nom = nom;
    return p;
}

void liberer_pers(struct pers* p)
{
    free(p);
}

void parler(struct pers* p, char*
    msg)
{
    printf("%s dit '%s'\n", p->nom,
        msg);
}
```

Chaîne de compilation



Compilation séparée

```
cc -c main.c -o main.o  
cc -c personne.c -o personne.o  
cc main.o personne.o -o main
```

donnera le même résultat que la commande suivante :

```
cc main.c personne.c -o main
```

Possibilité de faire un script Bash ou utiliser l'outil `make` pour automatiser la première méthode.

Quel est l'avantage de la première méthode ?

Autre découpage - modification

Modifions le second découpage pour ajouter une énumération définissant comment `struct pers` parle :

```
enum ton_e {  
    PARLER,  
    CHUCHOTER,  
    CRIER,  
    HURLER  
};
```

- ▶ Quelle(s) modification(s) faut-il faire ?
- ▶ Dans quel(s) fichier(s) ?
- ▶ Où doit être déclarée l'énumération ?
- ▶ Que faut-il recompiler ?

Gestion de fichiers

Lecture et écriture de fichiers

Fonctionnement général

1. Ouverture du fichier → récupération d'un *descripteur de fichier*
2. Lecture / écriture
3. Fermeture du fichier

Deux interfaces disponibles

- ▶ Appels systèmes
 - ▶ Fonctions `open`, `close`, `read`, `write`, ...
 - ▶ Descripteur de fichier : `int`
 - ▶ Fonctions bas-niveau
 - ▶ Vues en IF210 Programmation système au S7
- ▶ **Fonctions de la bibliothèque C**
 - ▶ Fonctions `fopen`, `fclose`, `fread`, `fwrite`, `fprintf`, ...
 - ▶ Descripteur de fichier : `FILE*`
 - ▶ Vues dans la suite de ce cours

Ouverture d'un fichier

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

- ▶ **chemin** : chemin du fichier à ouvrir, absolu ou relatif par rapport au chemin depuis lequel est exécuté le programme
- ▶ **mode** :

Mode	Signification	Position	Existence du fichier
"r"	Lecture	Début	Doit déjà exister
"r+"	Lecture et écriture	Début	Doit déjà exister
"w"	Écriture, suppression du contenu déjà existant	Début	Peut être créé
"w+"	Lecture et écriture, suppression du contenu déjà existant	Début	Peut être créé
"a"	Écriture	Fin	Peut être créé

- ▶ renvoie un descripteur de type **FILE***

`man fopen`

Fermeture d'un fichier

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

- ▶ Quand on a fini de manipuler le fichier
- ▶ Toujours fermer un fichier ouvert : nombre de fichiers ouvert par processus limité (`ulimit -n`)

`man fclose`

Fonction d'écriture : `fwrite`

```
size_t fwrite(  
    void* buffer,  
    size_t taille,  
    size_t n,  
    FILE* fd  
);
```

- ▶ Écrit `taille`×`n` octets situés à partir de l'adresse `buffer` dans `fd`
- ▶ Renvoie le nombre d'éléments vraiment écrits (pas forcément le nombre d'octets, peut être différent de ce qui est demandé en cas d'erreur!)
- ▶ Déplace le curseur dans le fichier de ce qui a été écrit

Exemple d'écriture

```
#include <stdio.h>

#define NB_NOMBRES 4

int main()
{
    FILE* fd = fopen("foo.txt", "w");
    if (fd == NULL)
    {
        printf("Erreur à l'ouverture\n");
        return 1;
    }

    int nombres[NB_NOMBRES] = {35, 17, 4, 1};

    size_t len_written = fwrite(nombres, sizeof(int),
                                NB_NOMBRES, fd);
    if (len_written != NB_NOMBRES)
    {
        printf("Erreur à l'écriture\n");
    }

    fclose(fd);
    return 0;
}
```

Affichez le contenu du fichier `foo.txt`.
Que voyez-vous ? Pourquoi ?

Fonction d'écriture : `fprintf`

```
int fprintf(FILE* fd, char* format, ...);
```

- ▶ Comme `printf`, mais on ajoute comme premier paramètre le descripteur de fichier où écrire
- ▶ Renvoie le nombre de caractères écrits
- ▶ Écrit la chaîne de caractères formatée
- ▶ Déplace le curseur dans le fichier de ce qui a été écrit

Exemple

```
fprintf(  
    fd,  
    "%d %d %d %d\n",  
    nombres[0], nombres[1], nombres[2], nombres[3]  
);
```

Affichez le contenu du fichier. Que voyez-vous ? Pourquoi ?

Fonction de lecture : `fread`

```
size_t fread(  
    void* buffer,  
    size_t taille,  
    size_t n,  
    FILE* fd  
);
```

- ▶ Lit `taille`×`n` octets depuis `fd` vers l'adresse `buffer`
- ▶ Renvoie le nombre d'éléments vraiment lus (pas forcément le nombre d'octets, peut être différent de ce qui est demandé en cas d'erreur!)
- ▶ Déplace le curseur dans le fichier de ce qui a été lu

Exemple de lecture

```
#include <stdio.h>

#define NB_NOMBRES 4

int main()
{
    FILE* fd = fopen("foo.txt", "r");
    if (fd == NULL)
    {
        printf("Erreur à l'ouverture\n");
        return 1;
    }

    int nombres[NB_NOMBRES];

    size_t len_read = fread(nombres, sizeof(int), NB_NOMBRES, fd);
    if (len_read != NB_NOMBRES)
    {
        printf("Erreur à la lecture\n");
    }

    printf("%d %d %d %d\n", nombres[0], nombres[1], nombres[2],
           nombres[3]);

    fclose(fd);
    return 0;
}
```


Fonction de lecture : fscanf

```
int fscanf(FILE* fd, char* format, ...);
```

- ▶ Lit des chaînes de caractères depuis `fd` vers les zones mémoires dont les adresses sont passées en paramètre, selon le `format`
- ▶ `%s` s'arrête au premier espace rencontré
- ▶ Renvoie le nombre d'éléments trouvés
- ▶ Déplace le curseur dans le fichier de ce qui a été lu

Exemple

```
fscanf(  
    fd,  
    "%d %d %d %d\n",  
    &nombres[0], &nombres[1], &nombres[2], &nombres[3]  
);
```

Fonction de lecture : `fgets`

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

- ▶ Depuis `fd`, lit vers `s` au plus `size-1` caractères ou s'arrête après le premier `\n`
- ▶ Déplace le curseur dans le fichier de ce qui a été lu

Exemple

```
#define BUFFER_SIZE 64  
  
char buffer[BUFFER_SIZE];  
fgets(buffer, BUFFER_SIZE, fd);
```

Exercice : dictionnaire

Un fichier dictionnaire va contenir une entrée par ligne, sous le format :

```
mot définition avec plusieurs mots
```

La longueur d'un mot est limité à 20 caractères, celle de la définition à 100 caractères.

Programme `add_def`

Permet d'ajouter une entrée au dictionnaire :

```
./add_def t1 "Première année Télécom"
```

Programme `print_def`

Permet d'afficher une entrée du dictionnaire

```
./print_def t1
```

À vous de jouer !

Exercice : dictionnaire

Un fichier dictionnaire va contenir une entrée par ligne, sous le format :

```
mot définition avec plusieurs mots
```

La longueur d'un mot est limité à 20 caractères, celle de la définition à 100 caractères.

Programme `add_def`

Permet d'ajouter une entrée au dictionnaire :

```
./add_def t1 "Première année Télécom"
```

Programme `print_def`

Permet d'afficher une entrée du dictionnaire

```
./print_def t1
```

Bonus

Rendre le nom du fichier dictionnaire configurable à la compilation

Déplacer le curseur dans le fichier

```
int fseek(FILE* fd, long offset, int whence);
```

- ▶ `fd` : descripteur de fichier dont il faut déplacer le curseur
- ▶ `offset` : de combien d'octets déplacer le curseur par rapport à `whence` (peut être négatif)
- ▶ `whence` : origine de `offset`

<code>whence</code>	Signification
<code>SEEK_SET</code>	Début du fichier
<code>SEEK_CUR</code>	Position actuelle
<code>SEEK_END</code>	Fin du fichier

Descripteurs de fichiers toujours ouverts

Ces descripteurs sont automatiquement ouverts au début du programme et fermés à la fin.

- ▶ `stdin` : entrée standard (« *clavier* »)
- ▶ `stdout` : sortie standard (« *console* »)
- ▶ `stderr` : sortie d'erreur (aussi « *console* »)

`printf(...)` \Leftrightarrow `fprintf(stdout, ...)`

Redirections :

```
./programme < entree > sortie_std 2> sortie_err
```

Vous en voulez encore ?

Récupérer une saisie utilisateur

- ▶ Fonction `scanf`
- ▶ Comme `fscanf`, mais lit directement sur l'entrée standard
- ▶ `%s` ne lira jamais d'espace

`scanf(...)` \Leftrightarrow `fscanf(stdin, ...)`

```
#include <stdio.h>

#define STR_SIZE 12

int main()
{
    char prenom[STR_SIZE];
    int age;
    char ville[STR_SIZE];

    printf("Prénom âge ville ? ");
    scanf("%s %d %s", prenom, &age, ville);

    printf("%s, %d ans et habite à %s\n",
           prenom, age, ville
    );

    return 0;
}
```

- ▶ Pourquoi `&age`
et pas `&prenom`,
`&ville`?
- ▶ Que renvoie
`scanf`?

Le danger de `scanf`

Exécuter le programme précédent avec comme prénom
Jean-MichelTresLongPrenom et comme ville
TresLongNomDeVilleARalonge. Que se passe-t-il ? Pourquoi ?

Le danger de scanf

Utilisez plutôt `fgets` :

```
#include <stdio.h>

#define STR_SIZE 10

int main()
{
    char buffer[3*STR_SIZE];

    printf("Prénom âge ville ? ");
    fgets(buffer, 3*STR_SIZE, stdin);

    printf("%s\n", buffer);

    return 0;
}
```

Comment ensuite récupérer le prénom, l'âge et la ville dans des variables séparées ?

Formats dans printf

%d, %u, %f, ..., mais aussi :

```
float f = 1024.2048, g = -2.12, h = 40960000.2048;  
int a = -12, b = 13254;
```

```
// Notation scientifique avec 6 décimales :
```

```
printf("%e\n", f); // 1.024205e+03
```

```
printf("%e\n", h); // 4.096000e+07
```

```
// Affiche seulement deux décimales :
```

```
printf("%.2f\n", f); // 1024.20
```

```
// Préciser le nombre de caractères à utiliser
```

```
// (alignement à droite) :
```

```
printf("%5d\n", a); // -12
```

```
printf("%5d\n", b); // 13254
```

```
printf("%8.2f\n", f); // 1024.20
```

```
printf("%8.2f\n", g); // -2.12
```

```
// Préfixer par un caractère :
```

```
printf("%06d\n", b); // 013254
```

Pointeurs constants

Le terme situé après `const` est constant :

```
// l'adresse stockée dans ptr est constante :  
// ("ptr pointe vers un int")  
int* const ptr = &var1;  
ptr = &var2; // interdit  
  
// la valeur à l'adresse stockée dans ptr est const. :  
// ("ptr pointe vers un const int")  
int const* ptr = &var1;  
const int* ptr = &var1; // équivalent  
*ptr = 1; // interdit  
  
// possibilité d'utiliser les deux en même temps :  
const int* const ptr = &var1;  
*ptr = 1; // interdit  
ptr = &var2; // interdit
```

Beaucoup utilisé dans les types de paramètres de fonctions.

Pointeurs de fonctions

- ▶ Les instructions du programme sont chargées en mémoire → chaque fonction a une adresse.
- ▶ Possibilité de stocker cette adresse dans un *pointeur de fonction*.
- ▶ On peut ensuite appeler la fonction depuis le pointeur.
- ▶ Utile si on ne connaît pas à la compilation la fonction qu'il faudra exécuter (*callback*) ou pour factoriser le code.

Syntaxe

```
type_retour (*nom_ptr) (type_param1, type_param2);
```

Exemple

```
int (*compare)(const char*, const char*) = strcmp;  
compare(s1, s2); // Compare en respectant la casse  
compare = strcasecmp;  
compare(s1, s2); // Compare en ignorant la casse
```

Variable locale statique

Rappel : la durée de vie d'une variable est celle de son bloc d'instructions parent (condition, boucle, fonction, ...).

La durée de vie d'une **variable locale statique** est celle du programme :

```
void f(void)
{
    static int x = 0;
    printf("%d\n", x);
    x++;
}

int main()
{
    for (int i = 0; i < 3; i++)
    {
        f();
    }

    return 0;
}
```

Va afficher :

```
0
1
2
```

- ▶ Crée un *effet de bord*, la fonction n'est plus *pure*
- ▶ Quand même utile dans certains cas (patron de conception *singleton*, par exemple)

Variable globale

- ▶ Définition à l'extérieur des fonctions
- ▶ Portée dans tout le fichier source (et autres fichiers sources en utilisant `extern`)
- ▶ Durée de vie égale à la durée de vie du programme

```
int a = 5;

void f(void)
{
    a++;
}

int main()
{
    f();
    f();
    printf("a = %d\n", a);

    return 0;
}
```

Variable globale

- ▶ Définition à l'extérieur des fonctions
- ▶ Portée dans tout le fichier source (et autres fichiers sources en utilisant `extern`)
- ▶ Durée de vie égale à la durée de vie du programme

main.c :

```
#include "foo.h"

extern int a;

int main()
{
    f();
    f();
    printf("a = %d\n", a);

    return 0;
}
```

foo.c :

```
int a = 5;

void f(void)
{
    a++;
}
```

foo.h :

```
#ifndef __FOO_H
#define __FOO_H

void f(void);

#endif // !__FOO_H
```


Variable globale privée

Pour empêcher qu'une variable globale puisse être vue par un autre fichier :

```
static int a = 5;

void f(void)
{
    a++;
}
```

Fonction privée

Pour éviter qu'une fonction ne puisse être vue par un autre fichier :

```
static void f(void)
{
    a++;
}
```

- ▶ Évite les risques de conflits de nommage entre fonctions de fichiers différents
- ▶ Peut accélérer (un peu) la compilation

Résumé des visibilitées des symboles

Par défaut

- ▶ Variable locale visible et valide uniquement dans le bloc d'instructions courant
- ▶ Variable globale visible et valide dans tout le programme
- ▶ Fonction visible dans tout le programme

`static` sur une variable locale

- ▶ La variable garde sa valeur entre les appels

`static` sur une variable globale ou une fonction

- ▶ La variable ou la fonction n'est plus visible en-dehors du fichier

`extern` sur une variable globale ou une fonction

- ▶ Cette variable globale ou fonction existe quelque part

Fonction `inline`

Indique au compilateur de remplacer l'appel à la fonction par le contenu de la fonction.

- ▶ Utilisé surtout avec de petites fonctions
- ▶ Lorsque les performances sont importantes : économise un appel de fonction

La fonction doit être définie `static` ou bien directement dans un fichier d'en-tête. Pourquoi ?

```
static inline int max(int a, int b)
{
    return a > b ? a : b;
}
```

Gestion des erreurs

Erreurs

- ▶ Mauvaise utilisation de vos fonctions : paramètre invalide, ...
- ▶ Erreur venant de l'utilisateur : fichier inexistant, ...
- ▶ Erreur système : pas assez d'espace sur le disque, pas assez de mémoire, ...

Gestion des erreurs

- ▶ Pas de mécanisme dédié (pas d'exception ou autre)
- ▶ Valeur de retour des fonctions (en cas d'erreur : différent de zéro, négatif, ...) : `int foo(params...);`
- ▶ Pointeur passé en paramètre pour récupérer le statut :
`void foo(params..., int* status);`

Gestion des erreurs : `errno`

```
#include <errno.h>
```

- ▶ Variable globale `int errno` contenant le dernier code d'erreur rencontré
- ▶ Utilisé par les fonctions système : gestion de fichiers, de la mémoire, ...
- ▶ Fonction `void perror(const char *s)` affiche le dernier code d'erreur avec comme préfixe la chaîne de caractères `s`

Exemple

```
FILE* fd = fopen("inexistant.txt", "r");  
if (fd == NULL)  
{  
    perror("Erreur à l'ouverture");  
    return 1;  
}
```

RTFM : [man errno](#), [man perror](#) et les section ERREURS des pages de [man](#) des fonctions utilisées.

Exercice : utiliser `perror` dans `print_def.c`

Gérez les messages d'erreurs possibles dans `print_def.c` avec `perror()`.

À vous de jouer !

Fonction `assert()`

```
#include <assert.h>
```

`assert(expr)`; arrête le programme si `expr` est évalué à 0

- ▶ Aide au développement, programmation par contrat
- ▶ Permet de vous assurer que les conditions de validité de vos algorithmes sont respectées

Exemple

```
int algo(int param)
{
    // TODO: make it work with param < 0
    assert(param >= 0);

    // ...
}
```


Fonction `assert()`

```
#include <assert.h>
```

`assert(expr)` ; arrête le programme si `expr` est évalué à 0

- ▶ Aide au développement, programmation par contrat
- ▶ Permet de vous assurer que les conditions de validité de vos algorithmes sont respectées

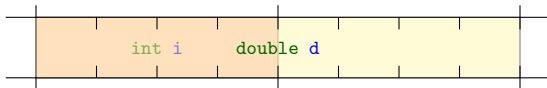
Les instructions `assert` peuvent être enlevées ! Il suffit de compiler avec `-DNDEBUG`

```
FILE* fd = fopen(file, "r");  
assert(fd); // Mal ! Pourquoi ?
```

Unions

- ▶ Comme une structure, mais seul un attribut peut-être utilisé en même temps
- ▶ Tous les membres se partagent le même espace mémoire
- ▶ La taille de l'union correspond à la taille de son plus grand membre

```
union union_u {  
    int i;  
    double d;  
};  
  
int main()  
{  
    union union_u u;  
    u.i = 2;  
    u.d = 3.14; // u.i est écrasé  
    printf("i: %d d: %lf\n", u.i, u.d);  
  
    return 0;  
}
```



Unions : exemple

Souvent utilisé conjointement avec une structure et une énumération

```
enum type_nombre_e {
    ENTIER,
    FLOTTANT
};

union nombre_u {
    int i;
    double d;
};

struct nombre_s {
    enum type_nombre_e type;
    union nombre_u valeur;
};
```

```
struct nombre_s nombre = {
    ENTIER, {.i = 12}
};

switch (nombre.type)
{
    case ENTIER:
        printf(
            "%d\n",
            nombre.valeur.i
        );
        break;
    case DOUBLE:
        printf(
            "%lu\n",
            nombre.valeur.d
        );
        break;
}
```

Types anonymes

Pour les `struct`, `enum` et `union`, il est possible de ne pas nommer les nouveaux types, utilisés alors comme des *sous-types* :

```
struct nombre_s {  
    enum {  
        ENTIER,  
        FLOTTANT  
    } type;  
    union {  
        int i;  
        double d;  
    } valeur;  
};
```

Tableaux multi-dimensionnels

- Stockage contigu en mémoire, *row-major* : dans un tableau `t[n][m]`, la case `t[i+1][0]` est stockée juste après la case `t[i][m-1]`
- Tableaux denses

```
int t[2][3] = { { 1, 2, 3 }, { 4, 5, 6 } };
int* li0;    /* Pointeur sur la première ligne */
int* li1;    /* Pointeur sur la deuxième ligne */

li1 = t[1];  /* t de type (int **), t[1] de type (int *) */
li0 = t[0];  /* Même valeur d'adresse, pas même type ! */
li1[1] = 7;  /* Écriture tout à fait correcte */
t[0][4] = 7; /* Pas de vérification de débordement ! */
```

Tableaux multi-dimensionnels manuels

- ▶ Un tableau unidimensionnel
- ▶ Linéarisation manuelle pour accéder à la bonne case

```
int t[6]; // utilisé comme t[2][3]
t[index(3, 1, 1)] = 12 // = t[1][1] = 12
```

```
int index(int width, int row, int col)
{
    return row * width + col;
}
```

Tableaux multi-dimensionnels avec allocation dynamique

- ▶ Le stockage n'est plus (nécessairement) contigu en mémoire
- ▶ Tableaux pas nécessairement denses

```
int** tab = malloc(n_lignes*sizeof(int*));
for (int i = 0; i < n_lignes; i++)
{
    tab[i] = malloc(n_cols*sizeof(int));
}

tab[2][3] = 45;

// On n'oublie pas *tous* les free
```

Juste pour que vous sachiez que ça existe...

typedef

- ▶ Permet de renommer un type
- ▶ Mauvais cas d'utilisation :
 - ▶ S'économiser l'écriture du mot-clef `struct`
- ▶ Cas acceptables d'utilisation :
 - ▶ Renommer le type d'un pointeur de fonction
 - ▶ Dissocier la définition d'un type de son implémentation

```
// le nouveau type est "print_function"
typedef void (*print_function)(void *);
void print_array(int len, void **array, print_function fun);
```

```
typedef int Age;

Age a = 27;
```

goto

- ▶ Permet de sauter à un autre endroit du code
 - ▶ Peut être vu comme une généralisation de `break` et `continue`
- ▶ Saut uniquement au sein de la même fonction
- ▶ **Ne devrait jamais être utilisé**
 - ▶ Fait du code spaghetti
 - ▶ Sauf pour éventuellement sortir d'un nid de boucles ou factoriser la gestion d'erreurs
 - ▶ Il est toujours possible d'éviter d'utiliser `goto`

```
int* t = malloc(...);
if (!t)
    goto exit;
// ...
if (some_error)
    goto error;
// ...
if (other_error)
    goto error;
// ...

error:
    free(t);

exit:
    return -1;
```

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < t[i]; j++)
    {
        for (int k = 0; k < t[i][j]; k++)
        {
            if (is_ok(t[i][j][k]))
            {
                goto foo;
            }
        }
    }
}

foo:
    // Keep working
```

volatile

- ▶ Qualificateur de type
- ▶ Indique au compilateur que le contenu de la variable peut-être modifié par un événement extérieur au programme
 - ▶ Par exemple : du matériel, un signal, ...
- ▶ Empêche le compilateur de faire certaines optimisations

```
volatile int ready = 0;
```

- ▶ À n'utiliser que si vous savez ce que vous faites
- ▶ Pour des modifications de mémoires entre programmes, *cf* IF210 Programmation système

restrict

- ▶ Qualifie un pointeur
- ▶ Indique au compilateur qu'aucun autre pointeur `restrict` dans le contexte courant ne pointe au même endroit
- ▶ Permet au compilateur de réaliser des optimisations
- ▶ **Attention** : le compilateur ne vérifiera pas un mauvais usage de `restrict`

```
void f(int n, int * restrict p, int * restrict q)
{
    while (n-- > 0)
        *p++ = *q++; // Copie le contenu de q dans p
}

void g(void)
{
    int d[100];
    // ...
    f(50, d + 50, d); // OK
    f(50, d + 1, d);  // Undefined behavior: d[1] est accédé via p
                     // et q dans f
}
```

Les différents standards C

- ▶ *Standard* : ce qui est permis ou non par le langage, quelle est la syntaxe, quelle est la sémantique des mots-clés, ...
- ▶ Précisé avec l'option `-std=` de GCC (la valeur par défaut dépend de la version de GCC)
- ▶ Les compilateurs peuvent ajouter leurs extensions
- ▶ Changer la version du compilateur peut nécessiter des changements dans votre code pour toujours compiler
- ▶ C99
 - ▶ Enseigné dans ce cours
 - ▶ Le plus utilisé en pratique
 - ▶ Ajoute la syntaxe pour les commentaires `// Commentaire`
 - ▶ Permet de déclarer des variables dans `for` :

```
for (int i = 0; i < 10; i++)
```
- ▶ C11
 - ▶ Programmation générique avec `_Generic`
 - ▶ Structures et unions anonymes
 - ▶ Support du multi-thread et de l'unicode
- ▶ C23
 - ▶ Beaucoup de petits ajouts (fonctions, mots-clés, ...)

Résumé : tous les mots-clefs du langage C

Liste des mots-clefs du langage C

<code>auto</code>	<code>do</code>	<code>goto</code>	<code>return</code>	<code>typedef</code>
<code>break</code>	<code>double</code>	<code>if</code>	<code>short</code>	<code>union</code>
<code>case</code>	<code>else</code>	<code>inline</code>	<code>signed</code>	<code>unsigned</code>
<code>char</code>	<code>enum</code>	<code>int</code>	<code>sizeof</code>	<code>void</code>
<code>const</code>	<code>extern</code>	<code>long</code>	<code>static</code>	<code>volatile</code>
<code>continue</code>	<code>float</code>	<code>register</code>	<code>struct</code>	<code>while</code>
<code>default</code>	<code>for</code>	<code>restrict</code>	<code>switch</code>	

- ▶ Hormis `auto` et `register`, vous les connaissez tous
 - ▶ `auto` : classe de stockage sur la pile, par défaut (donc toujours omise) lorsqu'une variable n'est pas `static` ou `extern`
 - ▶ `register` : indique au compilateur de stocker la variable dans un *registre* du processeur ; inutile de nos jours, car les compilateurs savent mieux que les développeurs où stocker les variables

Conclusion

Conclusion

Vous connaissez maintenant (presque) toute la syntaxe du langage C et (presque) tous ses mécanismes.

Il n'y a plus qu'à mettre en pratique !

- ▶ 6 séances de TP
- ▶ Dans d'autres cours :
 - ▶ S6 : PG110 | TE6-PROG1 Projet de programmation
 - ▶ S7 : RE216 | TE7-RESA1 Programmation réseau
 - ▶ S7 : IF210 | TE7-IFON1 Programmation système
 - ▶ ...

Quelques conseils pour progresser :

- ▶ Programmez ! Internet regorge d'exercices/concours de programmation
- ▶ Lisez du code C (le code source de tous les logiciels libres est accessible...)
- ▶ Lisez les messages d'erreur du compilateur
- ▶ RTFM : pages de [man](#), votre moteur de recherche favori, StackOverflow, ...

Pour satisfaire votre curiosité...

- ▶ Si vous voulez vous faire peur :
<https://jorenar.com/blog/less-known-c>