Institut
Mines-Télécom

Introduction
Langage C et
Systèmes d'exploitation
Partie 1: Langage C

Responsable : Florian Brandner (4D59)

Auteurs : Bertrand Dupouy et Etienne Borde

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs(1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie
- Deuxième partie
  - Préprocesseur
  - Tableaux et pointeurs(2)
  - Variables : allocation en mémoire
  - Fonctions
    - Pile et arguments
  - Fonctions d'E/S
- Annexe : Java/C

Plan

- Première partie
  - ☞ **Introduction**
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs (1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

INF104 - Langage C

- L'objectif de cette première partie est de donner les éléments de base pour débuter la programmation en C :
  - On verra ici comment se servir du langage,
  - Son fonctionnement sera détaillé dans la seconde partie,
  - Sa compilation (traduction du langage de haut niveau vers le langage machine) sera présentée dans la troisième partie.

- Organisation du cours :
  - 2 TH de cours (partie 1),
  - 3 TH de TP (un minimum de pratique),
  - 2 TH de cours (partie 2),
  - 2 TH de TP (quelques exercices plus avancés)

Langage C : un peu d'historique

- Le langage C a été créé par B. Kernighan et D. Ritchie dans les années 1970 et formalisé en 1978 (*The C Programming Language, 2nd ed., by Kernighan and Ritchie, Prentice Hall*). A cette époque, le système UNIX a été écrit en C. Les créateurs du langage le caractérisent ainsi :
  - Économie d'expression,
  - Absence de restrictions,
  - Ni de très haut niveau, ni très riche,
    - Objectif : efficace et adapté à la programmation système (par exemple : systèmes embarqués)

- Si le langage remplit bien le rôle qui lui a été assigné, une des conséquences de ces hypothèses d'extrême simplicité est la suivante :
  - Pas de contrôle à l'exécution (exemples : pas de gestion de la mémoire, pas de gestion d'exceptions), le comportement des applications va dépendre du système sous-jacent.

Langage C : caractéristiques générales

- Peu de constructions (ou mots-clés) à connaître pour l'utiliser,
- Langage très concis, les contraintes syntaxiques sont faibles et peuvent être levées (*le cast operator*), un programme est donc **vite écrit**,
- De plus, le compilateur est « permissif » (avec gcc : il faut utiliser l'option –Wall pour afficher tous les warning de compilation) on obtient donc facilement un exécutable, mais:
  1. la plus grande partie de la mise au point (*i.e.* résolution de bugs) se fera durant l'exécution (pas de contrôle sur la taille des tableaux, typage par défaut des fonctions et de leur arguments, …)
  2. Le programmeur doit savoir comment fonctionne le langage pour l'utiliser correctement,
     - **L'aspect simple du C masque le fait qu'il faut plus de connaissances sur le système qu'en Java pour l'utiliser**

Structure d'un programme C : un jeu de fonctions et de variables « à plat »

- La structure d'un programme C est très simple :
  - une (et une seule) fonction main, point d'entrée du programme: première fonction qui s'exécute lorsqu'on lance le programme
  - généralement, plusieurs autres fonctions
  - la définition et l'initialisation de variables
- L'ensemble de ces fonctions et variables peuvent être réparties dans un ou plusieurs fichiers
- Les fonctions sont déclarées et implémentées sans structuration: les fonctions sont visibles (peuvent être appelées) par n'importe quelle fonction. On écrit ces fonctions directement dans un fichier .c
- Les variables sont déclarées et utilisées avec une structuration simple:
  - des variables globales, visibles (peuvent être utilisées) par n'importe quelle fonction. Ces variables sont déclarées et initialisées directement dans un fichier .c
  - Des variables locales, déclarées et initialisées dans une fonction; ces variables sont visibles par la fonction seulement.
- Les commentaires sont identifiés par
  - // un commentaire sur une ligne
  - /* un commentaire qui peut s'étendre sur plusieurs lignes */

Plan

- Première partie
  - Introduction
  - ☞ **Variables**
    - Types de base
    - Types composés
    - Tableaux et pointeurs (1)
    - Opérateur sizeof
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

Variables : déclaration et définition

- La déclaration d'une variable suit le schéma suivant:

  extern <type> <identifiant>;

- La définition d'une variable suit le schéma suivant:

  <type> <identifiant> = <valeur_initiale>;

La différence entre définition et déclaration vient du fait qu'on peut déclarer une variable (avec le même identifiant) plusieurs fois. Par contre, une variable ne peut être définie qu'une seule fois.

Variables : type, allocation mémoire, durée de vie

- Le type d'une variable définit :
  - l'encombrement mémoire de la variable (sa taille, mesurée en octets),
  - les règles d'utilisation de cette variable (opérateurs associés),
  - la zone mémoire où est placée la variable (on verra cela dans la deuxième partie du cours),

- L'emplacement de la déclaration d'une variable détermine :
  - sa durée de vie, qui peut être celle :
    - du programme (variable globale),
    - de la fonction (variable locale à la fonction)
  - sa visibilité (variables globales ou locales, à une fonction, à un fichier,…), on en a déjà parlé dans la description de la structuration d'un programme.

Plan

- Première partie
  - Introduction
  - Variables
    - ☞ **Types de base**
    - Types composés
    - Tableaux et pointeurs (1)
    - Opérateur sizeof
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

Types de base: les entiers

- Plusieurs types d'entiers : **`char`** (1 octet), **`short`** (2 octets), **`int`** et **`long`**
  - Seul la taille des types **`char`** et **`short`** est fixée: respectivement 1 et 2 octets
  - La taille des autres types peut varier selon les machines… Sur ma machine (64 bits) : `int` -> 4 octets, `long` -> 8 octets.

- L'ensemble de valeurs que l'on peut encoder avec un type particulier dépend du type en question: 2^{Nb_bits}.

- Le qualificatif **`unsigned`** peut être utilisé pour indiquer qu'on n'utilise pas le bit de signe, ce qui double la dynamique des positifs:
  - **`char`** c; // valeurs allant de -128 à 127.
  - **`unsigned char`** c; // valeurs allant de 0 à 255.

Le type `char` pour les caractères

- Le type **`char`** peut aussi être utilisé pour représenter des caractères ASCII.
- Les valeurs de caractères sont identifiées via des guillemets simples: ‘a’, ‘A’, ‘b’, ‘C’…
- Des caractères spéciaux souvent utilisés :
  - ‘\n’: retour à la ligne; ‘\t’: marge; ‘\0’: caractère NULL
- Les **`char`** **sont traités comme des entiers sur un octet**, attention aux effets de bord (voir exemple qui suit)

Le type `char` : premier exemple de programme

Fichier: premier_program.c

```
#include <stdio.h>
char carac1;

int main(int argc, char*argv[]){
 unsigned char carac2;
 carac1 = 255;
 carac2 = 255;
 printf("carac1 = %d carac2 = %d\n", carac1, carac2);
 carac1 = 'A';
 carac2 = '1';
 printf("carac1 = %c carac2 = %c\n", carac1, carac2);
 printf("carac1 = %d carac2 = %d\n", carac1, carac2);
 /* Les char sont traitées comme des int !!!! */
 carac1 = carac1 + 1;
 printf("carac1 = %c (%d)\n", carac1, carac1);
 return 0;
}
```

`%d` : format décimal

`%c` : format "ASCII"

Résultats (affichés sur le terminal):

```
carac1 = -1 carac2 = 255

carac1 = A carac2 = 1
carac1 = 65 carac2 = 49

carac1 = B (66)
```

- Effets de bord sur l'interprétation des données (255→1; A→65;A+1→66),
- Faible structuration, fonctions et variables globales au même niveau,
- Nous expliquerons plus tard le fonctionnement de la fonction printf (), ainsi que le rôle du **#include**
- La fonction main a toujours cette forme, que nous expliquerons par la suite,
- Variable globale: carac1, de type char, utilisable dans toute fonction,
- Variable locale: carac2, utilisable dans la fonction main seulement.

Le type booléen n'existe pas

- Pour les booléens, on utilise en C le type char avec la convention suivante:
  - la valeur 0 représente la valeur booléenne *false.*
  - Toute valeur différente de 0 représente la valeur booléenne *true.*

```
char c = 0;
if(c) // toujours faux
{}
c = 25;
if(c) // toujours vrai
{}
```

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - ☞ **Types composés**
    - Tableaux et pointeurs (1)
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

Types composés : « `struct` »

- On peut composer les types de base pour créer des types plus élaborés, appelés `struct`.
- La déclaration ci-contre crée le type « `struct complexe` ».
- Pour accéder à un élément d'une composition « `struct` » on utilise le point (.),
- L'opérateur = est le seul qui s'applique aux `struct`,
- Une structure peut être initialisée au moment de sa définition (initialisation statique):

```
struct complexe {
  float reel;
  float imagine;
};


int main(int argc, char*argv[]){
  struct complexe c1;
  struct complexe c2;

  c1.reel = 2.0;
  c1.imagine = 3.0;

  c2 = c1;

  return 0;
}
```

```
struct complexe {
  float reel;
  float imagine;
};


struct complexe c1 = {2.0,3.0};
```

Définition de types : utilisation de `typedef`

- L'instruction `typedef` permet d'associer un raccourci à un nom de type, la syntaxe est la suivante :

```
typedef <type> <nom_nvx_type> ; //définition du type
<nom_nvx_type> <nom_variable> ; /*déclaration d'une variable
                                  avec le nouveau type*/
```

Exemple :
```
typedef unsigned int uint;
uint un_entier_positif;
```

Définition de types : `struct` et `typedef`

- Si on reprend l 'exemple de la structure « complexe », on pourrait définir un nouvel identifiant de type `complexe_t` :

```
typedef struct complexe complexe_t;
```

- On aurait pu le définir directement, sans utiliser l'intermédiaire `complexe` :

```
typedef struct {
    float reel;
    float imagine;
} complexe_t;
```

- Dans tous les cas, on peut alors déclarer les variables c1 et c2 comme suit :

```
complexe_t c1, c2 ;
```

```
struct complexe {
  float reel;
  float imagine;
};

typedef struct complexe complexe_t;


int main(int argc, char*argv[]){
  complexe_t c1;
  complexe_t c2;

  c1.reel = 2.0;
  c1.imagine = 3.0;

  c2 = c1;

  return 0;
}
```

le type enum

- On peut définir un type comme une énumération de valeurs constantes possibles.
- Illustrons cela avec les booléens. Attention, par défaut la première valeur (false dans notre exemple) est considérée comme un entier constant égal à 0. Si on écrit

```
enum bool {
  true,
  false

};
```

on risque d'avoir des soucis…

```
enum bool {
  false,
  true
};


int main(int argc, char*argv[]){
  enum bool b = false;
  return 0;
}
```

- Pour lever l'ambiguité, écrire:

```
enum bool {
  true = 1,
  false = 0
};
```

```
enum bool {
  true = 1,
  false = 0
};

typedef enum bool bool_t;

int main(int argc, char*argv[]){
  bool_t b = false;
  return 0;
}
```

- `typedef` peut évidemment être utilisé ici…
- Un même identifiant ne peut être utilisé plusieurs fois dans le programme: le code ci contre est incorrect car ***blue*** est utilisé deux fois

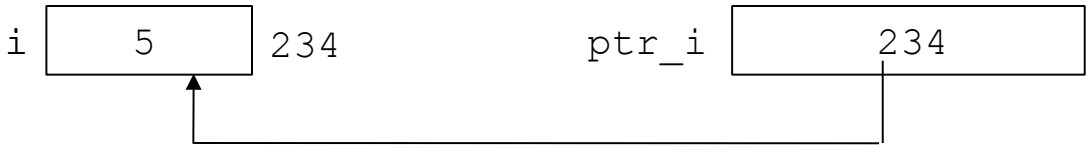
```
enum color1 {blue,green};
enum color2 {blue,yellow};
```

Plan

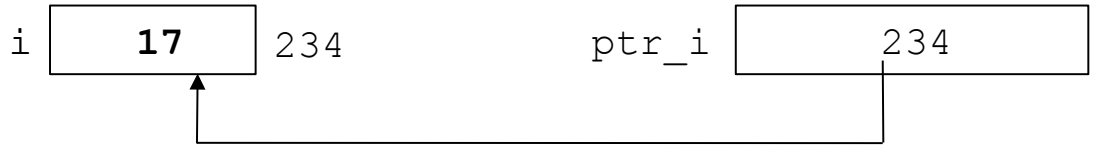
- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - ☞ **Tableaux et pointeurs (1)**
  - Fonctions
  - Instructions
  - Fonctions d'entrée/sortie

Pointeurs : définition

- Un pointeur est une variable contenant l'adresse (la référence) d'une autre variable.
- Soient les variables `i` et `ptr_i`. On veut que `ptr_i` pointe vers `i` (i.e. que `ptr_i` contienne l'adresse de `i`).
  - Supposons que `i` est implantée à l'adresse `234` et contient la valeur `5`.
  - On initialise alors `ptr_i` avec l'adresse de `i`: dans ce cas, `ptr_i` pointe vers `i`.



- Accéder au contenu pointé (i.e. à l'information qui se trouve à cette adresse) s'appelle un déréférencement ;
  - en C , cette opération est notée *
  - Exemple : `*ptr_i = 17 ;` a l'effet suivant:



Pointeurs : syntaxe

- Une variable de type pointeur se déclare avec le type de l'objet pointé suivi d'un astérisque (*):

```
<type_objet_pointé> * ptr;
```

- Il existe une valeur constante spéciale pour les pointeurs: `NULL` indique que le pointeur n'a pas été initialisé.
- Le programme ci-contre illustre comment sont déclarés et initialisés deux pointeurs, l'un sur des réels, l'autre sur des entiers.
- L'opérateur * déréférence le pointeur (permet d'accéder au contenu de l'objet pointée), cet opérateur ne doit jamais être utilisé sur un pointeur non-initialisé (`NULL`)
- L'opérateur & renvoie l'adresse (la référence) d'une variable
- Remarques :
  - Si on ne connait pas le type pointé, on peut utiliser le mot clé void :

```
void * ptr3;
```

  - Le type d'un pointeur va jouer sur son arithmétique (celle-ci sera vue en partie 2)

```
int i = 7;
float r = 3.14;

/* declarations */
int * ptr1 ;
float * ptr2;

/* initialisations */
ptr1 = &i;
ptr2 = &r;
```

Pointeurs : exemple

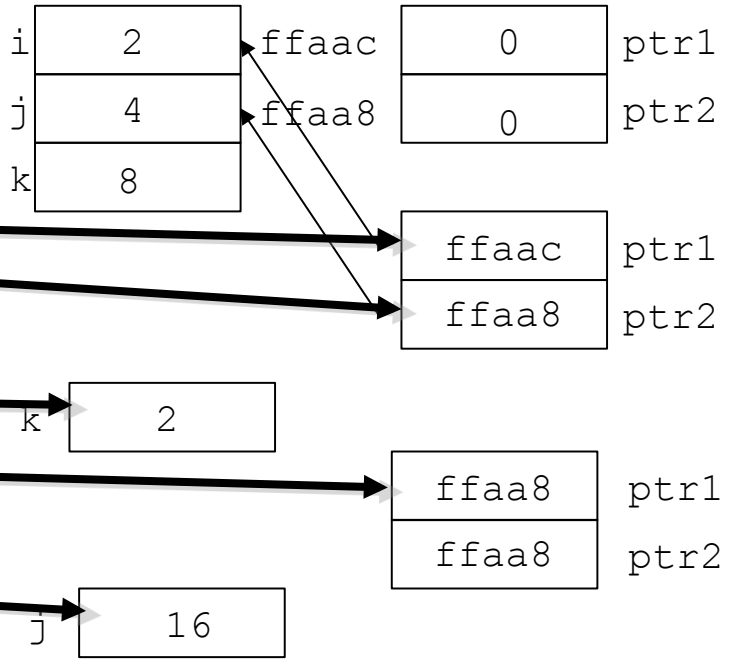
```
int i = 2, j = 4, k = 8;
/* déclarer des pointeurs sur des int  */
int * ptr1, * ptr2;

/* initialisation des pointeurs*/
ptr1 = &i;
ptr2 = &j;

/* k reçoit la valeur de l entier pointé
par ptr1 */
k = *ptr1;

ptr1 = ptr2;

/* j reçoit la valeur 16 */
*ptr1 = 16;
```



Tableaux

- Déclarations d'un tableau de 20 entiers et d'un tableau de 8 réels :

```
int tab1[20];      /* implanté de tab1[0] à
tab1[19] */

float tab2[8];    /* implanté de tab2[0] à tab2[7]
*/
```

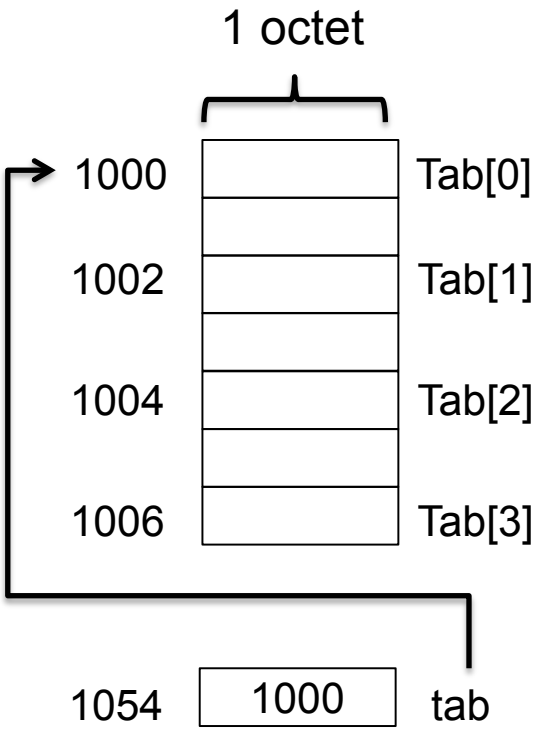
- En C, la taille des tableaux doit être connue à la compilation (sauf si on utilise la version C99), donc définie via une valeur numérique constante

- Il n'y a pas à proprement parler de type tableau en C : le nom d'un tableau est un pointeur constant sur l'adresse de début du tableau.

Exemple, soit la déclaration d'un tableau de 4 mots de 2 octets chacun :

```
short tab[4] = {0,-300,500,340};
```

Si on suppose que le tableau est implanté à l'adresse `1000`, la case dont le nom est `tab` contient la valeur `1000`.



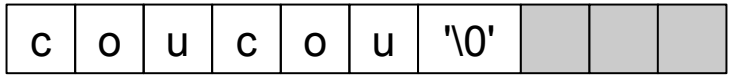
Chaînes de caractères : tableau de char

- C donne un support minimal pour le traitement des chaînes de caractères, le type `char` **ne correspond pas** à celui de Java.
  - Une chaîne de caractères est un simple tableau **d'octets (correspondant à un tableau de** `byte` **en Java**) auquel est ajoutée une propriété :
    - Le caractère « `NULL` » (noté '\0') indique la fin de chaîne,
  - Ceci a de nombreuses conséquences, en particulier :
    - On ne peut pas copier une chaîne dans une autre par affectation (=), il faut utiliser la fonction `strcpy` qui copie une chaîne y compris le «`NULL`».

      Il n'y a pas de vérification sur la taille de la chaîne destinataire.
- Exemple, le programme suivant écrit **7** octets dans le tableau `texte` :

```
char texte[10];
...
strcpy(texte, "coucou");
```

Contenu de `texte` après exécution de `strcpy`:



Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs (1)
  - ☞ **Fonctions**
  - Instructions
  - Fonctions d'entrée/sortie

Fonctions : déclaration et implémentation

- La déclaration d'une fonction suit le schéma suivant (appelée signature ou prototype):

```
<type_de_retour> <nom_de_la_fonction>(<liste_des_arguments>);
```

- L'implémentation d'une fonction suit le schéma suivant:

```
<type_de_retour> <nom_de_la_fonction>(<liste_des_arguments>)
{
    // déclaration de variables locales
    // instructions
    return <valeur_correspondant_au_type>;
}
```

- Utilisation:

```
    <type_de_retour> res = <nom_de_la_fonction>(<arguments>);
```

Note : La différence entre déclaration et implémentation vient du fait qu'il faut toujours déclarer une fonction avant de l'utiliser. Illustration au slide suivant

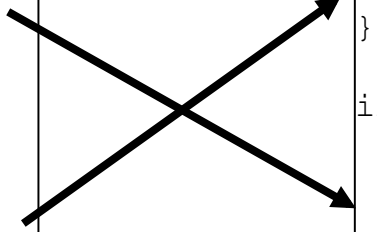
Note2 : la valeur de retour est optionnelle : si elle ne renvoie rien (c'est une procédure) elle est déclarée de avec le type de retour **`void`**, et on peut utiliser simplement **`return.`**
Voir les exemples fournis plus loin.

Pourquoi déclarer avant d'utiliser?

- Soit l'exemple :

```
int main (int argc, char * argv[]){
   int i = 0;
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}

float carre(float var){
    return (var * var);
}
```



```
float carre(float var){
    return (var * var);
}

int main (int argc, char * argv[]){
   int i = 0;
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}
```

Résultat : gcc crée un exécutable, mais l'exécution du programme peut donner ce résultat (le résultat va dépendre des machines et/ou compilateurs):

```
i = 0
```

Résultat : gcc crée un exécutable, et l'exécution du programme donnera ce résultat

```
i = 16
```

Mais doit-on ordonner toutes les fonctions du programme?

- Non! Il suffit de déclarer les fonctions utilisées au début du fichier source. En reprenant l'exemple précédent :

Déclaration ajoutée →

```
float carre(float var);

int main (argc, argv){
   int i = 0;
   i = carre (4);
   printf ("i = %d\n", i);
   return 0 ;
}

float carre(float var){
    return (var * var);
}
```

Résultat : gcc crée un exécutable, et l'exécution du programme donnera ce résultat

```
i = 16
```

Les fonctions : exemple

- Dans le programme ci-contre, `main` appelle deux fonctions qui prennent chacune un seul argument :
  - `carre` qui attend un argument de type `int` et qui renvoie un `int`,
  - `affich_car` qui attend un argument de type `char` et qui ne renvoie rien.

```
#include <stdio.h>

int carre(int);
void affich_car(char);

int main(int argc, char*argv[]){
 int k;
 char carac = '3';

 k = carre(3);
 printf("k = %d\n", k);
 affich_car(carac);
 return 0;
}
/**********  carre  ********************/
int carre(int j){
 return (j * j);
}
/*********** affich_car ****************/
void affich_car(char le_carac){
 printf("Caract. %c : code ASCII(hexa)-> %x\n »,
 le_carac, le_carac);
}
```

```
Résultat :
k = 9
Caract. 3 : code ASCII(hexa)-> 33
```

Notes sur l'opérateur `return`

- `return` interrompt l'exécution de la fonction en cours d'exécution (et donc du programme lorsque return est invoqué dans la fonction `main`)

exemples

```
void affiche_texte(char * texte);
char est_carre(int longueur, int largeur);

char est_carre(int longueur, int largeur){
  if(longueur==largeur)
    return 1; // interrompt l'execution de la fonction est_carre
  return 0;
}

int main (int argc, char * argv[]){
   char est_carre = est_carre(20,35);
   if(est_carre == false)
     return -1; // interrompt l'execution de la fonction main, et donc du programme.
   affiche_texte("on a un carre\n");
   return 0 ;
}

void affiche_texte(char * texte)
{
  if(texte == NULL)
    return; // pas de valeur retourné, mais fonction interrompu
  // suite de l'implémentation…
}
```

Quelques mots sur la fonction `main`

- Rappel: la fonction main est le point d'entrée du programme, tout programme doit donc avoir une et une seule fonction main.
- L'implémentation de la fonction main suit le schéma suivant

```
int main(int argc, char* argv[])
{
  // début du code du programme
  // …
  // fin du code du programme
}
```

- Les arguments:
  - argc : nombre d'arguments passés sur la ligne de commande au lancement du programme.
  - argv: tableau de chaînes de caractères contenant la liste des arguments passés sur la ligne de commande (séparés par un espace sur la ligne de commande).

  Exemple: soit un programme `mon_prog`, que l'on exécute ainsi: `./mon_prog 3 4` Alors le paramètre `argc` vaut `3`, et `argv` vaut { "./mon_prog", "3", "4"} lorsque la fonction `main` de `mon_prog` commence à s'exécuter.
- La valeur de retour sert à savoir dans quel état a fini le programme (par exemple en renvoyant un code de retour d'erreur prédéfini) ce qui est utile lorsque le programme est lancé par un autre programme (on verra tout ça dans la partie du cours sur les systèmes d'exploitation).

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs (1)
    - Opérateur sizeof
  - Fonctions
  - ☞ **Instructions**
  - Fonctions d'entrée/sortie

Expression vs instruction

- Une expression utilise des opérateurs appliqués à des variables ou constantes et **renvoie une valeur**, exemples :
  - `j*2  /* renvoie 2xj, pas d'affectation */`
  - `i=j<<1  /* valeur de j décalée d'un bit vers la gauche, puis rangée dans i (i=2xj), renvoie cette valeur */`
  - `i<j  /* si vrai l'expression renvoie 1 sinon 0 */`
- Une instruction peut se réduire à une expression suivie d'un point virgule, donc les instructions **suivantes ne provoquent pas d'erreur, au plus des avertissements, un fichier exécutable sera produit** :
  - `k+5;  /* suite à une erreur de frappe on a saisi k+5; au lieu de i=k+5; */`
  - `j>2;`
  - `;`

Instructions : syntaxe

- Les instructions :
  - sont terminées par un point virgule (`;`),
  - sont comprises dans un bloc d'instructions c'est à dire une liste d'instructions encadrées par des accolades (`{ }`) ,
  - peuvent se réduire à une expression suivie d'un point virgule,
  - à la différence des expressions qui renvoient une valeur, elles provoquent une action,
    - Remarque : une expression peut modifier une case mémoire si elle fait une affectation (opérateurs `=`, `++`, par exemple),
- Exemples :
  - `k=j*2;`
  - `i=j<<1;`
  - `if(i<j)k=2;`

Instructions : conditionnelles, itératives

- Instructions conditionnelles :
  - `if ( expression ) instruction(s)`
  - `if ( expression ) instruction(s) else instruction(s)`

- Instructions d'itération :
  - `for(expression1; expression2; expression3) instruction(s)`

    Avec : `expression1`: initialisations, `expression2`: conditions d'arrêt, `expression3`: incréments,

    Exemple : `for(i=0; i<10; i=i+1) tab[i] = 0 ;`
  - Instruction `while` et `do while` :

    1. `while (expression) instruction(s)`

       l'expression est évaluée, si sa valeur est nulle, on sort du `while`,

    2. `do instruction(s) while (expression)`

       l'expression est évaluée, si sa valeur est nulle, on sort du `while`, les instructions sont exécutées au moins une fois,

Instructions : rupture de séquence

- Instructions de rupture de séquence :
  - `break` : fait sortir du **bloc** d'instructions de la boucle (`for`, `while` , `do while`) ou du `switch` dans lequel il est inclus,
  - `continue` fait passer à **l'itération suivante** du `for`, `while` , `do while` dans lequel il est inclus ,
  - `goto` (!!)
- Exemples :

```
while (<expression>) {
  <instructions>
  if (<condition>) break;
  <instructions>
}

/* le break fait sortir ici */
```

```
while (<expression>) {
  <instructions>
  if (<condition>) continue;
  <instructions>
 /* le continue conduit ici */

}
```

Instructions : `switch`

- Fonctionnement : l'expression `expression-switch` est évaluée et donne une valeur V,
  - le flot de contrôle passe au `case` dont la valeur est égale à V, ou à `default` si aucune ne correspond. On continue dans le switch à **partir de cette entrée.**
- Les sources d'erreurs :
  - `default` est optionnel,
  - `break` est optionnel, sans `break` on passe au `case` suivant,
- cette instruction doit être vue comme une table à plusieurs points d'entrée : on entre dans la table et on exécute tout ce qui suit

```
switch  (expression-switch) {
  case expression-constante1 :
    instruction
    break;
  case expression-constante2 :
     instruction
  case expression-constante3 :
  case expression-constante4 :
      instruction
      break;
  default :
       instruction
}
```

Instruction `switch` : exemple

- Exemple :

```
Résultat :


var vaut 0, entree default

var vaut 1, entree 1

var vaut 2, entree 2
var vaut 2, entree 3
var vaut 2, entree 4

var vaut 3, entree 3
var vaut 3, entree 4

var vaut 4, entree 4

var vaut 5, entree default
```

```
#include <stdio.h>
int main(int argc, char*argv[]) {
int var = 0 ;
  for (var = 0; var < 6; var++) {
    switch (var) {
      case 1 :
         printf("var vaut %d, entree 1\n", var);
           break;
      case 2 :
         printf("var vaut %d, entree 2\n", var);
       case 3 :
         printf("var vaut %d, entree 3\n", var);
      case 4 :
         printf("var vaut %d, entree 4\n", var);
         break;
       /* option default facultative */
      default:
          printf("var vaut %d, entree default\n", var);
     }
   }
return 0;
}
```

Opérateurs : operateur d'affectation =

- Opérateur =
  - à gauche de cet opérateur, il faut mettre une variable modifiable, ce qu'on appelle une `lvalue`,
    - Remarque : une `lvalue` peut être le résultat d'une expression,
- Cet opérateur copie la valeur retournée par la partie droite dans la `lvalue`,
  - Si la valeur à déposer n'est pas du même type que la `lvalue`, il y a conversion de type automatique,
  - Remarque : en tant que membre d'une expression, cet opérateur renvoie la valeur déposée dans la variable,
- Attention, c'est pour cette dernière raison que :

`if(i == 0)` et `if(i = 0)` sont syntaxiquement correctes…

… et sources de multiples erreurs !

- si on utilise des constantes, écrire if (4 == x) au lieu de if (x == 4)

Opérateurs : autres operateurs d'affectation

- Les opérateurs unaires :
  - `++` , -- en post ou pré incrément/décrément sont des opérateurs d'affectation :
    - `i++` signifie `i=i+1`, ne pas confondre :
      - `j = i+1` : mettre **la valeur de i** incrémentée de un dans j, **seule** la variable j est modifiée,
      - `j = i++` : ranger la valeur de i dans j **; puis incrémenter i de un**,les **deux** variables i et j sont modifiées,
      - Exemple, soit : `j = 0 ; i = 2 ;`
        - Après `j = i++ ;` (post-incrément) :
          `i = 3, j = 2`
        - Après j = ++i ; (pré-incrément) :
          `i = 3, j = 3`

Opérateurs : operateurs de comparaison

- Ces opérateurs (`<`, `>`, `==`) travaillent sur des opérandes entiers ou réels et renvoient les entiers 1 (vrai) ou 0 (faux),

- Il n'y a pas de type booléen (même si la norme C99 introduit _bool),

- Attention :
  - La syntaxe du `if` est : `if(` `expression`). Lors de son exécution :
    1. L'expression est évaluée, si sa valeur est 0, elle est considérée fausse,
    2. Sinon elle est considérée comme vraie (tout ce qui n'est pas faux est vrai),

       Conséquence, en C : `if( i=-1 )` est toujours vrai

Opérateurs : operateurs « bit à bit »

- Ces opérateurs sont plutôt utilisés sur des variables du type `unsigned`
- Opérateurs de décalage : `>>` (à droite), `<<` (à gauche)
  - Exemple :

    `j=j<<2 ;` (revient à multiplier j par 4)

- Opérateurs bit à bit : `~` (NOT), `&` (AND), `|` (OR), `^` (XOR)
- Ne pas les confondre avec les opérateurs logiques `&&` et `||` qui renvoient 1 ou 0
  - Exemple (`0x` précédant une constante veut dire base 16) :

    `j=j&0xff ;` (pour garder l'octet de poids faible de la variable j)

Représentation binaire de `0xff` (sur 32 bits) `0000 0000 0000 0000 0000 0000 1111 1111`

Représentation binaire de `j&0xff` (sur 32 bits) `0000 0000 0000 0000 0000 0000 xxxx xxxx`

Opérateurs : opérateurs logiques

- Les opérateurs logiques sont notés : ! (NON), &&(ET), ||(OU) ,
  - Ils traitent les variables comme des entiers en appliquant la règle : 0 est faux, tout le reste est vrai,
  - Ils renvoient 1 si le résultat est vrai, 0 sinon,
  - ils acceptent n'importe quel opérande numérique,
  - attention : l'évaluation se fait de gauche à droite, et seulement si nécessaire :
    - Dans l'expression : `if ((i++>j) && (k++>l))` , la sous-expression `(k++>l)` n'est pas évaluée si `(i++>j)` est fausse,

Opérateurs : le « cast operator »

- L'opérateur «`cast`», sert à forcer les conversions de type (permissivité du C), sa syntaxe est la suivante :

```
(type voulu)expression
```

- Exemple d'utilisation de l'opérateur `cast`:

```
int i = 20;
int j = 7;
float r;
 •••
 r = i/j ;                  resultat  r = 2.000000
 r = (float)(i/j);               resultat  r = 2.000000
 r = (float)(i)/j;               resultat  r = 2.857143
 r = 20/7;                  resultat  r = 2.000000
 r = 20/7.0;                resultat  r = 2.857143
```

Opérateurs : les règles de conversion implicite

- L'opérande de type le plus faible est converti dans le type de l'opérande le plus fort. Le résultat est du type de l'opérande le plus fort,
- Le tableau ci-dessous indique comment se font les conversions de type dans une expression arithmétique

	`char, int short`	`unsigned`	`long`	`float double`
`char,int short`	`int`	`unsigned`	`long`	`double`
`unsigned`	`unsigned`	`unsigned`	`long`	`double`
`long`	`long`	`long`	`long`	`double`
`float double`	`double`	`double`	`double`	`double`

- En cas d'une opération arithmétique entre une opérande dont le type est sur une ligne i et une opérande dont le type est sur une colonne j, le type du résultat est donné à la case (i,j)

Plan

- Première partie
  - Introduction
  - Variables
    - Types de base
    - Types composés
    - Tableaux et pointeurs (1)
    - Opérateur sizeof
  - Fonctions
  - Instructions
  - ☞ **Fonctions d'entrée/sortie**

Fonctions d'entrée/sortie

- Vous allez utiliser en TP des fonctions d'entrée/sortie qui permettent d'avoir des interactions entre:
  - le programme et l'utilisateur, via le terminal (printf/scanf)
  - le programme et des fichiers sur le disque (fprintf/fscanf)
  - le programme et des chaînes de caractères en mémoire (sprintf/scanf)

- Dans cette famille de fonction,
  - celles contenant l'identifiant printf permettent de produire (afficher, écrire) des chaînes de caractères;
  - celles contenant l'identifiant scanf permettent de récupérer des données à partir de chaînes de caractères (dans un fichier, via le terminal, ou dans une chaîne de caractère en mémoire).

Fonction d'entrée/sortie et chaînes de caractères formatées.

- Les fonctions d'entrée sortie s'appuient sur la notion de chaîne de caractères formatée: une chaîne de caractères à trous, dont les trous sont positionnés par l'identifiant de format. Ces formats identifient le type de valeur qui viendra remplacer le texte manquant.

  Exemple:
  En français: Ceci est un __ , il manquait __ mot(s) dans cette phrase
  En C: "Ceci est un %s, il manquait %d mot(s) dans cette phrase"

- Les formats ont une syntaxe du type %FORMAT où FORMAT permet d'identifier le type de valeur attendu pour compléter le texte:

  %s : chaîne de caractères
  %c : un caractère
  %d : une valeur de nombre entier (de type char, int, ou short)
  %ld : une valeur de nombre entier (de type long)
  %f : une valeur de nombre réel
  %x : une valeur entière en hexadecimal
  …

La fonction `printf`

- Signature de la fonction:
  `int printf( const char *format [, arg1 [, arg2]...]);`
- Paramètres de la fonction:
  - `format` : une chaîne de caractère formatée (voir slide précédent);
  - `arg1, arg2, arg3` …: une liste d'arguments dont le nombre n'est pas défini mais **doit correspondre au nombre de champs à compléter** dans la chaîne de caractères format; le type de chaque argument **doit également correspondre au type** spécifié dans le format correspondant du texte à trou.
- Les trous dans le texte `format` sont remplacés par les valeurs des arguments, en respectant leur ordre d'apparition: le 1er trou par le 1er argument, le 2ème trou par le 2ème argument, etc.
- Reprenons l'exemple précédent:

```
printf("Ceci est un %s, il manquait %d mot(s) dans cette phrase", "exemple", 1);
printf("Ceci est un %s, il manquait %d mot(s) dans cette phrase", "texte complet", 2);
```

Ce code produit le texte suivant sur la sortie standard (i.e. le terminal par défaut):

Ceci est un exemple, il manquait 1 mot(s) dans cette phrase
Ceci est un texte complet, il manquait 2 mot(s) dans cette phrase

La fonction `fprintf`

- Signature de la fonction:
  `int fprintf(FILE * file, const char *format [, arg1 [, arg2]...]);`
- Par rapport à la fonction `printf`, le seul argument supplémentaire est `file` : un descripteur de fichier. Nous verrons plus en détail à quoi cela correspond dans la partie du cours sur les systèmes d'exploitation.
- Ici, nous considérons `file` comme un objet qui permet d'accéder à un fichier sur le disque. `fprintf` écrit le texte à trou complété dans ce fichier plutôt que sur la sortie standard.
- En termes d'usage, on peut obtenir cet objet de type `FILE *` comme suit:

  ```
  File * f = fopen(file_name, "rw");
  ```

  Ici, `file_name` est un chemin vers le fichier (`"mon_fichier"` si mon fichier se trouve dans le répertoire courant, ou `"../mon_fichier"` s'il se trouve dans le répertoire parent, ou un chemin absolu, etc.

  "rw" que le programme aura accès au fichier en lecture et en écriture; autres possibilités:
  "r", accès en lecture seulement,
  "w", accès en écriture seulement.

```
fprintf(f, "Ceci est un %s, il manquait %d mot(s) dans cette phrase", "exemple", 1);
```

La fonction `scanf` principes de fonctionnement

- Signature de la fonction:

```
int scanf( const char *format [, arg1 [, arg2]...]);
```

- Paramètres de la fonction:
  - `format` : une chaîne de caractère formatée (voir slide précédent);
  - `arg1, arg2, arg3` …: une liste d'argument dont le nombre n'est pas défini mais **doit correspondre au nombre de champs à compléter** dans la chaîne de caractères format; le type de chaque argument **doit correspondre à un pointeur vers le type** spécifié dans le format correspondant du texte à trou.

- Les trous dans le texte `format` permettent d'identifier des valeurs avec lesquelles seront initialisés les arguments, en respectant leur ordre d'apparition: le 1er argument avec la 1er valeur, le 2ème argument avec la deuxième valeur identifiée, etc.

- La fonction scanf est bloquante, en attente que l'utilisateur fournisse une chaîne de caractère sur l'entrée standard (par défaut les chaînes de caractère tapés sur le terminal)

La fonction `scanf` exemple

- Considérons le code:

```
#include <stdio.h>

int main ()
{
 char str [80];
 int i;

 printf ("Enter your family name: ");
 scanf ("%s", str);
 printf ("Enter your age: ");
 scanf ("%d", &i);
 printf ("Mr. %s , %d years old.\n",str,i);
}
```

`str`, le paramètre est bien une adresse

`&i`, le paramètre est bien une adresse

- Le comportement sera le suivant:

  le programme affiche `Enter your family name:`
  le programme attend (il est bloqué) qu'on entre un texte sur l'entrée standard
  je tape `Borde` et j'appuie sur *« Entrée » (retour à la ligne)*
  le programme affiche `Enter your age:`
  le programme attend (il est bloqué) qu'on entre un texte sur l'entrée standard
  je tape `33` et j'appuie sur *« Entrée »*
  le programme affiche `Mr. Borde, 33 years old.`

- Ce programme montre bien que les variables `str` et `i` ont respectivement été initialisées avec `"Borde" et "33"`

La fonction `fscanf`

- La fonction `fscanf` n'est pas bloquante, car elle utilise le contenu d'un fichier existant pour initialiser les paramètres

- Vous verrez son utilisation en TP directement (les principes de base sont similaires à ceux que nous avons vus dans les slides précédents).

Lire un fichier : exemple

- Soit le fichier `prog1.c` (l'exécutable va ouvrir et lire ce fichier source) :

```
#include <stdio.h>
#define NMOTS 10

int main(int argc, char*argv[]){
  FILE *fichier;
  char mot[30];
  int i, ret_lec;

/ **** ouverture du fichier en lecture ****/
  fichier = fopen("prog1.c", "r");
    if(fichier == NULL){
       printf("le fichier n'existe pas\n");
       return 0;
       }

 /**** lire des mots dans le fichier et les afficher ****/
  for(i=0; i<=NMOTS; i++){
    ret_lec = fscanf(fichier, "%s", mot);
      if(ret_lec != 1) printf ("i = %d : pb lecture\n",
i);
         printf ("mot %d : %s \n", i, mot );
     }
  return 1;
}
```

Résultat:

```
mot 0 : #include
mot 1 : <stdio.h>
mot 2 : #define
mot 3 : NMOTS
mot 4 : 10
mot 5 : int
mot 6 : main(int
mot 7 : argc,
mot 8 : char*argv[]){
mot 9 : FILE
mot 10 : *fichier;
```

Famille printf/scanf (E/S dites « formattées »)

	Lecture	Ecriture
Forme générale	`fscanf (fichier, format, liste arguments)`	`fprintf (fichier, format, liste arguments)`

Clavier/ écran	`scanf(format, liste arguments )`
	équivalent à `fscanf` sur `stdin`:
	`fscanf (stdin, format, liste arguments )`
		`printf (format, liste arguments )`
		équivalent à fprintf sur `stdout` :
		`fprintf (stdout, format, liste arguments)`

Zone mémoire	`sscanf (adresse mémoire, format, liste arguments )`	`sprintf (adresse mémoire, format, liste arguments )`

Fonctionnement des fonctions du type `fscanf`:

- A chaque directive (notation %) du format doit être associée une adresse,
- Ces fonctions renvoient le nombre d'éléments reconnus et affectés ;

Famille printf/scanf : sprintf

- Exemple d'écriture « formatée » en mémoire :

```
/* construire un entete du type : Paris, le lundi 1 avril  */
   char entete[128];
   char jour[] ="lundi", mois[] ="avril";
   int num = 1;

   sprintf(entete, "Paris, le %s %d  %s ", jour, num, mois)
```