IN102 Polycopié

Goran Frehse
basé sur le cours de François Pessaux
October 13, 2020

Contents

1 Un peu de C

1.1 Un peu d'histoire

- Voici Ken Thompson et Dennis Ritchie, chercheurs à Bell Labs en 1969

- En 1969, ils avaient écrit le système d'exploitation Unix (précurseur de Linux) *en assembleur* pour une PDP-7.
- La galère : 4501 lignes de code assembleur.

- En 1970, ils étaient en charge de l'adapter pour la nouvelle PDP-11.
- Tout était à refaire ...

- Pour faciliter la tâche, ils ont crée un langage qui permet d'écrire un programme une fois et de le traduire (compiler) automatiquement sur n'importe quelle machine.

- En 1972, ce langage est devenu C.
- Depuis 1973, Unix / Linux est écrit en C, et fait aujourd'hui tourner une bonne majorité d'ordinateurs (smartphone,...).
- Depuis 1988, la norme **POSIX** (Portable Operating System Interface) garantit un environnement uniforme pour compiler et lancer des programmes en C.
- compatible POSIX : MacOS, Linux, Android, iOS

1.2 Pourquoi apprendre C?

- classé no 1 ou 2 parmis les langages de programmation le plus utilisés depuis 30 ans. Tiobe-Index:

Programming Language	2019	2014	2009	2004	1999	1994	1989
Java	1	2	1	1	3	-	-
C	2	1	2	2	1	1	1
Python	3	7	6	6	22	20	-
C++	4	4	3	3	2	2	2

- très proche du matériel
- fait comprendre comment marche un ordinateur
- disponible sur n'importe quel ordinateur, du micro-onde jusqu'au supercalculateur
- très rapide

1.3 Exemple: Hello World

Hello World: petit programme introduit par Dennis Ritchie dans son livre sur C, en 1978

Voici le fameux programme "Hello World" en C:

```
#include <stdio.h>
int main ()
{
   printf("Hello World !!\n");
   return 0;
}
```

```
Hello World !!
```

Executer le code suivant avec `Shift-Entrée` où avec le bouton "Run":

```
/* Mon premier programme en C */
#include <stdio.h>
int main ()
{
   printf("Hello World !!\n");
   return 0;
}
```

```
Hello World !!
```

1.4 Comprendre le Hello World

Nous allons présenter la syntaxe (les réglès de forme) du langage C à l'exemple du programme "Hello World".

1.4.1 Bases de Syntaxe

- chaque instruction est suivie d'un ;
- les espaces, tabulations, etc. n'ont aucune importance

```
printf ( "Hello" ) ;
```

```
printf("Hello");
```

```
printf(   "Hello"              )         ;
```

- définir un **bloc d'instructions** avec {et }

```
{
   printf("Hello ");
   printf("World.");
}
```

1.4.2 Commentaires

```
/* un commentaire...
     ... sur plusieurs lignes ...
          ... voilà.                */

// commentaire sur une seule ligne
```

Quand on écrit un programme, on **commence avec les commentaires!**

1.4.3 Fonction: main

Format d'une **fonction** :

...type de retour... `nom_de_fonction ( ...` arguments `... ) {`

```
... instructions ...
}
```

Ici :

```
int main ( ) {
... instructions ...
}
```

Quand un programme est lancé, c'est la fonction `main` qui est appelé.

Une fonction peut donner une valeur de retour :

...type de tour... `nom_de_fonction ( ...` arguments `... ) {` > ... instructions ... > `return ...` valeur de retour `... ;`

```
}
```

Dans le main, toujours: - type de retour: `int` (entier) - valeur de retour: `0` si tout va bien; autre valeur indique un problème

Exemple d'utilisation dans le shell: Afficher "ok" si tout va bien.

```
./test1 && echo "ok"
```

1.4.4 Include

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

Pour voir ce que ça fait, on l'enlève du programme en le passant en commentaire :

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

```
test1.c: In function ‘main’: test1.c:4:1: warning: implicit
declaration of function ‘printf’ [-Wimplicit-function-declaration]
printf("Hello World !!\n");  ^~~~~~ test1.c:4:1: warning:
incompatible implicit declaration of built-in function ‘printf’
test1.c:4:1: note: include ‘<stdio.h>’ or provide a declaration of
‘printf’
```

stdio.h déclare `printf`. On peut regarder le contenu de stdio.h, qui se trouve dans `/usr/include/stdio.h...`

1.4.5 Compiler

Un programme s'écrit sous forme d'un texte « source ». On utilise donc un éditeur de texte. Une fois écrit, il faut transformer ce texte en exécutable, c'est à dire dans une suite d'instructions connus au processeur. On utilise un **compilateur** C, dans ce cours ça sera le "GNU C Compiler" `gcc`. Dans un terminal :

```
gcc -Wall -Wfatal-errors monprogramme.c
```

Le fichier exécutable créée par défaut se nomme `a.out`. Vous pouvez alors exécuter votre programme en tapant dans un terminal :

```
./a.out
```

1.4.6 Linker

Des gros projets contiennent des milliers de fonctions et de millions de lignes de code. Tout ça c'est écrit par des équipes de plusieurs personnes.

Pour mieux organiser le travail, les fonctions sont **distribués sur plusieurs fichiers**.

Compiler le programme en entier peut durer plusieurs minutes, voir heures. On compile donc **un fichier de code à la fois**. Mais le compilateur doit connaître les fonctions données dans les autres fichiers...

La solution c'est de **linker** le code à la fin (édition de lien).

Regardons ce qui se passe si on n'a pas de fonction qui s'appelle `main`:

```
#include <stdio.h>
int main2 ()
{
   printf("Hello World !!\n");
   return 0;
}
```

```
/usr/lib/gcc/x86_64-linux-gnu/7/../../../x86_64-linux-gnu/Scrt1.o :␣
 ↪Dans la
fonction « _start » :
(.text+0x20) : référence indéfinie vers « main »
collect2: error: ld returned 1 exit status
```

`ld` est le nom du linker. Il râle parce qu'il n'a pas trouvé de fonction "main". Pourtant, il en a besoin parce que c'est la fonction qui sera appelé quand on démarre le programme!

2 Création d'un programme

2.1 Mettre en place un terminal et un éditeur

1. Démarrer Ubuntu VM... Terminal...
2. Ouvrir un Terminal, Super-Left pour l'afficher sur la gauche. (Super = Windows ou Command)
3. Ouvrir gedit (editeur de texte), Super-Right pour l'afficher sur la droite.

On peut basculer entre terminal et editeur avec Super-Tab.

2.2 Organiser son travail dans un répertoire

- créer le répertoire avec `mkdir cours1`
- changer vers ce répertoire avec `cd cours1`

2.3 Editer et sauvegarder

On supposera que vous savez editer et sauvegarder des fichiers... En revanche, prenez l'habitude de *sauvegarder votre travail* (backup). C'est embêtant de perdre un programme sur lequel vous avez travaillé pendant deux heures, juste à la fin de l'examen (voir exemple en bas).

Une solution rapide est de faire un zip de votre répertoire de temps en temps:

```
zip -r cours1_debut.zip cours1
```

puis changez le nom de temps en temps:

```
zip -r cours1_moitie.zip cours1
```

et

```
zip -r cours1_presque_fini.zip cours1
```

Attention: Sans l'option `-r` votre zip contiendra un dossier vide!

2.4 Compiler et executer

```
gcc test.c
```

La commande `ls` montre que le fichier `a.out` a été créé.

On l'execute avec `./a.out` parce que juste `a.out` ne cherche *pas* dans le répertoire courant (pour éviter toute ambiguité).

Pour changer le nom du fichier produit avec l'option `-o`. On peut récuperer la ligne précédente du terminal avec la flêche vers le haut, puis l'éditer.

```
gcc -o test1 test1.c ls ./test1
```

Attention: Une erreur courante est de taper trop vite et faire

```
gcc -o test1.c test1
```

C'est quoi le problème? Eh oui... le texte du programme qui était dans test1.c est perdu.

2.5 Erreurs et Warnings (IMPORTANT)

Si le compilateur voit un problème avec votre programme, il produit des **warnings** (avertissements) et des messages d'**erreurs**. Pour les warnings, le compilateur ne s'arrête pas. Il est quand même fortement recommandé de les regarder et de les

résoudre! Pour les erreurs, il faut savoir que seulement la première erreur compte. Tous les autres peuvent être un effet secondaire de la première erreur.

Options du compilateur `gcc` essentielles:

- `-Wall`: activer tous les avertissements
- `-Werror`: traîter les warnings comme des erreurs
- `-Wfatal-errors`: s'arrêter à la première erreur

Imperatif pour ce cours: Toujours utiliser les trois options essentielles!

```
gcc -Wall -Werror -Wfatal-errors test1.c
```

2.6 Pour gagner du temps

Si vous lancez la compilation depuis un terminal, vous n'êtes pas obligés de retaper les commandes à chaque fois. Il y a deux façons de réutiliser vos commandes précédentes:

- **flèche vers le haut**: parcourt l'historique de toutes vos commandes
- `Ctrl-r` **+ texte**: cherche la dernière commande qui commence avec ce texte. Encore une fois `Ctrl-r`cherche l'avant-dernière etc.

2.7 Exercice: Bonjour

Dans la cellule ci-dessous, écrivez un programme C qui affiche "Bonjour". Rappel: Pour executer le programme dans ce notebook, taper `Shift-Entré`.

```
/* Ecrivez le programme ici ...*/
```

```
  File "<ipython-input-8-cd37fdb3b7d0>", line 1
*(Ecrivez, le, programme, ici, ...*/)
                                    ^
SyntaxError: invalid syntax
```

3 Types et variables

Les informations que l'on souhaite traiter peuvent être de natures variées : entiers naturels, rationnels, valeurs de vérité (vrai / faux), lettres, texte ...

Les langages de programmation fournissent des types de données variés à cet effet. Le type est nécessaire pour :

- savoir combien de place (octets) réserver en mémoire
- appeler la bonne version des fonctions et opérateurs +,-,/, etc. qui sont différent au niveau materiel si entiers ou flottants

3.1 Types de base

En C, il y a 3 types de base pour représenter des nombres:

- les entiers (`int`),
- les réels (`float` et `double`),
- les caractères (`char`) – pour l'ordi ce n'est qu'un nombre.

On appelle **booléen** le type des valeurs de vérité (« vrai » et « faux »). C ne fournit pas de vrais booléens. En C, ce sont des entiers avec pour convention $0 \equiv$ « faux » et (tout sauf 0) $\equiv$ « vrai ».

3.1.1 Nombres entiers

Les types de nombres entiers existent en version avec signe (`signed`) ou sans signe (`unsigned`):

- `unsigned char` : caractère sur 1 octet, valeurs 0...255
- `signed char` : caractère sur 1 octet, valeurs -128...127
- `char` : peut être signed ou unsigned (voir limit.h)
- `int` : entiers sur au moins 2 octets, mais très souvent 4 octets = 32 bit valeurs de -2^{31} à $2^{31}-1$ (1 bit pour le signe) = -2,147,483,648 à 2,147,483,647

3.1.2 Nombres flottants

- `float` : flottants de basse précision, sur 4 octets
- `double` : flottants de haute précision, sur 8 octets
- ... d'autres ...
- pas de boolean, utiliser `int` : faux = `0` vrai = toute autre valeur (1,-1,...)

La taille en octets peut varier selon la machine et le compilateur; si besoin consulter `limit.h` (cours ultérieur).

3.2 Déclaration de variables

Un programme manipule des données stockées en mémoire. Il faut donc pouvoir manipuler des « réceptacles » d'information : des variables. En C, on déclare une variable en donnant son type suivi de son nom :

```
int compteur;
```

On peut initialiser une variable directement au moment de sa déclaration en lui affectant une valeur par la construction = :

```
int compteur = 0;
```

3.3 Afficher une variable avec `printf`

Afficher une variable (entier):

```
#include <stdio.h>
int main() {
   int i = 23;
   printf("hello %d bye bye",i);
}
```

%d est remplaçé par la variable après le virgule.

Afficher plusieurs variables (entier):

```
#include <stdio.h>
int main() {
   int i = 23;
   int j = 17;
   int k = 3;
   printf("d'abord %d, après %d, ensuite %d",i,j,k);
   return 0;
}
```

```
d'abord 23, après 17, ensuite 3
```

Les %d sont remplaçés dans l'ordre d'apparition par les variables après le virgule.

Afficher sur plusieurs lignes avec \n:

```
#include <stdio.h>
int main() {
   int i = 23;
   int j = 17;
   int k = 3;
   printf("d'abord %d,\naprès %d,\nensuite %d",i,j,k);
   return 0;
}
```

```
d'abord 23,
après 17,
ensuite 3
```

Codes utiles:

- %c: caractère (ASCII)
- %d: entier
- %f: flottant (float ou double)
- %g: flottant en format compact
- ... d'autres ...

```
#include <stdio.h>
int main() {
   double x = 12345;
   printf("%f\n",x);
   printf("%g\n",x);
}
```

```
12345.000000
12345
```

Précision:

- %.3f : avec 3 décimales
- %.3g : avec 3 chiffres significatives

```
#include <stdio.h>
int main() {
   double x = 12345;
```

```
   printf("%.3f\n",x);
   printf("%.3g\n",x);
   printf("%.4g\n",x);
   printf("%.5g\n",x);
}
```

```
12345.000
1.23e+04
1.234e+04
12345
```

Alerte Bug: Ce que donne le programme suivant?

```
#include <stdio.h>
int main() {
   double x = 1.23;
   printf("%d",x);
}
```

```
-650301784
```

Les octets de x sont traités comme si c'était un entier – puisque le codage est complètement différent ça donne n'importe quoi.

3.4 Les flottants

zéro : fraction et exposant sont zéro précision : $2^{-52} \approx 2.2210^{-16}$ (*machine epsilon*)

3.5 Exercice: Précision

Pour tester la précision, affichez les valeurs - 1 - $1 + 110^{-16}$ - $1 + 210^{-16}$

avec précision maximale en utilisant le format `%.17g`.

```
#include <stdio.h>
int main() {
    /* ... a compléter ... */
}
```

```
#include <stdio.h>
int main() {
    printf("%.17g\n", 1.0 );
    printf("%.17g\n", 1.0 + 1e-16 );
    printf("%.17g\n", 1.0 + 2e-16 );
}
```

```
1
1
1.0000000000000002
```

3.6 Portée des variables

Une variable dans un bloc `{... }` est reconnue seulement dans ce bloc:

```
{
int i = 123;
}
```

On l'appele une variable **locale**.

Les variables qui ne sont dans aucun bloc sont des variables **globales**, disponibles partout dans le programme. Les variables globales entraînent facilement des bugs, donc on **préfère des variables locales**.

Elle est distinct des autres variables en dehors du bloc, même des celles qui portent le même nom.

```
int i = -7;
{
int i = 123;
printf("%d ",i);
}
printf("%d ",i);
```

affiche: `123 -7`

La variable est 'détruite' à la fin du bloc : la place réservée pour la mémoire est libérée et peut désormais être utilisée pour d'autres variables.

Les blocs peuvent être imbriqués :

```
int i = 4;
{
int i = -7;
{
int i = 123;
printf("%d ",i);
}
printf("%d ",i);
}
printf("%d ",i);
```

affiche: `123 -7 4`

4 Expressions et instructions

En simplifiant, on différencie 2 types de constructions de C :

- Les expressions, qui « ont une valeur » (elles « valent ») et n'ont pas d'effet.
- Les instructions, qui « ont un effet » (elles « font ») sans forcément valoir quelque chose. La différence réelle entre instruction et expression est plus complexe et plus floue en C.

4.1 Expressions de base

Expressions de base :

- Variables : toute variable d 'eclar 'ee et de port 'ee accessible.
- Entiers : 344578
- Caractères : 'U', '\n' (**notez les guillemets simples**)
- Flottants : −4.6, 5e-12
- Chaînes : "plop" (**guillemets doubles**) "\tGlop\n"

4.2 Expressions arithmétiques

- comme d'habitude : `x + y`, `x - y`, `x * y`
- **attention à la division** : `x / y`
  - division entière si `x`,`y` sont `int`
  - division flottante si un des deux `float` ou `double`
- modulo (reste de la division entière): `x % y`

```
#include <stdio.h>
int main() {
   int x = 7;
   int y = 3;
   printf("%d", x / y );
}
```

`2`

```
#include <stdio.h>
int main() {
   double x = 7;
   double y = 3;
   printf("%f", x / y );
}
```

`2.333333`

4.3 Exercice: Divison entière

Affichez le résultat du calcul `(x+y)/z` avec x=1, y=2, z=5

- d'abord tous `int`,
- ensuite un seul en `double`.

Est-ce que ça fait une différence lequel est `double`?

```
#include <stdio.h>
int main() {
   /** A compléter **/
}
```

4.4 Affectation et test d'égalité

- l'affectation `x = y` donne comme valeur de retour la nouvelle valeur de `x`
- ça permet d'écrire par exemple `x = y = z = 3` (déconseillé)

attention: facile à confondre avec le test d'égalité `x == y`

```
#include <stdio.h>
int main() {
   int x = 2;
   int y = 3;
   printf("%d\n", x==y );
   printf("%d\n", x );
   printf("%d\n", x=y );
   printf("%d\n", x );
}
```

```
0
2
3
3
```

Pour tester si x et y sont différentes, on peut utiliser `x != y`.

4.5 Opérations logiques

- et logique : `x && y`
- ou logique : `x || y`
- négation : `!x`

```
#include <stdio.h>
int main() {
   int x = 3; // vrai, car pas 0
   int y = 0; // faux, car 0
   printf("%d\n", x && y);
}
```

```
0
```

```
#include <stdio.h>
int main() {
   int x = 3; // vrai, car pas 0
   int y = 1; // vrai, car pas 0
   printf("%d\n", x && y);
   printf("%d\n", !0 );
}
```

```
1
1
```

attention: - et par bit : `x & y` - ou par bit : `x | y` - négation par bit : `~x`

```
#include <stdio.h>
int main() {
   int x = 2; // vrai, car pas 0
   int y = 1; // vrai, car pas 0
   printf("%d\n", x && y);
   printf("%d\n", x & y);
}
```

```
1
0
```

2 en binaire = `10`, 1 en binaire = `01` `2 && 1` en binaire = `00`

4.6 Exponentiation

- **attention**: `x ^ y` = opération logique “ou exclusif”!

```
#include <stdio.h>
int main() {
   double x = 2;
   double y = 3;
   printf("%f", x ^ y);
}
```

```
/tmp/tmpyqddk9nb/7bf94ebc-5c59-4cce-9e95-c6cea7b2acf1.c: In function␣
↪‘main’:
/tmp/tmpyqddk9nb/7bf94ebc-5c59-4cce-9e95-c6cea7b2acf1.c:5:19: error:␣
↪invalid
operands to binary ^ (have ‘double’ and ‘double’)
    printf("%f", x ^ y);
```

```
                    ^
compilation terminated due to -Wfatal-errors.
```

- utiliser `pow` de math.h

```
#include <stdio.h>
#include <math.h>     // pour pow
int main() {
   double x = 2;
   double y = 3;
   printf("%f", pow(x,y) );
}
```

```
gcc test3.c
```

```
/tmp/cceMWz9n.o : Dans la fonction « main » :
test3.c:(.text+0x39) : référence indéfinie vers « pow »
test3.c:(.text+0x65) : référence indéfinie vers « pow »
collect2: error: ld returned 1 exit status
```

C'est quoi le problème?

C'est le **linker** (ld) qui rale. Les fonctions dans `math.h`sont déjà précompilées dans un fichier `m`.

Il faut dire au linker de les retrouver dans `m`:

```
gcc test3.c  -lm
```

Attention: L'option `-lm` doit venir *à la fin*, sinon le linker ne la voit pas!

```
#include <stdio.h>
#include <math.h>
int main() {
   double x = 2;
   double y = 3;
   printf("%f", pow(x,y));
}
```

```
8.000000
```

5 Branchements et boucles

5.1 Branchement if-else

L'instruction conditionnelle permet effectuer un traitement si une condition est vraie:

```
if (expression) {
instruction(s) `;`
}
```

... ou (optionnellement) un autre traitement sinon :

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

```
#include <stdio.h>
int main() {
   int n = -1;
   if (n<0) {
      return 0;
   } else {
       // On continue...
   }
}
```

Attention:

- parenthèses obligatoires après `if`!
- définir un bloc avec `{`et `}` très fortement conseillé !

5.2 Boucle while

L'instruction while permet effectuer une boucle tant qu'une condition est vraie:

`while ( expression ) {`

`instruction(s) ;`

`}`

```
while (<test>) {
  ... corps de la boucle ...
}
```

`<test>` est executé:

- si 0 (faux), on continue après la boucle,
- autrement (vrai), on execute le corps de la boucle.

5.3 Exercice: Trouver la précision du type double

La *précision* d'un type de variable est la plus petite valeur qu'on peut toujours distinguer de 0 lors d'une opération. Pour l'addition, on appelera la précision la valeur x tel que $1 + x \neq 1$ mais $1 + y = 1$ pour tous les $y < x$.

Ecrivez un programme qui définit une variable `x` de type `double` qui vaut d'abord `1.0`. Ensuite, on divise `x` par `2.0` tant que `1.0 + x` est différent de `1.0`. Finalement, on multiplie `x` par `2.0` et affiche sa valeur.

Ca donne la structure suivante :

```
// On déclare x et lui donne la valeur 1.0.
while (<si 1.0+x est différent de 1.0>) {
  // diviser x par 2.0
}
// multiplier x par 2.0
// afficher x
```

```
#include <stdio.h>
int main() {
 // à vous de jouer...
}
```

```
#include <stdio.h>
int main() {
  double x = 1.0;
  while (1.0+x != 1.0) {
    x = x/2.0;
  }
  x = x*2.0;
  printf("précision de double: %g",x);
}
```

```
précision de double: 2.22045e-16
```

5.4 Boucle for

La structure suivante est extrèmement courrante :

```
int i = 1;
while (i<=n) {
  ... calcul ...
  i = i+1;
}
```

plus généralement :

```
<instruction1>
while (<test>) {
  ... calcul ...
  <instruction2>
}
```

100% équivalent:

```
for (<instruction1>; <test>; <instruction2>) {
  ... calcul ...
}
```

Après chaque `<instruction2>`, `<test>` est executé pour voir si on arrête la boucle.

5.5 Exemple: Multiples de 3

Ecrivez un programme pour afficher les multiples de 3 jusqu'à $3n$ pour un nombre `n` donné.

```
#include <stdio.h>
int main() {
 // à vous de jouer...
```

```
}
```

```
#include <stdio.h>
int main() {
    int n = 9;
    for (int i = 1; i<=n; ++i) {
        printf("%d ",3*i);
    }
}
```

```
3 6 9 12 15 18 21 24 27
```

5.6 Exemple: Affichage d'une série avec virgules

Affichez les nombres carrés de 1^1 jusqu'à n^2, séparés par des virgules (sans virgule à la fin, ni au début). On supposera $n = 10$.

```
#include <stdio.h>
int main() {
 // à vous de jouer...
}
```

```
#include <stdio.h>
int main(){
    int n = 10;
    for (int i=1; i<=n; ++i) {
        if (i>1) {
            printf(",");
        }
        printf("%d",i*i);
    }
}
```

```
1,4,9,16,25,36,49,64,81,100
```

5.7 Boucles imbriquées

Souvent il faut répéter une tâche plusieurs fois, et la tâche est elle-même une répétition d'actions. On fait alors une boucle dans une boucle.

5.8 Exercice: Table de Multiplication

Ecrivez un programme pour afficher la table de multiplication jusqu'à `n`x`n` pour un nombre `n` donné.

On **commence** à écrire un programme en le décrivant **avec des commentaires** :

```
#include <stdio.h>
int main() {
    // Définir le nombre de tours n
    // Boucle: parcourir toutes les valeurs de i=1 à i=n
        // Boucle: parcourir toutes les valeurs de j=1 à j=n
            // afficher i x j = i*j

    // à vous de jouer...
}
```

```
#include <stdio.h>
int main() {
    // Définir le nombre de tours n
    int n = 4;
    // Boucle: parcourir toutes les valeurs de i=1 à i=n
    for (int i = 1;i<=n;++i) {
        // Boucle: parcourir toutes les valeurs de j=1 à j=n
        for (int j = 1; j <= n; ++j) {
            // afficher i x j = i*j
            printf("%d x %d = %3d   ",i,j,i*j);
        }
        printf("\n");
    }
}
```

```
1 x 1 =   1   1 x 2 =   2   1 x 3 =   3   1 x 4 =   4
2 x 1 =   2   2 x 2 =   4   2 x 3 =   6   2 x 4 =   8
```

```
3 x 1 =   3   3 x 2 =   6   3 x 3 =   9   3 x 4 =  12
4 x 1 =   4   4 x 2 =   8   4 x 3 =  12   4 x 4 =  16
```

5.9 Boucle do-while

La boucle **while** est parfois pas très élégante :

```
#include <stdio.h>
int main() {
    int reponse;
    printf("Entrez 4 pour quitter: ");
    scanf("%d",&reponse);
    while (reponse!=4) {
        scanf("%d",&reponse);
    }
}
```

`scanf`est écrit deux fois : deux fois plus de chance d'un bug

variante : boucle `do-while` execute son corps au moins une fois

```
#include <stdio.h>
int main() {
    int reponse;
    printf("Entrez 4 pour quitter: ");
    do {
        scanf("%d",&reponse);
    } while (reponse!=4)
}
```

6 Pointeurs

Chaque variable est stocké dans la mémoire à une adresse réservée pour elle toute seule.

Cette adresse est un nombre entier positif, sur 32 ou 64 bit (selon la machine).

```
int i = 123;
double x = 0.1;
```

variable	adresse	contenu
x	1028	0.1
i	1024	123

On peut faire des calculs très malin si on stocke cette adresse aussi dans une variable : un **pointeur**.

Un pointeur est une variable qui contient l'adresse de mémoire d'une autre variable.

Un pointeur a le type de la variable cible, suivi par `*`: `int* px`

On obtient l'adresse du cible avec `&`: `px = &x`

```
int i = 123;
double x = 0.1;
int* pi = &i;   // pointer vers i
double* px = &x;   // pointer vers x
```

variable	adresse	contenu
px	1044	1028
pi	1036	1024
x	1028	0.1
i	1024	123

Un pointeur peut pointer vers un autre pointeur:

```
int i = 123;
double x = 0.1;
int* pi = &i;   // pointeur vers i
double* px = &x;   // pointeur vers x
double** ppx = &px; // pointeur vers px
```

variable	adresse	contenu
ppx	1052	1044
px	1044	1028
pi	1036	1024
x	1028	0.1
i	1024	123

```
int main() {
    int i = 123;
    double x = 0.1;
    int* pi = &i;   // pointeur vers i
    double* px = &x;   // pointeur vers x
    double** ppx = &px; // pointeur vers px
}
```

Avec un pointeur on peut modifier la variable cible ajoutant un & devant le pointeur :

```
int i = 123;
double x = 0.1;
int* pi = &i;   // pointeur vers i
```

variable	adresse	contenu
pi	1036	1024
x	1028	0.1
i	1024	123

```
*pi = -7;
```

variable	adresse	contenu
pi	1036	1024
x	1028	0.1
i	1024	-7

a retenir :

pointeur à partir d'une variable : `&x`

variable à partir d'un pointeur : `*px`

Les pointeurs seront essentiel pour les fonctions...

7 Fonctions

Format d'une fonction :

...type de retour... `nom_de_fonction ( ... arguments ... ) {`

```
... instructions ...
}
```

Les arguments sont des variables locales dont la portée est le bloc de la fonction:

```
int carre (int x) {   return x * x; }
```

7.1 Exercice : Conversion Fahrenheit-Celsius dans une fonction

```
#include <stdio.h>  /* Pour acceder a printf. */

int main () {
  double x = 80;                          // temperature en␣
 ↪Fahrenheit
  double y = 5.0 / 9 * x - 160.0 / 9;
  printf ("%.2f F -> %.2f C\n", x, y) ;

  return 0;
}
```

```
80.00 F -> 26.67 C
```

```
#include <stdio.h>  /* Pour acceder a printf. */

/* Convertir un température x de Fahrenheit en Celsius */
double F2C(double x) {
  double y = 5.0 / 9 * x - 160.0 / 9;
  return y;
}

int main () {
  double x = 80;                          // temperature en␣
 ↪Fahrenheit
  double y = F2C(x);
  printf ("%.2f F -> %.2f C\n", x, y) ;

  return 0;
}
```

```
80.00 F -> 26.67 C


double F2C(double x) {
    double y = 5.0 / 9 * x - 160.0 / 9;
    return y;
}
int main () {
<@> double x = 80; // x est alloué 8 octets et 80 y est copiée
    double y = F2C(x);
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu
x (main)	1028	80

```
double F2C(double x) {
<@> // la variable locale x est crée,
    // et la valeur de l'argument effectif y est copié
    double y = 5.0 / 9 * x - 160.0 / 9;
    return y;
}
int main () {
    double x = 80;
    double y = <@>F2C(x);
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu
x (F2C)	1044	80
y (main)	1036	???
x (main)	1028	80

```
double F2C(double x) {
<@> double y = 5.0 / 9 * x - 160.0 / 9; // y est créé et affecté sa valeur
    return y;
}
int main () {
    double x = 80;
    double y = <@>F2C(x);
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu (4 octets)
y (F2C)	1052	26.66667
x (F2C)	1044	80
y (main)	1036	???
x (main)	1028	80

```
double F2C(double x) {
    double y = 5.0 / 9 * x - 160.0 / 9;
<@> return y; // la valeur de retour est copié dans un tampon (registre)
}
int main () {
    double x = 80;
    double y = <@>F2C(x);
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu
val. retour	registre	26.66667
y (F2C)	1052	26.66667
x (F2C)	1044	80
y (main)	1036	???
x (main)	1028	80

```
double F2C(double x) {
    double y = 5.0 / 9 * x - 160.0 / 9;
    return y;
<@> } // fin du bloc : variables locales libérées
int main () {
    double x = 80;
    double y = <@>F2C(x);
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu
val. retour	registre	26.66667
y (main)	1036	???
x (main)	1028	80

```
double F2C(double x) {
    double y = 5.0 / 9 * x - 160.0 / 9;
    return y;
}
int main () {
    double x = 80;
<@> double y = F2C(x); // val. de retour copié dans y
    printf ("%.2f F -> %.2f C\n", x, y) ;
    return 0;
}
```

variable	adresse	contenu (4 octets)
y (main)	1036	26.66667
x (main)	1028	80

7.2 Exemple: Fonction d'affichage

Une fonction qui ne donne pas de valeur de retour est déclaré avec :

`void` nom-de-fonction(...)

```
#include <stdio.h>  /* Pour acceder a printf. */

/* Convertir un température x de Fahrenheit en Celsius */
double Fahrenheit2Celsius(double x) {
  double y = 5.0 / 9 * x - 160.0 / 9;
  return y;
}

int main ()
{
  double x = 80;                          // temperature en␣
 ↪Fahrenheit
  double y = Fahrenheit2Celsius(x);
  printf ("%.2f F -> %.2f C\n", x, y) ;

  return 0;
}
```

```
80.00 F -> 26.67 C
```

```
#include <stdio.h>  /* Pour acceder a printf. */

/* Convertir un température x de Fahrenheit en Celsius */
double Fahrenheit2Celsius(double x) {
  double y = 5.0 / 9 * x - 160.0 / 9;
  return y;
}

/* Afficher le résultat */
void afficher_resultat(double x, double y) {
    printf ("%.2f F -> %.2f C\n", x, y) ;
}

int main ()
{
  double x = 80;                          // temperature en␣
 ↪Fahrenheit
  double y = Fahrenheit2Celsius(x);
  afficher_resultat(x,y);

  return 0;
}
```

```
80.00 F -> 26.67 C
```

7.3 Passage par valeur

Les arguments sont des **variables**:

```
int carre_par_valeur(int z) {
return z*z;
}
```

```
int main() {
int x = 3;
x = carre_par_valeur(x);
printf("%d",x);
}
```

1. Je fais une **copie** de x dans z.
2. Je fais un calcul avec.
3. Je copie le résultat du calcul dans la **valeur de retour** (registre).

7.3.1 Avantage de passage à valeur

- facile à écrire

7.3.2 Inconvénient de passage à valeur

- copies des valeurs (x vers z) –> lent (ça dépend)

7.4 Passage par adresse

L'argument est un **pointeur**:

```
void carre_par_adresse(int* px) {
(*px) = (*px) * (*px);
// rappel: (*px) identique à x
}

int main() {
int x = 3;
carre_par_adresse(&x);
printf("%d",x);
}
```

1. Je fais le calcul directement avec x.
2. Je manipule x via un pointeur.

7.4.1 Avantage de passage par adresse

- pas de copie –> rapide
- on peut manipuler plusieurs valeurs!

7.4.2 Inconvénient de passage par adresse

- plus difficile à écire

7.4.3 Exemple: Incrémenter

On essaie d'écrire une fonction qui incrémente la valeur de son argument par 1. Observez ce que fait la version avec passage par valeur:

```
#include <stdio.h>
void incrementer(int x) {
    x = x + 1;
}
int main() {
   int x = 7;
   incrementer(x);
   printf("%d",x);
}
```

```
7
```

x dans main ne change pas. Il est impossible de modifier x avec une fonction avec passage par valeur, sans passer par la valeur de retour.

Voici une version avec passage par adresse:

```
#include <stdio.h>
void incrementer(int* px) {
    *px = *px + 1;
}
int main() {
   int x = 7;
   incrementer(&x);
   printf("%d",x);
}
```

```
8
```

Cette fois, la variable x de main change de valeur. Avec passage par adresse, la fonction a accès à la variable x du main, en passant par le pointeur px.

7.4.4 Exemple: Echanger deux valeurs (swap)

```
#include <stdio.h>
void swap(int x, int y) {
   int temp = x;
   x = y;
   y = temp;
}
int main() {
   int x = 7; int y = 3; // <@>
   swap(x,y);
   printf("%d,%d",x,y);
}
```

```
7,3
```

```
#include <stdio.h>
void swap(int* px, int* py) {
   int temp = *px;
   *px = *py;
   *py = temp;
}
int main() {
   int x = 7; int y = 3; // <@>
   swap(&x,&y);
   printf("%d,%d",x,y);
}
```

```
3,7
```

7.5 Prototype de fonction

Le prototype, ou **déclaration**, d'une fonction est

valeur-de-retour `nom_de_fonction(`...arguments...`);`

Exemples:

```
double Fahrenheit2Celsius(double x);
```

```
void afficher_resultat(double x, double y);
```

```
void swap(int* px, int* py);
```

Pour utiliser la fonction dans **autre fichier**, il suffit de connaître son prototype.

Rappel: la **définition** d'une fonction inclut aussi son corps (les instructios entre accolades):

valeur-de-retour `nom_de_fonction(`...arguments...`) {`...instructions...`};`

8 Lire des Entrées au Clavier

Pour afficher une variable :

`` `printf("message: %d",x)` ``

Pour lire une valeur au clavier :

`` `scanf("%d", &x)` ``

Attention: Pas de message dans `scanf`! Donner seulement le format (`%d`,`%f`,...)

`&x`: l'adresse de `x`(pourquoi on utilise l'adresse?)

valeur de retour : nombre de valeurs lus

Pour afficher deux valeurs :

`` `printf("message: %d,%d",x,y)` ``

Pour lire deux valeurs au clavier :

`` `scanf("%d,%d", &x, &y)` ``

valeur de retour : nombre de valeurs lus

```
#include <stdio.h>
int main() {
    int x,y;
    printf("Entrer deux valeurs: ");
    int n = scanf("%d,%d",&x,&y);
    printf("x=%d,y=%d,n=%d\n",x,y,n);
}
```

```
$ ./a.out
Entrer deux valeurs: 3,4
x=3,y=4,n=2
$ ./a.out
```

```
Entrer deux valeurs: 3 4
x=3,y=1759981600,n=1
$ ./a.out
Entrer deux valeurs: 3 a
x=3,y=1604196336,n=1
```

8.1 Format de scanf

Utiliser les codes suivants selon le type de variable:

- `%d`: `int` (avec signe)
- `%u`: `unsigned int` (sans signe)
- `%f`: `float` (mais pas `double`)
- `%lf`: `double` (mais pas `float`)

Attention:

- `printf("%f",x);` marche si `x` est `float` et aussi si `x` est `double`.
- `scanf("%f",&x);` marche seulement si `x` est `float`!

9 Structures

Les structures répondent au besoin d'aggréger des données de types différents. Une structure est un groupement de données par champs nommés. Déclaration d'un type structure :

```
struct nom-type {
    nom-champ1 type-champ1 ;
    ... ;
    nom-champn type-champn ;
} ;
```

Déclaration d'une variable de type `struct` :

```
struct nom-type nom-variable
```

Comme pour les tableaux, il est possible d'initialiser directement les valeurs des champs d'une variable de type structure *lors de sa déclaration*, mais impossible d'affecter d'un seul coup une telle variable. L'accès à un champ de la structure se fait par notation pointée :

```
nom-variable.nom-champ
```

9.1 Exemple: Cercle

```
#include <stdio.h>

struct circle {
    int x ;
    int y ;
    unsigned int radius ;
};

int main() {
    struct circle c = { 100,50,7 };
    printf("Un cercle à (%d,%d) avec radius %d.\n",
          c.x,c.y,c.radius);
    struct circle d = c;
    printf("Une copie du cercle à (%d,%d) avec radius %d.\n",
          d.x,d.y,d.radius);
}
```

```
Un cercle à (100,50) avec radius 7.
Une copie du cercle à (100,50) avec radius 7.
```

Attention: Pour définir un cercle après sa première déclaration, on ne peut pas utiliser la notation `{ valeur, valeur, ... }`.

```
#include <stdio.h>

struct circle {
    int x ;
    int y ;
    unsigned int radius ;
};

int main() {
    struct circle c;
    c = { 100,50,7 }; // ne marche pas, car est trop tard
    printf("Un cercle à (%d,%d) avec radius %d.\n",
          c.x,c.y,c.radius);
}
```

```
/tmp/tmp6wxqtzk4/b07e8abb-b2f2-45a5-865a-d5911c2310e9.c: In function␣
 ↪‘main’:
/tmp/tmp6wxqtzk4/b07e8abb-b2f2-45a5-865a-d5911c2310e9.c:11:9: error:␣
 ↪expected
expression before ‘{’ token
     c = { 100,50,7 }; // ne marche pas, car est trop tard
         ^
compilation terminated due to -Wfatal-errors.
```

Une fois la structure est crée, il ne peut que modifier les champs un à la fois:

```
#include <stdio.h>

struct circle {
    int x ;
    int y ;
    unsigned int radius ;
};

int main() {
    struct circle c;
    c.x = 100;
    c.y = 50;
    c.radius = 7;
    printf("Un cercle à (%d,%d) avec radius %d.\n",
           c.x,c.y,c.radius);
}
```

```
Un cercle à (100,50) avec radius 7.
```

9.2 Pointeurs vers structures

Si p est un pointeur vers une structure avec un champ x, `p.x` ne marche pas. Il faut utiliser `p->x`.

Par exemple, on crée une fonction pour déplacer un cercle:

```
#include <stdio.h>

struct circle {
    int x ;
    int y ;
    unsigned int radius ;
};

void deplacer(struct circle* p, int dx, int dy) {
    p->x = p->x+dx;
    p->y = p->y+dy;
}

int main() {
    struct circle c = { 100,50,7 };
    deplacer(&c,3,4);
    printf("Un cercle à (%d,%d) avec radius %d.\n",
           c.x,c.y,c.radius);
}
```

```
Un cercle à (103,54) avec radius 7.
```

9.3 Exercice: Modifier le radius

Ajoutez une fonction `elargir(struct circle* p, int dr)` pour augmenter le radius d'un cercle par une quantité `dr`. Testez avec `dr=4`.

```
#include <stdio.h>
struct circle {
    int x ;
    int y ;
    unsigned int radius ;
};

int main() {
    struct circle c = { 100,50,7 };

    /** A vous de jouer */
```

```
    printf("Un cercle à (%d,%d) avec radius %d.\n",
           c.x,c.y,c.radius);
}
```

```
Un cercle à (100,50) avec radius 7.
```

10 Chaînes de caractères

En C, un caractère est en entier de 0 à 255. Chaque numéro correspond à une lettre:

- A = 65, B = 66, . . . , Z = 90
- a = 97, b = 98, . . . , z = 122
- 0 = 48, 1 = 49, . . . , 9 = 57

Une chaîne est une suite de caractères **terminé par le numéro 0** (pas besoin de stocker la longueur).

- "ENSTA" = 69,78,83,84,65,**0**
- "IN120" = 73,78,49,50,48,**0**

10.1 Créer une chaîne

chaîne modifiable avec `[]`:

```
char ma_chaine[] = "ENSTA";
```

chaîne **non modifiable** avec `*`:

```
char* ma_chaine = "ENSTA";
```

- `[]`: tableau (voir cours 4)
- chaînes non modifiables sont stockés dans mémoire partagé entre plusieurs instances du même programme
- dans les deux cas, `ma_chaine` est un pointeur `char*` vers `E`

10.2 Afficher une chaîne

```
#include <stdio.h>
int main() {
   char ma_chaine[] = "IN102";
   printf("%s\n", ma_chaine );
}
```

```
IN102
```

10.3 Modifier les chaînes

Rappel : une chaîne déclaré avec `char*` n'est pas modifiable si elle est initialisée avec une chaîne entre guillemets (`"..blabla..."`).

```
#include <stdio.h>
int main() {
   char* chaine = "ENSTA";
   *chaine = 'I';              // erreur : non modifiable
   printf("%s\n", chaine );
}
```

La cellule ci-dessous n'affiche rien parce que le programme se plante.

Dans le programme suivant tout va bien, parce qu'on déclare la chaîne avec `[]`:

```
#include <stdio.h>
int main() {
   char chaine[] = "ENSTA";
   chaine[0] = 'I';               // ok
   printf("%s\n", chaine );
}
```

```
INSTA
```

10.4 Opérations sur les chaînes

10.4.1 Affectation

Attention: L'instruction

```
chaine2 = chaine1;
```

ne fait pas de copie! On fait juste pointer `chaine2` vers la même adresse que `chaine1`.

Voici un exemple pour montrer que les deux pointent vers les mêmes lettres. Dans l'exemple suivant, une modification de `chaine1` change également `chaine2`.

```
#include <stdio.h>
int main() {
   char chaine1[] = "ENSTA";
   char* chaine2;
   chaine2 = chaine1;
   chaine1[0] = 'I';
   printf("%s\n", chaine1 );
   printf("%s\n", chaine2 );
}
```

```
INSTA
INSTA
```

10.4.2 Copier une chaîne

Pour faire une copie d'une chaîne, il y la commande `strcpy`, fournie par la bilbiothèque `string.h`.

```
char* strcpy (dest, source)
```

copie à partir de l'adresse `source` lettre par lettre toute la chaîne vers l'adresse `dest`. La valeur de retour est ici inutile, c'est simplement l'adresse `dest`. Attention: il faut prévoir assez de place à la destination, sinon il peut y avoir des erreurs graves.

```
#include <stdio.h>
#include <string.h>
int main() {
   char chaine1[] = "ENSTA";
   char chaine2[100];
   strcpy(chaine2,chaine1); // on fait une copie
   *chaine1 = 'I';          // une modification ne touche pas␣
↪l'original
   printf("%s\n", chaine1 );
   printf("%s\n", chaine2 );
}
```

```
INSTA
ENSTA
```

10.4.3 Comparer deux chaînes

Pour comparer deux chaines, on ne peut pas utiliser un comparaison de la forme `chaine1 == chaine2`. Cela compare les adresses de `chaine1` et `chaine2` au lieu des lettre de la chaîne.

```
#include <stdio.h>
int main() {
   char chaine1[] = "ENSTA";
   char chaine2[] = "ENSTA";
   printf("%d\n", chaine1 == chaine2 );
   printf("%p\n", chaine1 );
   printf("%p\n", chaine2 );
}
```

```
0
0x7ffeb131142c
0x7ffeb1311432
```

On compare ici les adresses au lieux des lettres et ces adresses ne sont pas les mêmes!

10.4.4 strcmp

L'instruction

```
int strcmp(chaine1,chaine2)
```

donne la **différence** entre les chaînes

- 0 si égales
- <0 si chaine1 < chaine2 dans l'ordre lexicographique
- >0 si chaine1 > chaine2

```
#include <stdio.h>
#include <string.h>
int main() {
   char chaine1[] = "ENSTA";
   char chaine2[] = "ENSTA";
   printf("%d\n", strcmp(chaine1,chaine2));
}
```

```
0
```

Ci-dessus, les chaînes sont égaux, donc `strcmp` donne `0` (pas de différence).

```
#include <stdio.h>
#include <string.h>
int main() {
   char chaine1[] = "ENSTA";
   char chaine2[] = "FNSTA";
   char chaine3[] = "GNSTA";
   printf("%d\n", strcmp(chaine1,chaine2));
   printf("%d\n", strcmp(chaine1,chaine3));
}
```

```
-1
-2
```

Ci-dessus, `strcmp` donne `-1` car la première lettre de `chaine1` qui est différente de celles de `chaine2` est `E`, ce qui dans l'alphabet est 1 place avant `F`.

Ensuite `strcmp` donne `-2` car la première lettre de `chaine1` qui est différente de celles de `chaine3` est `E`, ce qui dans l'alphabet est 2 places avant `G`.

10.4.5 Chercher une sous-chaîne dans une chaîne

```
char* strstr (botte_de_foin, aiguille)
```

si trouvé, donne le pointeur où `aiguille` commence dans `botte_de_foin`

si pas trouvé, donne 0.

```
#include <string.h>
#include <stdio.h>
int main() {
   char botte_de_foin[] = "J'adore l'ENSTA, c'est top.";
   char aiguille[] = "ENSTA";
   char* trouve = strstr(botte_de_foin,aiguille);
   printf("%p\n",trouve);
   printf("%s\n",trouve);
   trouve = strstr(botte_de_foin,"toto");
   printf("%p\n",trouve); // nil = 0
}
```

```
0x7ffd474cc68a
ENSTA, c'est top.
(nil)
```

11 Les tableaux

Un tableau est un ensemble de « cases » mémoire consécutives. Toutes les « cases » ont le même type. On accède immédiatement à une case particulière (« élément ») par indexation. Un tableau répond au besoin de stocker plusieurs données de même type et d'accéder rapidement (en temps constant) à n'importe quel élément.

11.1 Tableaux statiques en C

Par statique, on entend « dont la taille est connue à la compilation ». La taille des tableaux statiques est un nombre fixe au lieu d'une variable.

Comme les variables, les tableaux doivent être déclarés :

type-élément nom `[` *taille-constant* `];`

Voici un programme :

```
int main () {
  float tf [10] ; // Tableau de 10 flottants.
  int ti [5] ; // Tableau de 5 entiers signés.
  return (0) ;
```

```
}
```

11.2 Tableaux à longueur variable

Si la taille d'un tableau est spécifiée par une variable, on parle d'un **tableaux à longueur variable**. Ceci est permis seulement depuis le standard C99, introduit en 1999. Attention: ça ne veut pas dire qu'on peut changer la taille du tableau après l'avoir défini.

Déclaration d'un tableau à longueur variable :

type-élément nom `[` *variable-entier* `];`

Voici un programme :

```
int main () {
  int n =17;
  float tf [n] ; // Tableau de 17 flottants.
  int ti [n+1] ; // Tableau de 18 entiers signés.
  return (0) ;
}
```

11.3 Accès à un tableau

L'élément d'indice `i` d'un tableau `t` est dénoté par `t[i]`. Les indices de tableaux (« numéros de cases ») commencent à 0! Donc un tableau de taille `n` a des «numéros de cases » de 0 à $n-1$.

```
#include <stdio.h>
int main() {
    int T[3];
    T[0]=17;
    printf("%d\n",T[0]);
}
```

```
17
```

11.4 Tableaux et arithmétique des pointeurs

Si on declare un tableau `T` avec `int T[3]` la variable `T` est un pointeur qui pointe vers la première case du tableau. On peut accéder au premier élément du tableau avec `*T`, comme avec tout autre pointeur.

```
#include <stdio.h>
int main() {
    int T[3];
    T[0]=17;
    printf("%d\n",*T); // on peut utiliser T comme pointeur
}
```

```
17
```

Pour accéder à l'élément d'index `i`, on peut calculer son adresse avec la formule:

adresse de `T[i]` = adresse stocké dans `T` + `i`*(taille des éléments de `T`)

Dans C, ceci s'écrit simplement avec

`adresse = T+i`

car le compilateur replace automatiquement le `+i` avec

`i`*(taille des éléments de `T`).

Ce calcul s'appele **l'arithmétique des pointeurs**.

```
#include <stdio.h>
int main() {
    int T[3];
    int* adresse = T+2; // calcul de l'adresse de T[2]
    printf("%p\n",&T[2]); // l'adresse de T[2]
    printf("%p\n",adresse); // l'adresse calculé est la même
}
```

```
0x7ffc0aa89244
0x7ffc0aa89244
```

On peut accéder à un élément du tableau en utilisant un pointeur dont l'adresse était calculé.

```
#include <stdio.h>
int main() {
    int T[3];
```

```
    T[2]=3;
    int* adresse = T+2; // calcul de l'adresse de T[2]
    printf("%d\n",T[2]);
    printf("%d\n",*adresse); // accès à T[2] via le pointeur
}
```

```
3
3
```

Voici un exemple pour illustrer que les cases du tableau se suivent. On peut le voir en affichant leur adresses:

```
#include <stdio.h>
int main() {
    int T[3];
    T[0]=17;
    T[1]=31;
    T[2]=22;
    int* adr = T+2; // calcul de l'adresse de T[2]
    printf("%p\n",&T[0]); // l'adresse de T[0]
    printf("%p\n",&T[1]); // l'adresse de T[1]
    printf("%p\n",&T[2]); // l'adresse de T[2]
    printf("%p\n",adr);   // le contenu du pointer adr
    printf("%d\n",*adr);  // accès à T[2] via le pointeur
    printf("%p\n",&adr);  // l'adresse où est stocké le pointer adr
                          // (peut être avant ou après T)
}
```

```
0x7fff30ea283c
0x7fff30ea2840
0x7fff30ea2844
0x7fff30ea2844
22
0x7fff30ea2830
```

variable	adresse	contenu
adr	1036	1032
T[2]	1032	22
T[1]	1028	31
T[0]	1024	17

11.5 Débordement

Pour être le plus rapide possible, C ne véfifie pas si l'indice dépasse la taille du tableau. Si on accède au tableau avec un indice trop grand (ou négatif), le programme peut s'arrêter brutalement quand on accède à une zone de mémoire interdite ("segmentation fault"). Mais tant qu'on reste à l'intérieur de la zone mémoire du programme, on n'a pas d'erreur. Il se peut alors qu'un bug du programme reste indétecté.

```
#include <stdio.h>
int main() {
    int T[3];
    T[0]=5;
    T[1]=11;
    T[2]=17;
    printf("%d\n",T[2]);
    printf("%d\n",T[3]); // pas d'erreur, mais résultat aléatoire
    printf("%d\n",T[-1]); // pas d'erreur, mais résultat aléatoire
}
```

```
17
-1724943872
1051588000
```

Si on dépasse un tableau, on peut (sans se rendre compte) accéder à d'autres variables du même programme.

```
#include <stdio.h>
int main() {
    int T[2];
    T[0]=5;
    T[1]=11;
```

```
    int X[2];
    X[0]=23;
    X[1]=47;
    printf("%d\n",T[1]); // le dernier élément de T
    printf("%d\n",T[2]); // en dépassant T, on accède X, qui suit
    printf("%d\n",T[3]); // en dépassant T, on accède X, qui suit
    printf("%d\n",X[0]);
    printf("%d\n",X[1]);
}
```

```
11
23
47
23
47
```

Pour accéder à `T[2]`, le compilateur calcule l'adresse avec la formule `T+2 * sizeof(int)`, puisque T est un tableau de `int`. Typiquement, `sizeof(int)` vaut 4. Dans l'exemple ci-dessous, c'est l'adresse 1024+2*4=1032, ci qui est l'adresse de `X[0]`.

variable	adresse	contenu
X[1]	1036	47
X[0]	1032	23
T[1]	1028	11
T[0]	1024	5

11.6 Tableaux et fonctions

Pour passer un tableau `T` à une fonction il y a deux façons équivalents : comme un tableau, suivi par `[]`, ou comme un pointeur, précédé par `*`. Par exemple :

```
void afficher(int T[], int n)

void afficher(int* T, int n)
```

Dans les deux cas, le corps de la fonction est le même. Voici un exemple pour afficher un tableau d'entiers de longueur `n`:

```
#include <stdio.h>
void afficher(int T[], int n) {
    for (int i = 0; i<n; ++i) {
        printf("%d\n",T[i]);
    }
}
int main() {
    int T[3];
    T[0]=5;
    T[1]=11;
    T[2]=17;
    afficher(T,3);
}
```

```
5
11
17
```

Si on passe `T` comme pointeur, ça ne change rien :

```
#include <stdio.h>
void afficher(int* T, int n) {
    for (int i = 0; i<n; ++i) {
        printf("%d\n",T[i]);
    }
}
int main() {
    int T[3];
    T[0]=5;
    T[1]=11;
    T[2]=17;
    afficher(T,3);
}
```

```
5
11
17
```

11.7 Tableaux de struct

Un tableau de struct se déclare de façon analogue aux tableau des types de base.

```
#include <stdio.h>
struct point {
    int x;
    int y;
};
int main(){
    struct point T[3];
    T[0].x=1;
    T[0].y=2;
    printf("%d,%d\n",T[0].x,T[0].y);
}
```

```
1,2
```

Voici un exemple où on passe un tableau de struct à une fonction :

```
#include <stdio.h>
struct point {
    int x,y;
};
void afficher_point(struct point P) {
    printf("(%d,%d)",P.x,P.y);
}
void afficher_tableau(struct point T[], int n) {
    for (int i=0;i<n;++i) {
        afficher_point(T[i]);
    }
}
int main(){
    struct point T[3];
    T[0].x=1;
    T[0].y=2;
    T[1].x=3;
    T[1].y=4;
    T[2].x=5;
    T[2].y=6;
    afficher_tableau(T,3);
}
```

```
(1,2)(3,4)(5,6)
```

12 Passage d'arguments par la ligne de commande

Vous avez déjà vu comment on lance votre programme dans un terminal en utilisant la ligne de commande:

```
> ./monprogramme
```

Dans cette leçon, on regardera comment passer des arguments par la ligne de commande. Lorsqu'on lance on programme dans un terminal, le shell (gestionnaire du terminal) identifie les arguments passés selon des règles fixes:

```
> ./monprogramme argument1 argument2 argument3
```

Notemment, les arguments sont séparés par des espaces. Pour donner un argument qui comporte des espaces, on peut le mettre entre guillemets doubles (les guillemets seront enlevés par le shell):

```
> ./monprogramme "un long argument1" argument2
```

12.1 Main avec argv et argc

Pour utiliser les arguments passés en ligne de commande en C, il faut déclarer le `main` du programme dans la forme

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

La variable `argc` contient le **nombre d'arguments passés**, mais attention : le nom du programme compte aussi comme argument.

La variable `argv` est un tableau (`[]`) de `char*`. Rappelez-vous la l'utilisation qu'on avait fait de `char*`; c'étaut pour stocker des chaînes de caractères. Alors, `argv` est un tableau de chaînes.

Si on lance le programme avec

```
> ./monprogramme "un long argument1" argument2
```

on a les valeurs suivantes

- `argc` = 3 (on compte `monprogramme`)
- `argv[0]` = `monprogramme`
- `argv[1]` = `un long argument1` (le shell a enlevé les guillemets)
- `argv[2]` = `argument2`

12.2 Le shell dans un Jupyter Notebook

Dans un Jupyter Notebook, il n'y a pas de terminal et donc pas de ligne de commande. Par contre, un peut passer une commande en utilisant le shell: il suffit de commencer la commande avec un point d'exclamation (`!`). Pour afficher le contenu du dossier courant avec `ls`, on utilise:

`!ls`

```
!ls
```

```
'IN102-00 Avant de commencer.ipynb'
'IN102-01 Premiers Pas.ipynb'
'IN102-02 Variables et Operations.ipynb'
'IN102-03 Branchements.ipynb'
'IN102-04 Pointeurs.ipynb'
'IN102-05 Fonctions.ipynb'
'IN102-06 Fonctions (suite) - Entrees - Struct.ipynb'
'IN102-07 Chaines.ipynb'
'IN102-08 Tableaux.ipynb'
'IN102-09 argv.ipynb'
'IN102-10 Chaines - Malloc.ipynb'
'IN102-11 Enum - Macro.ipynb'
'IN102-12 Variables (suite) et bugs.ipynb'
 IN102-all-Copy1.ipynb
 IN102-all.ipynb
 __pycache__
```

Afin de lancer un programme avec le shell dans un Jupyter Notebook, il faut stocker le code dans un fichier `.c` et le compiler avec `gcc`. Pour stocker, on utilse une cellule avec le mot magique

```
%%writefile monprogramme.c
#include <stdio.h>
... et ensuite le code ...
```

et pour compiler avec `gcc`, c'est

```
!gcc -Wall -Wfatal-error monprogramme.c
```

```
%%writefile monprogramme.c
#include <stdio.h>
int main() {
    printf("Bonjour!\n");
}
```

```
Writing monprogramme.c
```

```
!gcc -Wall -Wfatal-errors monprogramme.c
```

```
!./a.out
```

```
Bonjour!
```

12.3 Exemple: Afficher les arguments

On ecrit un programme pour afficher tous les arguments passés par la ligne de commande:

```
%%writefile monprogramme.c
#include <stdio.h>
int main(int argc, char *argv[]) {
    for (int i = 0; i<argc; ++i) {
        printf("Argument %d: %s\n",i,argv[i]);
    }
}
```

```
Overwriting monprogramme.c
```

```
!gcc -Wall -Wfatal-errors monprogramme.c
```

```
!./a.out
```

```
Argument 0: ./a.out
```

```
!./a.out "un argument long"
```

```
Argument 0: ./a.out
Argument 1: un argument long
```

```
!./a.out 1 23 4 -7 "8 9" '10 11'
```

```
Argument 0: ./a.out
Argument 1: 1
Argument 2: 23
Argument 3: 4
Argument 4: -7
Argument 5: 8 9
Argument 6: 10 11
```

12.4 Passer des nombres en argument

Les arguments passés par la ligne de commande sont forcement des chaînes de caractères. Pour passer un nombre, il faut convertir la chaîne en nombre. Pour cela, il existent les fonctions suivantes en C (requièrent `#include <stdlib.h>`):

- `int atoi(char* c)` : produit un entier de type `int`
- `double atof(char* c)` : produit un flottant de type `double`

Voici un exemple qui prend en argument deux entiers et affiche leur somme:

```
%%writefile monprogramme.c
#include <stdio.h>
#include <stdlib.h> // necessaire pour atoi
int main(int argc, char *argv[]) {
    int x = atoi(argv[1]); // premier argument
    int y = atoi(argv[2]); // deuxième argument
    printf("%d + %d = %d",x,y,x+y);
}
```

```
Overwriting monprogramme.c
```

```
!gcc -Wall -Wfatal-errors monprogramme.c
```

```
!./a.out 3 4
```

```
3 + 4 = 7
```

12.5 Exercice: Quotient de deux flottants

Ecrivez un programme qui calcule le quotient de deux nombres flottants.

```
%%writefile monprogramme.c
#include <stdio.h>
int main(int argc, char *argv[]) {
    // à compléter
}
```

```
Overwriting monprogramme.c
```

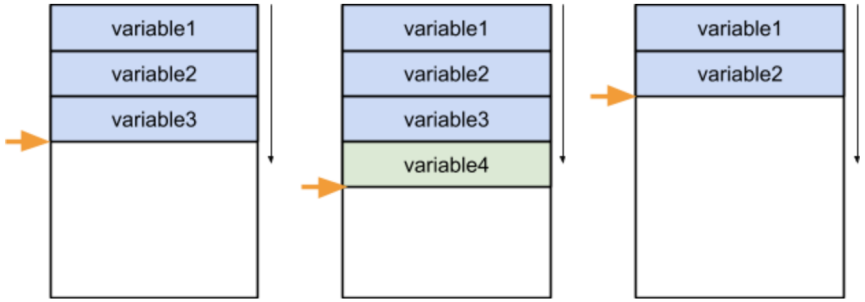
```
!gcc -Wall -Wfatal-errors monprogramme.c
```

```
!./a.out 3.14 2
```

13 La Pile (Stack)

Les variables locales sont stockés dans une zone mémoire appelée *la pile* (*stack*).

Sa fin est indiqué par le pointeur de pile.



- mémoire est libre au dessus de pointeur de pile, occupé en-dessous
- simplifie la gestion de mémoire
- rapide
- taille limitée par l'OS: *stack overflow error*

On peut regarder la taille de la pile avec la commande shell: `ulimit -a`

Cela affiche:

```
stack size              (kbytes, -s) 8192
```

La pile peut stocker 8192 kilo-octets, donc 8192*1024 octets. Si on essaie de faire un tableau statique de cette taille, le programme se plante, parce qu'il n'y aura pas assez de place (la pile est déjà un peu rempli avec quelques d'autres données).

```
#include <stdio.h>
int main() {
    char grandtableau[1024*8192]; // aussi grand que la pile
    // le programme s'arrête ici parce que le tableau est trop␣
 ↪grand
    printf("%p",grandtableau);
}
```

En diminuant un peu la taille du tableau, ça passe:

```
#include <stdio.h>
int main() {
    char grandtableau[1024*8172]; // plus petit que la pile
    printf("%p",grandtableau);
}
```

```
0x7ffd32f097d0
```

14 Le Tas (Heap)

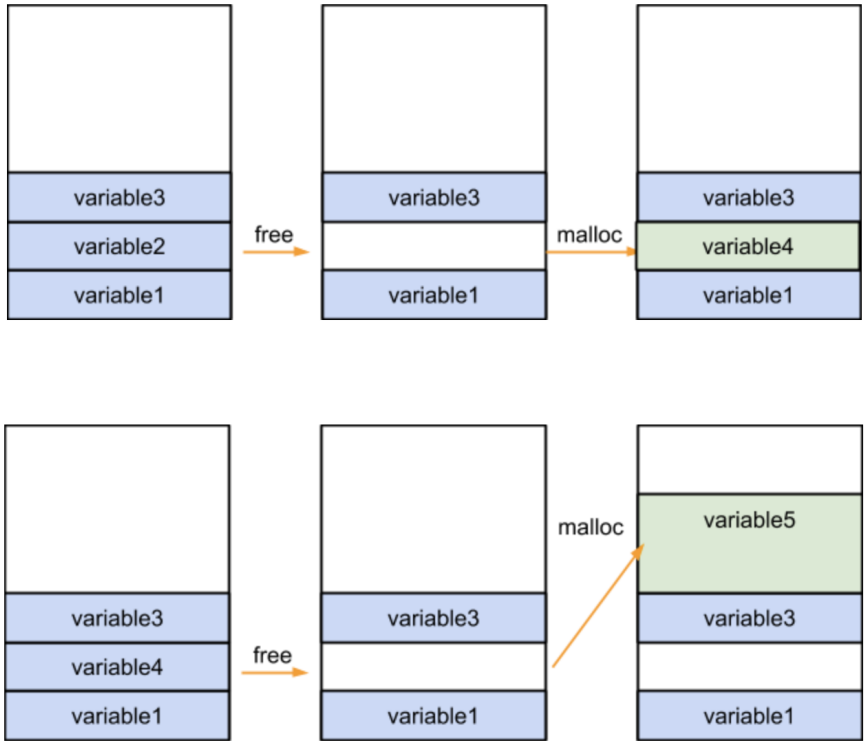
Des zones de mémoire arbitrairement grandes peuvent être réservés sur *le tas* (heap).

réservation "manuelle" avec

- `malloc` : réserver X octets et obtenir l'adresse d'une zone
- `free` : libérer la zone

14.1 Principe malloc-free

1. Réserver la mémoire et obtenir le pointeur : `type* mon_pointeur = malloc(taille);`
2. Travailler avec mon_pointeur…
3. A la fin, libérer la mémoire : `free(mon_pointeur);`



- il peut y avoir des "trous" de mémoire libre
- liste de blocs libres gérée par malloc/free
- gestion plus compliqué, plus lente

14.2 Malloc

`void* malloc(size_t nombre_d_octets)` - réserve une zone de `nombre_d_octets` octets dans le tas (plus un peu de place pour une en-tête) - stocke le nombre d'octets réservés dans une en-tête de la zone - retourne un pointeur vers le premier octet réservé - si nécessaire, le tas du processus est agrandi; **si impossible retourne 0**

`size_t` : type entier non-signé assez grand

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char* grandtableau = malloc(1024*8182);
    grandtableau[0] = 13;
    printf("%d\n",grandtableau[0]);
    printf("%p\n",grandtableau);
}
```

```
13
0x7f436b53e010
```

14.3 Free

La fonction

```
void free(void* ptr)
```

libère la zone associé avec `ptr`.

Si accès à une adresse libérée (ou autrement interdite): **segmentation fault**

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    char* grandtableau = malloc(1024*8182);
    grandtableau[0] = 13;
    printf("%p\n",grandtableau);
    printf("%d\n",grandtableau[0]);
    free(grandtableau);
    printf("%p",grandtableau);
    printf("%d\n",grandtableau[0]);
}
```

Pour réutiliser la mémoire après `free`: de nouveau un `malloc`

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int* grandtableau = malloc(sizeof(int)*1024);
    grandtableau[0] = 13;
    printf("%p\n",grandtableau);
    printf("%d\n",grandtableau[0]);
    free(grandtableau);
    int* autretableau = malloc(sizeof(int)*2048);
    printf("%p\n",autretableau);
    printf("%d\n",autretableau[0]);
    free(autretableau);
}
```

```
0x55e25705b260
13
0x55e25705d280
0
```

14.4 Exercice: Fonction de cryptage

Ecrire une fonction `crypter` qui prend en argument une chaîne et une clé `c`, et qui donne en valeur de retour la chaîne crypté (sans détruire l'original). Les caractères sont cryptés en additionnant `c`.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* crypter(char* chaine,int cle){
    // 1. réserver la mémoire pour la chaine cryptée
    // 2. pour chaque lettre x dans chaine,
    //    écrire x+cle dans la chaine cryptée
    // 3. retourner le pointeur vers la chaine cryptée
}
int main() {
    char* orig = "ENSTA";
    char* cryp = crypter(orig,3);
    printf("%s\n",cryp);
    free(cryp); // libérer la mémoire!
}
```

```
munmap_chunk(): invalid pointer
```

Solution:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* crypter(char* chaine,int cle){
    int longueur = strlen(chaine);
    char* chaine_cryptee = malloc(sizeof(char)*longueur);
    for (int i=0; chaine[i]!=0; ++i) {
        chaine_cryptee[i] = chaine[i]+cle;
    }
    return chaine_cryptee;
}
int main() {
    char* orig = "ENSTA";
    char* cryp = crypter(orig,3);
    printf("%s\n",cryp);
    free(cryp); // libérer la mémoire!
}
```

```
HQVWD
```

Utiliser la même fonction pour décrypter:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
char* crypter(char* chaine,int cle){
    int longueur = strlen(chaine);
    char* chaine_cryptee = malloc(sizeof(char)*longueur);
    for (int i=0; chaine[i]!=0; ++i) {
        chaine_cryptee[i] = chaine[i]+cle;
    }
    return chaine_cryptee;
}
int main() {
    char* orig = "ENSTA";
    char* cryp = crypter(orig,3);
    printf("%s\n",cryp);
    char* decryp = crypter(cryp,-3);
    printf("%s\n",decryp);
    free(cryp); // libérer la mémoire!
    free(decryp); // libérer la mémoire!
}
```

```
HQVWD
ENSTA
```

15 Types enumérés

Les types enumérés représentent des valeurs choisies parmis un (petit) ensemble, par exemple :

- nord, est, sud, ouest,
- coeur, carreau, trèfle, pique,
- admis, refusé, indéterminé.

Afin de représenter ces valeurs dans un programme, il faut associer chaque valeur à un nombre. On pourrait choisir des valeurs entières :

- nord = 0, est = 1, sud = 2, ouest = 3,
- coeur = 0, carreau = 1, trèfle = 2, pique = 3,
- admis = 0, refusé = 1, indéterminé = 2.

Ensuite on pourrait les traiter comme des entiers dans le programme:

```
int d = 2; // on commence avec le sud
...
if (d == 3) { // vers l'ouest
   printf("Ce n'est pas par là.");
}
```

En revanche, il est pénible et sujette à erreurs de se souvenir des différentes nombres, surtout dans un grand programme qui est écrit par plusieurs personnes. En C, peut demander au compilateur de faire ce travail pour nous, en **déclarant un type** `enum`:

`enum` nom-type {nom-valeur1 , nom-valeur2 ,... };

Par défaut, le compilateur va associer nom-valeur1 avec 0, nom-valeur2 avec 1, etc. Le code devient beaucoup plus lisible et plus facile à modifier :

```
enum direction { NORD, EST, SUD, OUEST };

enum direction d = SUD; // on commence avec le sud
...
if (d == OUEST) { // vers l'ouest
   printf("Ce n'est pas par là.");
}
```

Voici un petit exemple:

```
#include <stdio.h>

enum direction { NORD, EST, SUD, OUEST };

int main(void) {
  enum direction d = SUD; // on commence avec le sud

  if (d == OUEST) { // vers l'ouest
    printf("Ce n'est pas par là.\n");
  } else if (d == OUEST) { // vers l'ouest
    printf("Ce n'est pas par là.\n");
  } else {
    printf("Par ici c'est bon.\n");
  }

  printf("entier associé à NORD: %d\n",NORD);
  printf("entier associé à EST: %d\n",EST);
  printf("entier associé à SUD: %d\n",SUD);
  printf("entier associé à OUEST: %d\n",OUEST);
  return 0;
}
```

```
Par ici c'est bon.
entier associé à NORD: 0
entier associé à EST: 1
entier associé à SUD: 2
entier associé à OUEST: 3
```

On peut utiliser les types `enum` commes les autres types, par exemple dans un tableau ou dans une fonction:

```
#include <stdio.h>

enum direction { NORD, EST, SUD, OUEST };

enum direction opposee(enum direction d) {
  if (d == NORD) {
    return SUD;
  } else if (d == EST) {
    return EST;
  } else if (d == SUD) {
    return NORD;
  } else {
    return OUEST;
  }
}

int main(void) {
  enum direction d1 = SUD; // on commence avec le sud

  // changer de sens
  enum direction d2 = opposee(d1);
  printf("l'opposée de SUD: %d\n",d2);

  printf("entier associé à NORD: %d\n",NORD);
  printf("entier associé à EST: %d\n",EST);
  printf("entier associé à SUD: %d\n",SUD);
  printf("entier associé à OUEST: %d\n",OUEST);
  return 0;
}
```

```
l'opposée de SUD: 0
entier associé à NORD: 0
entier associé à EST: 1
entier associé à SUD: 2
entier associé à OUEST: 3
```

Pour afficher un type `enum` de façon plus lisible, on peut les associer avec un tableau

de chaînes de caractères :

```
#include <stdio.h>

enum direction { NORD, EST, SUD, OUEST };

char* direction_chaine[] = {
  "Nord",
  "Est",
  "Sud",
  "Ouest"
};

enum direction opposee(enum direction d) {
  if (d == NORD) {
    return SUD;
  } else if (d == EST) {
    return EST;
  } else if (d == SUD) {
    return NORD;
  } else {
    return OUEST;
  }
}

int main(void) {
  enum direction d1 = SUD; // on commence avec le sud
  // changer de sens
  enum direction d2 = opposee(d1);
  printf("l'opposée de %s est %s\n",
       direction_chaine[d1],
       direction_chaine[d2]
    );

  return 0;
}
```

```
l'opposée de Sud est Nord
```

16 Constantes littérales

Si on utilise un nombre constant partout dans le programme, il est préférable de la remplacer par un **macro** qui l'associe à un nom.

Un exemple d'un programme qui utilise un paramètre partout qui pour l'instant vaut 10:

```
#include <stdio.h>
void ligne() {
    for (int i=0;i<10;++i) {
        printf("*");
    }
}
int main() {
    for (int i=0;i<10;++i) {
        ligne();
        printf("\n");
    }
}
```

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

Si on veut remplacer 10 par 20, il est facile de faire une erreur. Mieux utiliser une constante globale:

```
#include <stdio.h>

#define DIMENSION 10

void ligne() {
```

```
    for (int i=0;i<DIMENSION;++i) {
        printf("*");
    }
}
int main() {
    for (int i=0;i<DIMENSION;++i) {
        ligne();
        printf("\n");
    }
}
```

```
**********
**********
**********
**********
**********
**********
**********
**********
**********
**********
```

Les macros sont remplacés textuellement avant compilation par le **préprocesseur C**.

17 Variables (suite)

17.1 Initialisation des variables

Qu'est-ce que se passe si on n'itialise pas une variable? Les instructions

`int i; printf("%d",i);`

peuvent afficer la valeur 0, mais aussi tout autre valeur de `int` (`-98765,1234567,...`). C réserve la place dans la mémoire pour `i`, mais ne modifie pas le contenu des octets!

Compiler avec `-Wall` permet d'attraper des fautes d'initialisation :

```
$ gcc -Wall test1.c
test1.c: In function ‘main’:
test1.c:3:4: warning: ‘i’ is used uninitialized in this function
 [-Wuninitialized]
    printf("%d",i);
    ^~~~~~~~~~~~~~
```

Mieux initialiser tout de suite : `int i = 0;`

```
#include <stdio.h>
int main() {
   int i;           // declaration sans initialisation
   printf("%d",i);
}
```

```
0
```

17.2 Limites des types numériques

Chaque variable a une certain taille, donnée en nombre d'octets. Cela impose forcément des limites sur la plage de valeurs.

En conséquence, chaque type de variable a une valeur maximale et une valeur minimale qui peuvent être représentées.

Les limites sont disponibles dans `limits.h`:

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

int main() {
   printf("The number of bits in a byte = %d\n", CHAR_BIT);
   printf("The minimum value of INT = %d\n", INT_MIN);
   printf("The maximum value of INT = %d\n", INT_MAX);

   return(0);
}
```

```
The number of bits in a byte = 8
The minimum value of INT = -2147483648
The maximum value of INT = 2147483647
```

17.3 Débordement

Quand on dépasse la valeur maximale ou la valeur minimale d'une variable on parle de **débordement**. Le débordement peut entraîner des bugs difficiles à détecter. Pour un nombre nonsigné, le CPU calcule alors la valeur modulo (maximum + 1).

Exemples avec type nonsigné:

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

int main() {
   unsigned char c = 255;
   printf("%u\n", c);
   c = c + 1;              // 255 + 1 modulo 256 = 0
   printf("%u\n", c);
}
```

```
255
0
```

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

int main() {
   unsigned char c = 0;
   printf("%u\n", c);
   c = c - 1;              // 0 - 1 modulo 256 = 255
   printf("%u\n", c);
}
```

```
0
255
```

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

int main() {
   unsigned char c = 26;
   printf("%u\n", c);
   c = 10*c;              // 26*10 modulo 256 = 4
   printf("%u\n", c);
}
```

```
26
4
```

Si le type est signé, le débordement passe de la valeur maximale à la valeur minimale et vice versa:

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

int main() {
   signed char c = 127;
   printf("%d\n", c);
   c = c + 1;
   printf("%d\n", c);
}
```

```
127
-128
```

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

int main() {
   int i = 2147483647;
   printf("%d\n", i);
   i = i + 1;
   printf("%d\n", i);
}
```

```
2147483647
-2147483648
```

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

int main() {
   int i = -2147483648;
   printf("%d\n", i);
   i = i - 1;
   printf("%d\n", i);
}
```

```
-2147483648
2147483647
```

17.4 Bug méchant

C'est quoi le problême avec le programmme suivant ?

```
#include <stdio.h>
int main() {
   unsigned int i;
   for (i = 3; i >= 0; i=i-1) {
      printf("i = %u\n", i);
  }
}
```

```
i = 3
i = 2
i = 1
i = 0
i = 4294967295
i = 4294967294
...
```

La boucle était censé compter de 3 à 0 dans l'ordre décroissante, mais elle tourne sans fin! Puisque `i` est une variable nonsigné, le résultat de `i-1` quand `i` vaut `0` est le nombre `4294967295`, qui est le plus grand nombre représenté par le type `unsigned int`. Du coup, le test `i >= 0` est toujours vrai et la boucle ne s'arrête jamais.

Version plus méchante :

```
#include <stdio.h>
int main() {
   unsigned int i;
   for (i = 3; i >= 0; i=i-1) {
      printf("i = %d\n", i);
  }
}
```

```
i = 3
i = 2
i = 1
i = 0
i = -1
i = -2
...
```

Ci-dessus, la boucle est exactement la même. Par contre, l'affichage est trompeur : On affiche avec le format `%d` qui interprête `i` comme un nombre signé. Au lieu d'afficher le nombre `4294967295`, il affiche alors `-1`. Du coup on ne voit pas pourquoi la boucle ne s'arrête pas. Le seul moyen de trouver le bug c'est de regarder le *type* de la variable `i`.

17.5 Conversion implicite

Si on affecte une variable à une variable d'un autre type, C fait une conversion automatique. Si l'autre type ne peut pas représenter toutes les valeurs du type d'origine, ceci peut entraîner... (suspens)... des bugs.

Exemple:

```
unsigned char c;
int i = 123;
c = i;  // ok pour entiers entre 0 et 255
```

```
#include <stdio.h>
int main() {
   unsigned char c;
   int i = 123;
   c = i; // ok pour entiers entre 0 et 255
   printf("%d",c);
}
```

```
123
```

```
#include <stdio.h>
int main() {
   unsigned char c;
   int i = 256+123;
   c = i; // ok pour entiers entre 0 et 255
   printf("%d",c);
}
```

```
123
```

En allant d'un flottant vers un entier, on perd la fraction:

```
#include <stdio.h>
int main() {
   int i;
   double d = 3.4142;
   i = d; // ok entre -2147483648 et 2147483647
   printf("%d",i);
}
```

```
3
```

```
#include <stdio.h>
int main() {
   int i;
   double d = 3.99999;
   i = d; // ok entre -2147483648 et 2147483647
   printf("%d",i);
}
```

```
3
```

Si on affect un nombre plus grand que les limites du type, on se retrouve avec une sitation de dépassement :

```
#include <stdio.h>
int main() {
   int i;
   double d = 121474836480;
   i = d; // ok entre -2147483648 et 2147483647
   printf("%d",i);
}
```

```
-2147483648
```

Conversion correcte s'il n'y a pas de perte d'information: - char -> int : 1 octet tient dans 4 octets - int -> double : 4 octets = 31 bits (+signe) tiennent dans les 52 bit de la mantisse

Attention aux autres cas: le résultat peut être complètement faux!

17.5.1 Exemple de conversion char-int

`printf("%d",i)` : `%d` s'applique à un entier de type `int`.

Si on l'appelle avec un `char`, celui est automatiquement converti en `int`.

```
#include <stdio.h>
int main() {
    char c = 127;
    printf("%d",c);
}
```

```
127
```

17.5.2 Exemple de conversion float-double

`printf("%f",d)` : `%f` s'applique à un flottant de type `double`.

Si on l'appelle avec un `float`, celui est automatiquement converti en `double`.

```
#include <stdio.h>
int main() {
    float f = 1.23;
    printf("%f",f);
}
```

```
1.230000
```

17.5.3 Exemple de conversion double-int

```
#include <stdio.h>
void afficher(int x) {
    printf("%d",x);
}
int main() {
    double f = 2.34;
    afficher(f);
}
```

```
2
```

17.5.4 Exemples de conversion incorrecte

```
#include <stdio.h>
int main() {
    int i = -1234;
    char c = i;
    printf("%d -> %d",i,c);
}
```

```
-1234 -> 46
```

```
#include <stdio.h>
int main() {
    double z = 1e10;
    int i = z;
    printf("%g -> %d",z,i);
}
```

```
1e+10 -> -2147483648
```

18 Débogage avec printf

Pour chercher des bugs il y a deux techniques principales :

- afficher des informations supplémentaires avec `printf`,
- utilser un outil de débogage comme `gdb`.

Avec `printf`, on peut facilement attraper les erreurs les plus courantes.

Attention: L'affichage n'a lieu qu'après un retour à la ligne. **Toujours ajouter `\n` si `printf` est pour déboger.**

Voici quelques exemples.

18.1 Une fonction ne donne pas la valeur attendue

Afficher les **entrées** et les **sorties** de la fonction!

Voici un bug:

```
#include <stdio.h>
double div(int x, int y) {
    double z = x/y;
    return z;
}
int main() {
    double a = div(1,3);
    printf("%g\n",a);       // on veut 0.3333 mais ça donne 0
}
```

```
0
```

On ajoute des printf pour afficher entrées et sorties:

```
#include <stdio.h>
double div(int x, int y) {
    printf("%g,%g",x,y);
    double z = x/y;
    printf("%g",z);
    return z;
}
int main() {
    double a = div(1,3);
    printf("%g\n",a);       // on veut 0.3333 mais ça donne 0
}
```

```
6.9437e-310,6.95305e-31000
```

Ici, l'avertissement du compilateur nous pointe vers la source du problème : `x`et `y`devraient être déclarés comme `int`!

18.2 Boucle `while` n'arrête pas comme prévu

Afficher tout les parties de la condition de `while`. S'il y a un compteur de boucle, l'afficher également.

Le programme suivant ne calcule pas le bon résultat:

```
#include <stdio.h>
int main() {
    int iter = 0;
    int iter_max = 10;
    double x = 2;
    while (x*x<4 && iter < iter_max) {
        x = -0.5*x - 1;
        iter = iter + 1;
    }
    printf("%g ",x);
}
```

```
2
```

On ajoute un printf **avant** et **à la fin** de la boucle `while` pour afficher les valeurs des variables et les résultats des tests:

```
#include <stdio.h>
int main() {
    int iter = 0;
    int iter_max = 10;
    double x = 2;
    printf("x*x: %g, test1: %d, ",x*x,x*x<4);
    printf("iter: %d, test2: %d\n",iter,iter<iter_max);
    while (x*x<4 && iter < iter_max) {
        x = -0.5*x - 1;
        iter = iter + 1;
        printf("x*x: %g, test1: %d, ",x*x,x*x<4);
        printf("iter: %d, test2: %d\n",iter,iter<iter_max);
    }
    printf("%g ",x);
}
```

```
x*x: 4, test1: 0, iter: 0, test2: 1
2
```

Grace au printf on se rend compte qu'on n'entre jamais dans la boucle car le test `x*x<4` échoue. La solution était d'utiliser `x*x<=4`.

```
#include <stdio.h>
int main() {
    int iter = 0;
    int iter_max = 10;
    double x = 2;
    printf("x*x: %g, test1: %d, ",x*x,x*x<4);
    printf("iter: %d, test2: %d\n",iter,iter<iter_max);
    while (x*x<=4 && iter < iter_max) {
        x = -0.5*x - 1;
        iter = iter + 1;
        printf("x*x: %g, test1: %d, ",x*x,x*x<4);
        printf("iter: %d, test2: %d\n",iter,iter<iter_max);
    }
    printf("%g ",x);
}
```

```
x*x: 4, test1: 0, iter: 0, test2: 1
x*x: 4, test1: 0, iter: 1, test2: 1
x*x: 0, test1: 1, iter: 2, test2: 1
x*x: 1, test1: 1, iter: 3, test2: 1
x*x: 0.25, test1: 1, iter: 4, test2: 1
x*x: 0.5625, test1: 1, iter: 5, test2: 1
x*x: 0.390625, test1: 1, iter: 6, test2: 1
x*x: 0.472656, test1: 1, iter: 7, test2: 1
x*x: 0.430664, test1: 1, iter: 8, test2: 1
x*x: 0.451416, test1: 1, iter: 9, test2: 1
x*x: 0.440979, test1: 1, iter: 10, test2: 0
-0.664062
```

18.3 Le programme s'arrête subitement

Parfois le programme se plante subitement. Voici quelques causes potentielles :

- division par zero: `x/y` avec `y=0`
- dépassement d'un tableau `T[i]` avec `i` plus grand que la taille de `T` le permet
- utilisation d'un mauvais pointeur: `*p`où `p` ne pointe pas vers la bonne adresse
- appel de `free(p)` sur un mauvais pointeur (oublié `p=malloc(...)` ou déjà fait `free` avant)

On peut localiser l'instruction responsable pour l'erreur en l'imbriquant entre deux `printf` (principe de bisection). Ajouter des `printf("test A\n")` avant et `printf("test B\n")` après l'arrêt soupçonné, puis déplacer les deux jusqu'ils imbriquent l'instruction fautive.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i = 0;
    int N = 10;
    double* T = 0;
    //              oublié: T = malloc(sizeof(double)*N);
    double x = 2;
    while (x*x<=4 && i <= N) {
        x = -0.5*x - 1;
        T[i] = x;
        i = i + 1;
    }
    printf("test A\n");         // ----> erreur
    printf("%g ",T[0]);         // instruction soupçonnée
    printf("test B\n");         // erreur <----
    free(T);
}
```

… affiche ni `test A` ni `test B` (si lancé dans un terminal; dans un Jupyter Notebook un programme qui se plante n'affiche rien du tout). L'arrêt doit alors être plus tôt que `test A`.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i = 0;
    int N = 10;
    double* T = 0;
    //              oublié: T = malloc(sizeof(double)*N);
    double x = 2;
    while (x*x<=4 && i <= N) {
        printf("test A\n");         // ----> erreur
        x = -0.5*x - 1;
        T[i] = x;
        i = i + 1;
        printf("test B\n");         // erreur <----
    }

    printf("%g ",T[0]);
    free(T);
}
```

… affiche `test A` mais pas `test B` (si lancé dans un terminal; dans un Jupyter Notebook un programme qui se plante n'affiche rien du tout). L'arrêt doit alors être plus tard que `test A`.

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i = 0;
    int N = 10;
    double* T = 0;
    //              oublié: T = malloc(sizeof(double)*N);
    double x = 2;
    while (x*x<=4 && i <= N) {
        x = -0.5*x - 1;
        printf("test A\n");         // ----> erreur
        T[i] = x;
        printf("test B\n");         // erreur <----
        i = i + 1;
    }

    printf("%g ",T[0]);
    free(T);
```

```
}
```

... affiche `test A` mais pas `test B` (si lancé dans un terminal; dans un Jupyter Notebook un programme qui se plante n'affiche rien du tout). On encerclé une seule instruction qui doit être responsable de l'arrêt. On se souvient que le tableau `T` n'a jamais été alloué (`malloc` oublié) et on corrige l'erreur...

18.4 Plusieurs opérations par ligne

Attention: Eviter de faire plusieurs opérations dans une seule ligne d'instruction. Cela rend le débogage plus difficile. Au besoin, on sépare les opérations en introduisant des variables supplémentaires, qu'on peut afficher.

Exemple: Le programme suivant ne donne pas le résultat attendu. Comme la ligne de calcul est longue, ce n'est pas évident de trouver l'erreur.

```
#include <stdio.h>

int main() {
    int x = 3;
    double z = x/9+0.5*x;
    printf("%g\n",z);     // devrait donner 1.8333
}
```

```
1.5
```

En séparant les opérations et affichant les résultats intermédiaires, on trouve rapidement la faute:

```
#include <stdio.h>

int main() {
    int x = 3;
    double temp1 = x/9;
    printf("%g\n",temp1);
    double temp2 = 0.5*x;
    printf("%g\n",temp2);
    double z = temp1+temp2;
    printf("%g\n",z);     // devrait donner 1.8333
}
```

```
0
1.5
1.5
```

L'instruction `x/9` utilise la division entière, ce qui donne la faux résultat.

18.5 Eviter les bugs

Le mieux c'est d'attrapper les bugs le plus tôt possible. Voici quelques astuces :

- Compiler avec tous les vérifications : `gcc -Wall -Werror -Wfatal-errors monpgramme.c`
- Compiler le plus souvent possible (chaque fois vous avez écrit 2-3 nouvelles lignes).
- **Lire les messages d'erreur avec attention, pour bien comprendre la source de l'erreur.**

18.6 Stubs

Si une partie de votre programme ne marche pas, il faut la désactiver pour pouvoir compiler. De plus, il faut la remplacer par une substitution qui vous permet de continuer à développer le reste du programme:

1. Déplacez les instructions fautives vers une fonction et mettez-les en commentaires.
2. Donner une valeur de retour "utile" qui vous permet de continuer comme si la fonction marchait.

Une telle fonction temporaire qui se substitue pour d'autre code s'appelle un **stub** (*bouchon* en français).

Dans l'exemple suivant, l'entrée du nombre ne marche pas:

```
#include <stdio.h>
int main() {
    int x;
    do {
        scanf("Donner un nombre: %d",&x);
```

```
        printf("Le carré de %d est %d.\n",x,x*x);
    } while (x>0);
}
```

On met l'instruction en commentaires `/* ... */` pour la désactiver, mais du coup le reste du programme ne marche plus parce que `x` n'a pas la bonne valeur:

```
#include <stdio.h>
int main() {
    int x;
    do {
        /*
        scanf("Donner un nombre: %d",&x);
        */
        printf("Le carré de %d est %d.\n",x,x*x);
    } while (x>0);
}
```

Mieux: On déplace les instructions fautives dans une fonction, qui donne une valeur de retour "utile" pour pouvoir tester le reste du programme. Ici ont choisit `0` parce que sinon le programme ne s'arrête jamais:

```
#include <stdio.h>
int entree() {
    int x;
    /* Je ne trouve pas la faute ici:
       scanf("Donner un nombre: %d",&x);
    */
    // pour continuer, je donne une valeur par défaut
    x = 0;
    return x;
}
int main() {
    int x;
    do {
        x = entree();
        printf("Le carré de %d est %d.\n",x,x*x);
    } while (x>0);
}
```

```
Le carré de 0 est 0.
```

Vous pouvez même ajouter du code pour simuler plusieurs entrées :

```
#include <stdio.h>

/*********************************************
   Code pour simuler des entrées parce que
   je n'arrive pas à faire marcher scanf.
*/
int entree_compteur = 0;
int entrees_fixes[4] = {2,7,1,0};
int entree() {
    int x;
    /* Je ne trouve pas la faute ici:
       scanf("Donner un nombre: %d",&x);
    */
    // pour continuer, je donne 4 valeurs par défaut
    x = entrees_fixes[entree_compteur];
    ++entree_compteur;
    return x;
}
/*********************************************/
int main() {
    int x;
    do {
        x = entree();
        printf("Le carré de %d est %d.\n",x,x*x);
    } while (x>0);
}
```

```
Le carré de 2 est 4.
Le carré de 7 est 49.
Le carré de 1 est 1.
Le carré de 0 est 0.
```

Si plus tard vous trouvez l'erreur, il suffit de corriger la fonction, sans toucher au reste du programme:

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

```
int entree() {
    int x;
    printf("Donner un nombre: ");
    scanf("%d",&x);
    return x;
}
int main() {
    int x;
    do {
        x = entree();
        printf("Le carré de %d est %d.\n",x,x*x);
    } while (x>0);
}
```

La fonction principale d'un **stub** est de servir d'emplacement pour du code qui reste encore à écrire. Ca marche très bien pour structurer son programme au fur et à mesure, surtout si on ajoute des commentaires qui expliquent la fonctionalité qui reste à développer:

```
#include <stdio.h>

/* STUB: Demander à l'utilisateur de taper un entier */
int entree() {
   /* à faire */
   return 3;
}

/* STUB: Faire le calcul compliqué */
double calcul(int a) {
   /* à faire */
   return 3.1415;
}

/* STUB: Afficher le résultat */
void affichage(double x) {
   /* à faire proprement */
   printf("%g\n",x);
   return;
}
```

```
int main() {
    // grace aux fonctions, le programme principal
    // est simple et lisible:
    int a = entree();
    double x = calcul(a);
    affichage(x);
}
```

```
3.1415
```