Langage C

Katia Jaffrès-Runser, Xavier Crégut

Toulouse INP - ENSEEIHT,

1ère année, Dept. Sciences du Numérique, 2019-2020.

Déroulement du cours

Ce cours est découpé en deux parties, la première ayant lieu au semestre 6, la seconde au semestre 7.

- Au semestre 6, les acquis vous permettront de suivre les TPs du *cours d'automatisme*. Le cours de C en S6 est validé par une note inclue dans l'UE Signal et Automatisme.
- Au semestre 7, les acquis vous permettront de suivre les TPs du cours de *systèmes d'exploitation*. Le cours de C en S7 est validé par une note inclue dans l'UE Architecture et Systèmes.

A chaque semestre, vous suivrez 3 séances de TPs :

- Lors des deux premières séances, vous suivrez un notebook Jupyter présentant des éléments de cours associés à un ensemble d'exercices à réaliser. Ce travail se poursuit hors séance, en autonomie. Le notebook est archivé sous SVN mais ne sera pas noté.
- A la fin du notebook Jupyter sont listés un ensemble d'exercices à rendre sur SVN. Ces rendus donneront lieu à une note.

  > **Attention : Ce travail est individuel**. Des outils de détection de recopie de code seront utilisés pour détecter la fraude.

- A la fin de la 3e séance aura lieu un QCM de 30 minutes qui validera vos acquis.

La note finale est une moyenne des deux notes (QCM et exercices rendus).

Objectifs

Ce cours, sous la forme de notebooks Jupyter et d'un ensemble d'exercices à réaliser en TP, a pour objectif de vous présenter les spécificités de la programmation en langage C. Il se base sur vos acquis du cours de Programmation Impérative en algorithmique et vous détaille les éléments du langage C nécessaires à la production d'un programme en C.

Un support de cours PDF vous est également fournit sur Moodle : Cours C.

Plan du contenu étudié au semestre 6.

Les éléments suivants de la programmation en Langage C sont présentés dans les 3 premières séances de TP au semestre 6 :

- La structure d'un programme et sa compilation
- Les constantes, types et variables
- Les entrées / sorties
- Les structures de contrôle
  - Conditionnelles
  - Boucles
- Les types énumération, enregistrement et tableaux
- Les chaînes de caractère
- Le type pointeur
- Les sous-programmes en C
  - Leur signature
  - Passage par valeur
  - Passage par adresse

Jupyter notebook

Le support de cours que vous lisez est un notebook Jupyter. Pour visualiser le notebook, lancer l'editeur web avec la commande

```
jupyter-notebok
```

et rechercher le fichier dans l'arborescence. Le fichier est édité dans votre navigateur Web par défaut. L'enregistrement est automatique (`CTRL S` pour le forcer).

Pour fermer votre fichier, il faut fermer le navigateur et terminer le processus serveur qui s'exécute dans le terminal (`CTRL C`, puis `y`).

> **Important** :
>
> - Pour faire fonctionner le kernel C de jupyter notebook, il faut, avant votre **première utilisation** de ce Notebook, lancer la commande suivante dans un `Terminal` :
> - `install_c_kernel --user`

Il se compose de cellules présentant soit :

- Des éléments de cours, au format Markdown. Ce langage est interprété pour un affichage aisé quand on clique sur la flèche `Exécuter` et que la cellule est active.
- Du code en Langage C (ou Python, ou autre..). Pour compiler et exécuter le code écrit dans la cellule active, on clique sur la flèche `Exécuter`. Si la compilation se déroule sans erreur ni avertissement, le programme est exécuté et les sorties sont affichées en bas de la cellule. Si ce n'est pas le cas, les avertissements et warnings sont affichés en bas de la cellule.

En double-cliquant sur une cellule, on peut éditer son contenu. Vous pouvez ainsi :

- Editer une cellule markdown pour y intégrer vos propres notes.
- Modifier les programmes pour répondre aux questions et exercices proposés.

Il est possible d'exporter votre travail en PDF, HTML, etc.

Le programme dans la cellule suivante s'exécute sans erreur. Vous pouvez

- le tester en l'exécutant.
- y introduire une erreur (suppression d'un point-virgule par exemple) pour observer la sortie du compilateur.

In [3]:

```c
#include <stdlib.h>
#include <stdio.h>
int main()
{
    printf("*****************************\n");
    printf("******** Langage C ***********\n");
    printf("*****************************\n");
    return EXIT_SUCCESS;
}
```

```
*****************************
******** Langage C ***********
*****************************
```

Un premier programme en Langage C

Le fichier `pgcd.c` suivant comporte un programme en Langagce C. Exécutez-le.

In [8]:

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

```c
/* Afficher le pgcd de deux entiers strictement positifs. */
int main() {
    // Déclaration et initialisation de deux entiers
    int a = 105, b = 35;

    // Déterminer le pgcd de a et b
    int na = a, nb = b;  // gain de place ! À éviter !
    while (na != nb) { // na et nb différents
        // Soustraire au plus grand le plus petit
        if (na > nb) {
            na = na - nb;
        } else  {
            nb = nb - na;
        }
    }
    int pgcd = na;      // le pgcd de a et b

    // Afficher le pgcd
    printf("Le pgcd de %d et %d est %d\n", a, b , pgcd);
    return EXIT_SUCCESS;
}
```

```
Le pgcd de 105 et 35 est 35
```

Ce programme se compose de :

- Trois commandes **pré-processeur** `#include`.
  - Toutes les commandes pré-processeur commencent par le caractère #.
  - Ces deux commandes importent des librairies (i.e. des modules).
- La fonction `int main()`, qui correspond au programme principal. Ses instructions sont définies entre accolades.

> **Règle** : L'identificateur du programme principal est forcément `main()`.

- Un ensemble d'instructions entre les accolades.

> **Règle** : Chaque instruction se termine avec un **point-virgule**.

- Un appel au sous-programme d'affichage à l'écran `printf` du module `stdio`. A l'exécution, on observe que les valeurs des variables `a`, `b` et `pgcd` sont écrites en lieu et place des `%d`, dans l'ordre de leurs appels.
- Le retour d'une constante `EXIT_SUCCESS` définie dans le module `stdlib`. Cette constante vaut 0 et indique que l'exécution s'est terminée avec succès. Il existe aussi `EXIT_FAILURE` qui indique la mauvaise terminaison du programme.

> **Règle** : L'instruction `return` arrête et indique le résultat de la fonction.

- Une boucle TantQue avec la structure de contrôle `while`. Les instruction de corps de la boucle sont définies entre accolades.
- Une conditionelle `if (condition) then {..} else {..}`
- Des déclarations de variables, des opérations d'initialisation et d'affectations.

> **Règle** : L'opérateur d'affectation est `=`.

- Des tests.

> **Règle** : L'opérateur de test d'égalité est `==`, et d'inégalité est `!=`.

Compilation et pré-processeur

La compilation en C se décompose en deux étapes successives :

1. L'exécution du pré-processeur,
2. L'exécution du compilateur C.

Les deux étapes sont réalisées par un seul appel à la suite de compilation avec la commande :

```
gcc -Wall premier_programme.c -o premier_programme
```

Les options permettent :

- `Wall` : d'afficher l'ensemble des avertissements produits par la compilation
- `-o` : de choisir le nom de l'exécutable généré.

Le pré-processeur fournit un unique fichier au compilateur, qui le transforme en un fichier binaire exécutable. Ce pré-processeur :

- Supprime les commentaires de ligne `//` ou de bloc `/*  */`.
- Interprête les commandes pré-processeur qui commencent par `#` (`#define`, `#include`, etc.)

> **Règle :** Il n'y a pas de point-virgule à la fin d'une instruction pré-processeur.

Exercice 1 -- Compilation.

[1.1] Compiler votre premier programme dans un terminal. Pour se faire, créer un répertoire `Langage_C` et y ajouter un fichier nommé `pgcd.c`. Recopier le programme de l'exemple précédent. Le compiler avec le compilateur `gcc`et l'exécuter avec la commande `./pgcd`

[1.2] Introduire une erreur dans les instructions et observer le retour du compilateur :

- Suppression d'un point-virgule en fin de ligne,
- Ajout d'un point-virgule supplémentaire en fin de ligne,
- Supprimer la déclaration de la variable `a`.
- Supprimer l'accolade de fin de bloc de la boucle `while`.

[1.3] Observer l'unique fichier généré par le pré-processeur avec l'appel à la commande `cpp -P premier_programme.c`. Quel est l'effet de la commande `#include <stdio.h>` ?

Exercice 2 -- Comprendre la macro `assert()`.

Voyons comment fonctionne la macro `assert` du langage C. Nous nous appuyons sur le programme `assert-comprendre.c`.

[2.1] Compiler et exécuter dans Jupyter Notebook. Qu'observez-vous ?

In [9]:

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

void assert_ok() {
    int n = 10;
    assert(n > 0);
    printf("(assert_ok) n = %d\n", n);
}


void assert_erreur() {
```

```
    int n = 10;
    assert(n <= 0);
    printf("(assert_erreur) n = %d\n", n);
}


int main(void) {
    assert_ok();
    assert_erreur();
    return EXIT_SUCCESS;
}
```

```
(assert_ok) n = 10
```

```
tmpa3sl91x_.out: /tmp/tmpwd_o7dqg.c:14: assert_erreur: Assertion `n <= 0' failed.
[C kernel] Executable exited with code -6
```

L'appel à `assert_ok` se déroule normalement car le paramètre effectif de `assert` s'évalue à vrai.

Au contraire, l'appel à `assert_erreur` provoque l'arrêt du programme car le paramètre effectif de assert s'évalue à faux. Un message d'erreur indique la ligne dans le fichier source contenant l'appel à `assert`.

[2.2] Compiler et exécuter le programme dans un terminal, dans le répertoire SVN fourni, avec les commandes :

```
make assert-comprendre
./assert-comprendre
```

> *Note* : La commande `make` sera présentée à la fin du cours. Elle permet d'automatiser la compilation. Elle est paramétrée par le fichier `Makefile`. Vous pouvez le consulter mais sa compréhension n'est pas l'objet de cette question.

Qu'observez-vous ?

[2.3] L'évaluation des assert peut être désactivée en définissant la macro `NDEBUG` (no debug). Par exemple, en début du fichier `assert-comprendre.c` (mais avant l'inclusion de `assert.h`), on peut ajouter la commande préprocesseur suivante qui définit NDEBUG :

```
#define NDEBUG
```

Modifier le fichier `assert-comprendre.c`, compiler et exécuter à nouveau pour constater que les assert ne sont plus vérifiés. Vous pouvez aussi le tester sur le notebook Jupyter.

> *Note* : En général, on positionne `NDEBUG` à la compilation, sans l'écrire dans le fichier C, en utilisant l'option `-D` du compilateur (`-DNDEBUG`) :
>
> ```
> gcc -Wall -pedantic -DNDEBUG assert-comprendre.c -o assert-comprendre
> ```
>
> On peut aussi ajouter `-DNDEBUG` à la définition de `CFLAGS` dans le fichier `Makefile`.

Types

Plusieurs types fondamentaux sont définis en C :

- Des types discrets : `int` (entier), `bool`(bouléen), `char` (caractère)
- Des types réels : `float` et `double`, à simple et double précision.

> Note : Il faut inclure le module `stdbool` pour utiliser le type booléen, et ses valeur `true` et `false`

> Note : Il faut inclure le module `stdbool` pour utiliser le type booléen, et ses valeur `true` et `false`.

Exemples

```
int entier_1 = 20;
bool est_vide = false;
char initiale = 'B'; //Caractère constant 'B' entre guillemets simples/
```

Modificateurs de type

Il existe aussi des modificateurs de type : `long`, `short`, `unsigned`. Ils sont utilisés pour modifier certains types fondamentaux `int`, `double`, `float`.

La taille en mémoire d'une variable entière de type `short int` est inférieure à la taille mémoire d'une variable de type `int`, qui elle même est inférieure à une variable de taille `long int`.

Le modificateur `unsigned` définit un type à valeurs positives ou nulles.

Exercice 3 -- Valeurs maximales et conversion implicites

La valeur maximale des types dépend du système d'exploitation. Elles sont enregistrées dans les bibliothèques `limits.h` pour les entiers et `float.h` pour les flottants.

[3.1] Exécuter l'exemple suivant pour les observer.

In [31]:

```
#include <stdio.h>
#include <stdlib.h>
// Liste les valeurs maximales des entiers pour votre système
#include <limits.h>
// Liste les valeurs maximales des flottants pour votre système
#include <float.h>

int main(){
    printf("Valeur maximale d'un entier %d \n", INT_MAX);
    long int entier_long = -20000000; // Déclaration d'un entier long
    printf("Valeur maximale d'un entier long %ld > %ld \n\n", LONG_MAX, entier_long);

    unsigned long int entier_non_signe = entier_long; // Il y a conversion implicite
    printf("Valeur maximale d'un entier non signé %u \n", UINT_MAX);
    printf("Valeur maximale d'un entier non signé long %lu > %lu \n\n", ULONG_MAX,
entier_non_signe);

    float flottant_simple = 20.13;
    double flottant_double;
    long double long_double = 200001102.2;
    printf("Valeur maximale d'un réel simple : \n%f \n < valeur max double : \n%lf \n < valeur max
long double : \n%Lf ", FLT_MAX, DBL_MAX, LDBL_MAX);

    return EXIT_SUCCESS;
}
```

```
Valeur maximale d'un entier 2147483647
Valeur maximale d'un entier long 9223372036854775807 > -20000000

Valeur maximale d'un entier non signé 4294967295
Valeur maximale d'un entier non signé long 18446744073709551615 > 18446744073689551616

Valeur maximale d'un réel simple :
340282346638528859811704183484516925440.000000
 < valeur max double :
179769313486231570814527423731704356798070567525844996598917476803157260780028538760589558632766878
404589535143824642343213268894641827684675467035375169860499105765512820762454900903893289440758685
513394230458323690322294816580855933212334827479782620414472316873817718091929988125040402618412485
.000000
 < valeur max long double :
```

```
x valeur max long double :
11897314953572317650212638530309702051690633222946242004403237338917370055229707226164102903365288
54569780749557731442744315367028843419812557385374367867359320070697326320191591828296152436552951C
91086614311790632169778838896134786560600399148753433211454911160088679845154866512852340149773037E
91254793939662231513836224178385427439178381387178058894875405751682263476592355769748051137256490Z
85522249479139937758502601177354918009979622602685950855888360815984690023564513234659447638493985S
56284579661772930407806609229102715046085388087959327781622986827547830768080040150694942303411728S
71003357140105597752421240573470073862516601108283791196230084692772009651535002084744707924438485A
88672300061908512647211195136146752763351956292759795725027800298079590419313960302147099703527646T
30922022679656280991498232083329641241038509239184734786121921697210543484287048353408113042573002Z
13489173471742348007148807510020643905172342476560047217680964861079949434157034763206435586242074A
42438056613601760883747816538902780957697597728686007148702828795556714140463261583262360276289631G
78484254486860609948270867968048078702511858930838546584223040908805996294594586201903766048446790S
22254105307759010657606713472001258464069570302571389609837579989269545530523685607586831792231136C
46885088077187210470520395758748001314313144425494391994017575316933939236688185618912993172910425Z
36835159922322050998001677102784035360140829296398115122877768135706045789343535451696539561254048E
71697868932116710872290880827783505182288576460622187397028516550837209923494833344352289847512327S
63606621390228126470623407535207172405866507951821730346378263135339370677490195019784169044182473E
62828586857741432581165364040218402724913393320949219498422442730427019873044536620350262386957804G
36014472919971230955300572061418669748528468561865148327159744812031219467516863793430961896151073E
55242148519520176285859509105183947250286387163249416761380499631979144187025430270675849519200883T
69401581740046711477877201459644461175204059453504764721807975761111720846273639279600339670470037G
45095531841500737964126050479232516613548412918842113408230154733047540670728187635036173329080059S
32520707167390454777712968226520622565143991937680440029238090311243791261477625596469422198137514G
79446870358004392507659451618379811859392049544036114915310782251072691486979809240946772142727012A
71874092167566136349389004512323516681460893224006979931760178053381918499819330084109859939387602S
39091141452600372028487213241195542428210183120421610446740462163533690058366460659115629876474552S
45003932941404131495400677602951005962253022823003631473824681059648442441324864573137437595096416I
80241293518762046681356368775328146755387988717718365128939471953350618850032676073543886733680020T
84965701457609034985757124304510203873049485425670247933932280911052604153852899484920399109194612S
91633289917998094380337879522093131466946149705939664152375949285890960489916121944989986384837022A
22491489246784102061833646274169695763076324802355879752452537370354338829608627534277400163334340S
53704850737454481975472222897528108302089868263302028525992308416805453968791141829762998896457648Z
87504562854924265165217750799516259669229114977788962356670956627138482018191348321687995863652637E
82850700993372943967846398790249145142227425270063639423279984839767399871544185542015622441549266S
51550468548925862027608576183712976335876121538256512963353814166394951655600026415918655485005705Z
31952919918807954522394649627635630178580896692226406235382898535867595990647008385687123810329591S
48462507689922584193054807636202150890221492205280698420183508405869384938154989094454619778930291I
51677540623227829831403347327660395223160342282471752818181884430488092132193355086987339586127607S
66652375555675803171490108477320096424318780070008797346032906278943553743564448851907191616455141I
19393996907674151564028265436640267600950875239455073415561358679330660317447209244465135323666476A
40085196704077110364053815007348689179836404957060618953500508984091382686953509006678332447257871Z
04415284924840041850932811908963634175739897166596000759487800619164094854338758520657116541072260S
81501231443779440087493019447443307843889957018427100048083050121771235606228950762690428568000477I
15808935851559386317665294808903126774702966254511086154895839508779675546413794489596052797520987A
39762578592105756284401759349324162148339565350189196811389091843795734703269406342890087805846940S
34793980806742732362978871008671758025315613023560648787092598652884163509725295370911143172048877A
53905400942537542411931794417513706468964386151771884986701034153254238591108962471088538580868883T
58648564145934262121086647588489260031762345960769508849149662444156604419552086811989770240.000000
```

[3.2] Il est possible d'initialiser un entier non signé avec un entier signé. Observer la valeur obtenue pour l'entier non signé. D'où provient-elle ?

> *Note* : pour afficher ces valeurs maximales avec `printf`, on doit modifier les lettres qui suivent le signe `%` dans `printf` pour adapter le format au type des variables :
>
> - `%d`, `%ld` : permet d'afficher un entier, un entier long.
> - `%u`, `%lu` : permet d'afficher un entier non signé, un entier non signé long.
> - `%f`, `%lf`, `%Lf` : permet d'afficher un flottant, un double et un long double.

[3.3] Modifier le code ci-dessus pour que :

1. l'entier non signé soit affiché comme un entier signé par `printf`.
2. l'entier non signé soit affiché comme un flottant simple.

Qu'observez-vous dans les deux cas ? Que peut-on en conclure sur les avertissements du compilateur ?

Variables

Déclaration de variables

Les variables peuvent être déclarées n'importe quand. Typiquement, **on déclare une variable au moment où on l'utilise** de la façon suivante :

```
type identifiant_variable;
```

Exemples :

```
int valeur, produit; // déclaration de deux variables entières,
double numerateur; // déclaration d'une variable réelle.
char initiale; // déclaration d'une variable caractère
```

Affectation de variables

L'initialisation et l'affectation des variables est réalisé **avec l'opérateur =**.

In [ ]:

```
#include <stdlib.h>
int main(){
    int valeur = 10, produit = 23; // déclaration et initialisation de deux entiers,
    double numerateur = 10.3;
    char initiale = 'A';

    produit = produit * valeur; // affectation
    valeur = valeur + 1;
    return EXIT_SUCCESS;
}
```

Opérateurs arithmétiques

Les opérateurs binaires **+, - , * et /** s'appliquent à des variables de type entier (signés ou non), booléens, flottant ou double.

Les opérateurs binaires **/ et %** utilisés sur **des entiers**, fournissent respectivement le quotient de le reste de la division entière des deux termes.

> **Règle** :
>
> - Division **entière** : la division `a \ b` **si `b` est entier** fournit le quotient de la division entière.
> - Division **réélle** : la division `a \ b` **si `b` est réel** fournit un résultat réel.

Les opérateur unaires **- et +** s'appliquent aux entiers signés et aux types réels.

> Note : Des opérateur mathématiques avancés sont disponibles dans la bibliothèque `<math.h>` (puissance, log, etc.)

Exemples (à exécuter).

In [37]:

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

int main(){
    int quantite = 10;
    int prix = 15;

    int total = quantite * prix;
    assert(total == 150);
```

```
    float nb_personnes = 60;
    float prix_par_personne = total / nb_personnes;
    assert(prix_par_personne == 2.5);

    int nb_personnes_int = 60;
    prix_par_personne = total / nb_personnes_int;
    assert(prix_par_personne != 2.5); // Quelle est la valeur de prix_par_personne ici ?

    printf("Le prix par personne est de %f euros", prix_par_personne);
    return EXIT_SUCCESS;
}
```

```
Le prix par personne est de 2.000000 euros
```

Affectations avec opération

Les instructions de la forme `x = x # y` avec `#` un opérateur arithmétique binaire, se synthétisent en C par :

`x #= y`

Il est aussi possible de simplifier l'incrémentation et la décrémentation avec les opérateurs **++ et --**. On a :

- `i++;` équivalent à `i = i+1;`
- `i--;` équivalent à `i = i-1;`.

Exemples (à exécuter):

In [39]:

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

int main(){
    int valeur = 10, produit = 23;
    produit += valeur;  // On ajoute valeur à produit
    assert(produit == 33);

    produit *= 2; // multiplication par 2 puis affectation
    assert(produit == 66);

    produit /= 3; // division par 3 puis affectation
    assert(produit == 22);

    valeur++; // incrémentation de valeur
    produit--; // décrémentation de produit
    assert(valeur == 11 && produit == 21);

    valeur+=1; //similaire à Python
    produit-=1;
    assert(valeur==12 && produit ==20); // && est équivalent à EtAlors

    printf("%s", "Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Tous les tests passent.
```

Opérateurs de comparaison

Les opérateurs **==, !=, <, >, >= et <=** permettent de comparer deux variables. La proposition `vrai` s'évalue à `1` et proposition `faux` à `0`.

Opérateurs logiques

C définit les opérateurs logiques suivants :

- Le EtAlors algorithmique : **&&**,
- Le OuSinon algorithmique : **||**,

- La négation : **!**

Types caractère et entier en langage C

Un caractère est représenté en mémoire comme un entier non-signé (`unsigned int`) qui correspond au code ASCII de ce caractère. Les types caractère et entier (non-signé) sont donc compatibles.

L'exemple suivant (à exécuter) présente les différentes opérations permettant de convertir un entier en caractère, et réciproquement.

In [40]:

```c
#include <assert.h>
#include <stdio.h>
#include <stdlib.h>
int main(){
    // Conversion du char '1' en l'entier 1
    char c_char = '1';
    int c_int = c_char - '0'; //on retire le code ascii du caractère '0'
    assert(c_int == 1);

    // Conversion de l'entier 1 en un char qui vaut '1'
    int new_int = 1;
    char c_char2 = new_int + '0'; //on ajoute le code ascii du caractère '0'
    assert(c_char2 == '1');  // c_char2 est bien égal au caractère

    printf("%s", "Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Tous les tests passent.
```

Exercice 4 -- Comprendre les opérateurs arithmétiques et les relations entre caractere et entier.

[4.1] Dans cet exercice, suivre la consigne présentée dans les commentaires.

In [42]:

```c
#define XXX -1

// Consigne : dans la suite *** uniquement ***, remplacer XXX par le bon
// résultat (une constante littérale).

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

int main(void) {
    // Comprendre les opérateurs arithmétiques
    assert(-5 == 5 - 2 * 5);
    assert(5 == 25 % 10);
    assert(2 == 25 / 10);
    assert(2.5 == 25 / 10.0);

    // Comprendre les relations caractères et entiers
    assert(5 == '5' - '0');
    assert('7' == '0' + 7);
    assert('D' == 'A' + 3);

    printf("%s", "Bravo ! Tous les tests passent.\n");

    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

Portée et masquage des variables

Portée et masquage des variables.

Un bloc est une série d'instructions délimitée par **une paire d'accolades**.

- **Portée** : Les variables déclarées dans un bloc sont libérées quand l'accolade fermante est exécutée. On dit que leur portée se limite au bloc où elles sont déclarées.

```
{ //debut du bloc B1
    int age = 20;
    { // debut du bloc B2
        int nouvel_age = 25 ; // variable locale à B2
    } // fin du bloc B2
    // La variable nouvel_age n'existe plus.
    age = age + 1;
} //fin du bloc B1
```

- **Masquage** : Les variables déclarées dans un bloc peuvent avoir le même identifiant qu'une variable déclarée avant l'ouverture du bloc. Dans ce cas, la variable déclarées dans le bloc masque la variable homonyme déclarées avant : c'est elle qui est utilisée par les instructions du bloc.

Masquage et portée sont illustrés dans l'exemple (à exécuter) suivant :

In [44]:

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

int main() { //debut du bloc B1
    int alea = 20, diviseur = 2;

    { //debut du bloc B2

        int alea = 3; // masquage de la variable entière alea par la variable alea entière
        float diviseur = 2.0; // idem pour le diviseur réel qui masque le diviseur de type entier.
        float res_reel = alea / diviseur;
        assert(res_reel = 1.5);

    } // du bloc B2
    int res_int = alea / diviseur;
    assert(res_int = 10);

    printf("%s", "Les tests passent\n");
    return EXIT_SUCCESS;
} //fin du bloc B1
```

```
Les tests passent
```

Exercice 5 -- Portée et masquage des variables

[5.1] Dans cet exercice, suivre la consigne présentée dans les commentaires.

In [59]:

```
// Objectifs : Illustrer portée et masquage.

#define XXX -1

// Consigne : *** dans la suite uniquement ***, remplacer XXX par le bon résultat (une
// constante littérale).  Ne compiler et exécuter que quand tous les XXX ont été traités.

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

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

int main(void) {
    int x = 10;
    assert(10 == x);
    {
    int y = 7;
    assert(10 == x);
    assert(7 == y);

    {
        char x = '?';
        assert('?' == x);
        assert(7 == y);
        y = 69;
    }

    assert(10 == x);
    assert(69 == y);
    }
    assert(10 == x);

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

Constantes

Les constantes se déclarent généralement juste après l'inclusion des bibliothèques. Leur valeur ne peut être modifiée. Il est possible de définir des constantes de deux façons :

1. En utilisant le mot-clé `const` pour obtenir une constante typée

```c
const int MAJORITE_EU = 18; // déclaration d'une constante typée MAJORITE_EU
    const int MAJORITE_US = 21; // déclaration d'une constante typée MAJORITE_US
    const char CM = 'c'; // déclaration d'une constante caractère.
```

> **Règle** : Un caractère se distingue par l'utilisation de guillements simples (apostrophe) : 'A', 'c', 'D', '\n', '\t'.

1. En définissant une constante pré-processeur :

```c
#define MAJORITE_EU 18 // déclaration d'une constante pré-processeur 18,
```

Le pré-processeur remplace les occurrences de `MAJORITE_EU` par la valeur 18.

> **Règle** : Pas de point-virgule à la fin d'une instruction pré-processeur.

Constantes littérales

Ce sont les valeurs numériques écrites directement dans les instructions :

```c
int age = 20; // 20 est une constante littérale
    char initiale_nom = 'M' // Le caractère 'M' est une constante littérale.
```

Expressions et compatibilité entre types

Définition d'une expression

Une expression est une instruction qui est caractérisée par une valeur de retour. Voici quelques exemples :

- Une variable initialisée

```
int val = 20;
   val; // La variable `val` vaut 20 dans cette instruction.
```

- Une comparaison : `(b > 20)`. Cette expression vaudra `true` ou `false`.
- L'utilisation d'opérateur arithmétiques :

```
int x = 3;
  x + 3; // Cette expression vaut 6
  (x * 2) / 3; // Cette expression vaut 2
```

> *Note* : Une affectation est aussi une expression : `val = 40` est une expression qui vaut 40. L'utilisateur de l'affectation comme expression est à éviter en C.

Priorité des opérateurs

En C, la priorité des opérateurs évalués dans une même expression est la suivante :

Priorité	Opérateurs
1	+, -, **!** (unaires)
2	*, /(entier), /(flottant), %, **&&**
3	+, -, __\
4	**<, >, <=, >=, ==, !=**

La priorité 1 est la plus forte. Les opérateurs booléens sont présentés en gras.

Compatibilité entre types

En C, **une expression peut être composée d'expressions de types différents si ces types sont compatibles** . Voici quelques exemples :

In [66]:

```c
#include <assert.h>
#include <stdlib.h>
#include <stdio.h>
int main(){
    int quantite = 5;
    float prix = 12.3;

    float total = prix * quantite; //l'entier quantité est compatible avec les flottants
    float recette = 12; // l'entier 12 est compatible avec le flottant recette.
    assert(total == 12.3*5.0 && recette == 12.0);

    quantite = total / recette;
    assert(quantite != total / recette);
    assert(quantite==5);
    assert(recette==12.0);
    assert(recette==12);
    // le réel obtenue par la division de total et recette
    // n'est pas compatible avec l'entier quantite.

    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo ! Tous les tests passent.
```

Si un type A est compatible avec un type B, on peut interchanger une expression de type B par une expression de type A sans changer la valeur de l'expression.

Typiquement, un type A est compatible avec un type B si le passage de l'un à l'autre n'engendre pas de perte de donnée :

- Un entier est compatible avec un réel (12 devient 12.0)
- Un réel n'est pas compatible avec un entier (le passage de 1.3 à 1 introduit une perte d'information).

Conversion explicite

Il est possible de convertir explicitement une expression pour qu'elle soit évaluée avec un autre type. Pour cela, on utilise la notation :

```
(type) expression
```

Exemple Voici l'exemple illustrant la division entière et réelle présenté précédement. Il a été modifié pour déclarer le nombre de personnes avec un entier, et dériver tout de même un prix par personnes avec une division réelle grâce à une conversion explicite.

In [1]:

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

int main(){
    int quantite = 10;
    int prix = 15;

    int total = quantite * prix;
    assert(total == 150);

    int nb_personnes_int = 60;
    float prix_par_personne = total /(float) nb_personnes_int;
    assert(prix_par_personne == 2.5); // Maintenant on effectue bien une division réelle

    printf("Le prix par personne est de %1.2f euros", prix_par_personne);
    return 0;
}
```

```
Le prix par personne est de 2.50 euros
```

> *Note* : On observe que le descripteur de format `%f` a été étendu à `%1.2f` pour limiter le nombre de décimales à 2.

Exercice 6

[6.1] Ecrire un programme qui calcule le périmètre et l'aire d'un cercle, étant donné un rayon qui vaut `15`. Le rayon est une variable entière. Les éventuelles constantes seront déclarées comme des constantes pré-processeur.

[6.2] Ecrire les deux résultats réels à l'écran.

In [70]:

```c
#include <stdlib.h>
#include <stdio.h>
#define pi 3.14

int main(){
    int r=15;
    float p=2*pi*r;
    float a=pi*(r*r);
    printf("Le périmètre du cercle est %f \nL'aire du cercle est %f \n",p,a);
```

```
    return EXIT_SUCCESS;
}
```

```
Le périmètre du cercle est 94.199997
L'aire du cercle est 706.500000
```

Exercice 7

[7.1] Compléter et corriger le corps des fonctions ci-dessous (voir TODO)

In [31]:

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

/**
 * \brief obtenir le chiffre des unités d'un entier naturel.
 * \param[in] nombre le nombre dont on veut obtenir le chiffre des unités
 * \return le chiffre des unités de nombre
 * \pre nombre positif : nombre >= 0
 */
int chiffre_unites(int nombre)
{
    assert(nombre >= 0);
    return nombre%10;
}

/**
 * \brief obtenir le chiffre des dizaines d'un entier naturel.
 * \param[in] nombre le nombre dont on veut obtenir le chiffre des dizaines
 * \return le chiffre des dizaines de nombre
 * \pre nombre positif : nombre >= 0
 */
int chiffre_dizaines(int nombre)
{
    assert(nombre >= 0);
    return (nombre/10)%10;
}

/**
 * \brief Indiquer si une année est bissextile.
 * \param[in] annee l'année à considérer
 * \return vrai si l'année est bissextile
 * \pre année positive : annee > 0
 */
bool est_bissextile(int annee) {
    // TODO: Donner le bon code !
    // Attention : on n'utilisera pas de conditionnelle,
    // seulement les opérateurs logiques.
    assert(annee>0);
    return ((((chiffre_dizaines(annee)*10) + chiffre_unites(annee))%4==0 && annee%100!=0) || annee%
400==0 );
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                   NE PAS MODIFIER CE QUI SUIT...                       //
//                                                                        //
////////////////////////////////////////////////////////////////////////////


void test_chiffre_unites(void) {
    assert(5 == chiffre_unites(1515));
    assert(2 == chiffre_unites(142));
    assert(0 == chiffre_unites(0));
```

```
    printf("%s", "chiffre_unites... ok\n");
}

void test_chiffre_dizaines(void) {
    assert(1 == chiffre_dizaines(1515));
    assert(4 == chiffre_dizaines(142));
    assert(9 == chiffre_dizaines(91));
    assert(8 == chiffre_dizaines(80));
    assert(0 == chiffre_dizaines(7));
    assert(0 == chiffre_dizaines(0));
    assert(0 == chiffre_dizaines(1900));
    printf("%s", "chiffre_dizaines... ok\n");
}


void test_annee_bissextile(void) {
    // cas simples
    assert(! est_bissextile(2019));
    assert(est_bissextile(2020));
    assert(est_bissextile(2016));

    // multiples de 100
    assert(! est_bissextile(1900));
    assert(! est_bissextile(2100));

    // multiples de 400
    assert(est_bissextile(1600));
    assert(est_bissextile(2000));
    assert(est_bissextile(2400));

    printf("%s", "annee_bissextile... ok\n");
}


int main(void) {
    test_chiffre_unites();
    test_chiffre_dizaines();
    test_annee_bissextile();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
chiffre_unites... ok
chiffre_dizaines... ok
annee_bissextile... ok
Bravo ! Tous les tests passent.
```

Entrées et sorties en Langage C

Définition

- Les entrées sont des instructions qui permettent de lire des données provenant de l'environnement d'exécution (clavier, souris, capteur, fichier, réseau, etc.).
- Les sorties sont des instructions qui permettent de transférer des données à l'environnement d'exécution (moniteur, actuateur, fichier, réseau, etc.).

Cette partie présente l'utilisation des **entrées clavier ou des sorties moniteur**. Les autres types de périphériques seront abordés dans d'autres enseignements.

> Attention : **la gestion des entrées/sorties en C n'est pas triviale !**

Flux de données

La notion de flux de données est fondamentale. Les flux permettent d'interagir avec les périphériques pour échanger des données. Un flux est une file d'attente de type FIFO (*fist in first out*) :

- la données la plus ancienne peut être lue. Elle est alors consommée (supprimée de la file),
- la donnée la plus récente est insérée en fin de file. On dit qu'elle est écrite dans la file.

En C, on peut manipuler des flux qui enregistrent des données de deux types :

- **texte** : on y enregistre une suite de caractères, séparés par des retour-chariots,
- **binaire** : on y enregistre une suite d'octets.

Il existe des files sont définies par défaut :

- `FILE* STDOUT` : flux de sortie vers le moniteur
- `FILE* STDIN`: flux d'entree depuis le clavier Elles sont toutes de type `FILE*`

> Note :
>
> - Ces files sont définies dans le module `stdlib`
> - Les sous-programmes mentionnés par la suite sont définis dans le module `stdio`
>
> Remarque :
>
> - En 1SN nous ne traiterons que les entrées sorties en mode **texte**

Ecrire les sorties

Sorties formattés

L'objectif du sous-programme `printf()` est d'écrire des données typées à l'écran. L'objectif est d'afficher en une seule instruction le contenu de variables de type entier, flottant, chaine de caractères, voire une combinaison hétérogène de variables.

Le sous-programme `printf` du module `stdio` est définit comme suit :

```
int printf("format", param1, param2, etc.);
```

La chaîne "format" est une chaîne de caractères, parsemée de **spécificateurs de format**.

Un spécificateur de format commence par le caractère %. Il y autant de spécificateurs de format que de paramètres. A l'exécution,

- le 1er spécificateur est remplacé par la valeur du 1er paramètre,
- le 2er spécificateur est remplacé par la valeur du 2er paramètre,
- etc.

Le spécificateur indique comment afficher la variable qui lui correspond :

- %d ou %i : indique à printf que l'on souhaite afficher le paramètre comme un entier signé
- %u : indique à printf que l'on souhaite afficher le paramètre comme un entier **non**signé
- %f, %lf, %Lf : indique à printf que l'on souhaite afficher le paramètre comme un float, double ou long double. Il est possible de limiter le nombre de décimales : %1.3f limite le nombre de décimales à 3.
- %c : indique à printf que l'on souhaite afficher le paramètre comme un caractère
- %s : indique à printf que l'on souhaite afficher le paramètre comme une chaîne de caractères
- %p : indique à printf que l'on souhaite afficher le paramètre comme une adresse
- etc.

> **ATTENTION** : le compilateur ne vérifie pas forcément la cohérence entre le spécificateur et le type du paramètre correspondant ! Des warnings sont généralement observés.

Exemples :

In [32]:

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

```c
#include <stdlib.h>

int main(){
    float cote = 2.0;     //longueur du côté
    char unite = 'm';
    printf("Le périmètre du carré de côté %1.0f%c est : ", cote, unite);
    //affichage du flottant avec 0 chiffres après la virgule

    //calcul et affichage du perimètre
    float perimetre = 4 * cote;
    printf("%1.2f%c\n", perimetre, unite);
    //affichage du flottant avec 2 chiffres après la virgule

    return EXIT_SUCCESS;
}
```

```
Le périmètre du carré de côté 2m est : 8.00m
```

> *Note* : Il existe d'autres sous-programmes d'écriture qui ne seront pas présentés ici : `putchar()`, `fputc()`, `sprintf()`, `fprintf()`.

Lire les entrées

Entrées formattés

L'objectif du sous-programme `scanf()` est de lire des données typées depuis le clavier. Le sous-programme `scanf` du module `stdio` est définit comme suit :

```
int scanf("format", &param1, &param2, etc.);
```

La chaîne "format" ne comporte **principalement des spécificateurs de formats**. Chaque format fait référence à un des paramètres, dans l'ordre d'apparition. Le spécificateur indique comment lire la variable qui lui correspond :

- %d ou %i : indique à `scanf` qu'il doit lire un entier
- %f : indique à scanf qu'il doit lire un float
- etc.

La donnée lue est écrite à **l'adresse** de `param1`, `param2`. etc.

> Si dans le "format" on insère un espace entre deux spécificateurs, tous les caractères 'blancs' (espace, tabulation) sont consommés mais non interprétés.

L'entier retourné par `scanf` représente le nombre de paramètres lus avec succès.

Exemples :

```c
// Lire un entier
int monentier;
scanf("%i", &monentier);
// Lire un flottant avec 2 décimales maximum.
float monfloat;
scanf("%1.2f", &monfloat);
// Lire deux caracteres d'affilée non blancs
char c1, c2;
scanf("%c %c", &c1, &c2);
```

> *Note* : Il existe d'autres sous-programmes de lecture des entrées qui ne seront pas présentés ici : `getchar()`, `fgetc()`, `sscanf()`, `fscanf()`.

Les structures de contrôle

Elles permettent de contrôle l'ordre d'exécution des instructions. En C, il existe

- La séquence
- Les structures conditionnelles :
  - `if ... then ... else`
  - `switch ... case ...`
- Les boucles :
  - Répéter : `do ... while`
  - TantQue : `while ...`
  - Pour : `for ...`

Les conditionelles

1. La conditionnelle simple :

```
if (cond) {
    sequence1
} else {
    sequence2
}
```

Si la condition `cond` est vraie, `séquenc1` est exécutée, sinon, `sequence2` est exécutée.

La clause `SinonSi` n'existe pas, on imbrique les conditionnelles pour introduire une étape de sélection supplémentaire :

```
if (cond1) {
    sequence1
} else if (cond2) {
    sequence2
} else {
    sequence3
}
```

`sequence3` est exécuté si `cond1` et `cond2` sont fausses.

Exercice 8 - Ecrire des conditionnelles.

[8.1] Compléter et corriger le corps des fonctions ci-dessous (voir TODO).

In [35]:

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

```c
/**
 * \brief Retourner '<', '>' ou '=' pour indiquer si n est strictement négatif,
 * strictement positif ou nul.
 * \param[in] nombre le nombre dont on veut évaluer le signe
 * \return un caractère donnant le signe d'un nombre
 */
char signe(int nombre)
{
    char a;
        if (nombre<0){
            a='<';
        }else if (nombre>0){
            a='>';
        }else
            a='=';
    return a;
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                     NE PAS MODIFIER CE QUI SUIT...                     //
//                                                                        //
////////////////////////////////////////////////////////////////////////////

void test_signe() {
    assert('<' == signe(-821));
    assert('<' == signe(-1));
    assert('=' == signe(0));
    assert('>' == signe(125));
    assert('>' == signe(1));
    printf("%s", "signe... ok\n");
}


int main(void) {
    test_signe();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
signe... ok
Bravo ! Tous les tests passent.
```

2. La conditionelle multiple : Elle s'exprime avec la structure de contrôle `switch .. case`. Elle suit la syntaxe suivante :

```c
switch (expr) {
    case choix1 :
        sequence1;
        break;
    case choix2 :
        sequence2;
        break;
    case default :
        sequence_def;
}
```

A l'exécution:

1. `(expr)` est évalué. `(expr)` est une expression de type **discret** (entier, booléen ou caractère)
2. L'exécution se poursuit au niveau du `case` qui correspond a la valeur de `(expr)` ou au niveau du `default` si aucune correspondance n'est trouvée.

Autrement dit, si .

- Si `expr == choix1`, toutes les instructions sont exécutées **à partir de sequence1**.
- Si `expr == choix2`, toutes les instructions sont exécutées **à partir de sequence2**.
- Si `expr != choix1 && expr != choix2`, sequence_def est exécuté.

Si l'instruction **break;** est rencontrée, les instructions suivantes du bloc `switch` ne sont jamais exécutées.

> Note : Il est important d'utiliser l'instruction **break;** pour n'exécuter qu'une séquence par choix possible pour retrouver le comportement algorithmique d'un `Selon .. Dans`.

Exercice 9 - Comprendre le `switch ... case`

[9.1] Dans la fonction `test_f` du programme suivant, remplacer XXX par la valeur qui sera retournée par l'appel correspondant à la fonction `f`.

In [43]:

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

#define XXX -1

// Une fonction f qui retourne un entier en fonction du paramètre n fourni.
int f(int n) {
    int r = 0;

    // modifier r
    switch (n) {
    case 1:
        r += 1;
        break;
    case 2:
    case 3:
        r += 8;
        break;
    case 4:
    case 5:
    case 7:
        r += 10;
    case 10:
    case 11:
        r += 5;
        break;
    case 12:
        r += 50;
        break;
    case 13:
        r += 100;
    default:
        r -= 1;
    }

    return r;
}

void test_f(void)
{
    assert(8 == f(3));
    assert(-1 == f(-5));
    assert(-1 == f(0));
    assert(50 == f(12));
    assert(99 == f(13));
    assert(8 == f(2));
    assert(5 == f(10));
    assert(15 == f(5));
}

int main(void) {
    test_f();
    printf("%s", "Bravo !  Pas d'erreur détectée.\n");
    return EXIT_SUCCESS;
}
```

```
Bravo !  Pas d'erreur détectée.
```

Exercice 10 - Ecrire un `switch .. case`

[10.1] Compléter et corriger le corps de la fonction `nb_jours_mois` ci-dessous (voir TODO).

In [45]:

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

/**
 * \brief Obtenir le nombres de jour d'un mois d'une année non bissextile.
 * \param[in] mois le mois considéré (de 1, janvier, à 12, décembre)
 * \return le nombre de jours du mois considéré
 */
char nb_jours_mois(int mois)
{
    int j=0;
    switch(mois) {
        case 1:
            j=31;
            break;
        case 2:
            j=28;
            break;
        case 3:
            j=31;
            break;
        case 4:
            j=30;
            break;
        case 5:
            j=31;
            break;
        case 6:
            j=30;
            break;
        case 7:
            j=31;
            break;
        case 8:
            j=31;
            break;
        case 9:
            j=30;
            break;
        case 10:
            j=31;
            break;
        case 11:
            j=30;
            break;
        case 12:
            j=31;
            break;
    }
    return j;
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                     NE PAS MODIFIER CE QUI SUIT...                     //
//                                                                        //
```

```
//////////////////////////////////////////////////////////////////////////////
void test_nb_jours_mois() {
    assert(31 == nb_jours_mois(1));
    assert(28 == nb_jours_mois(2));
    assert(31 == nb_jours_mois(3));
    assert(30 == nb_jours_mois(4));
    assert(31 == nb_jours_mois(5));
    assert(30 == nb_jours_mois(6));
    assert(31 == nb_jours_mois(7));
    assert(31 == nb_jours_mois(8));
    assert(30 == nb_jours_mois(9));
    assert(31 == nb_jours_mois(10));
    assert(30 == nb_jours_mois(11));
    assert(31 == nb_jours_mois(12));
    printf("%s", "nb_jours_mois... ok\n");
}


int main(void) {
    test_nb_jours_mois();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
nb_jours_mois... ok
Bravo ! Tous les tests passent.
```

Les boucles / répétitions

La répétition `do ... while`

On répète au moins une fois une séquence. La condition d'arrêt est testée une fois la séquence exécutée.

```
   do {
          sequence;
      }
      while (cond);
```

Exemple :

In [48]:

```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <assert.h>
/*
 * \brief Obtenir une valeur aléatoire entre min et max, inclus.
 * \param[in] min borne minimale,
 * \param[in] max borne maximale,
 * \return valeur aleatoire entre min et max
 * \pre min >= 0, max <= RAND_MAX
 */
int alea_borne(int min, int max){
    assert(min >= 0);
    assert(max <= RAND_MAX);

    // Initialisation du générateur de nombres aléatoires avec la date courante
    srand(time(NULL));
    int alea;
    do {
```

```
        alea = rand(); // valeur aleatoire entre 0 et RAND_MAX
    }
    while (alea < min || alea > max);
    return alea;
}

int main(void) {
    int val = alea_borne(4, 10);
    assert(val >= 4 && val <= 10);
    printf("val = %d\n", val);

    val = alea_borne(2, 25);
    assert(val >= 2 && val <= 25);
    printf("val = %d\n", val);
    return EXIT_SUCCESS;
}
```

```
val = 10
val = 4
```

La répétition `while`

On répète une séquence qui peut ne jamais être exécutée. On sort de la boucle quand la condition est fausse :

```
   while (cond) {
           sequence;
       }
```

Exemple :

In [49]:

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

#define LIMITE 300

int main(void) {
    int prec = 1, un = 2;
    int rang = 2;
    int nouveau;

    while (un < LIMITE) {
        // Determiner le nouveau terme de la suite de Fibonacci
        nouveau = un + prec;
        // Enregistrer les termes un et prec
        prec = un;
        un = nouveau;
        // Calculer le rang
        rang ++;
    }
    printf("La valeur de la suite de fibonacci >= %d est %d. Elle est de rang %d\n", LIMITE, un, ra
ng);

    return EXIT_SUCCESS;
}
```

```
La valeur de la suite de fibonacci >= 300 est 377. Elle est de rang 13
```

La répétition `for`

Si on connait le nombre d'itérations, on utilise une boucle Pour :

```
   for (instruction_init_compteur; condition_boucle; instruction_incr_compteur) {
           sequence;
       }
```

On a ici :

- `instruction init increment` : une instruction qui initialise (voire déclare) le compteur,

- `condition_boucle` : une condition qui, **si fausse**, arrête la répétition.
- `instruction_incr_compteur` : une instruction qui précise comment le compteur varie à chaque répétition.

Exemple :

In [50]:

```c
#include <stdlib.h>
#include <stdio.h>
#define LIMITE 30

int main(void) {
    //calcul de la moyenne des LIMITE premiers entiers
    int somme = 0;

    // Déclaration du compteur i et initialisation à 1
    // Répétition si i <= LIMITE
    // Incrémentation i = i + 1 à chaque répétition.
    for (int i = 1; i <= LIMITE; i++) {
        somme += i;
    }
    float moyenne = somme / (float) LIMITE;
    printf("La moyenne des %d premiers entiers est %1.2f\n", LIMITE, moyenne);
    return EXIT_SUCCESS;
}
```

```
La moyenne des 30 premiers entiers est 15.50
```

Exercice 11 - Ecrire un `TantQue`

[11.1] Compléter et corriger le corps de la function `sommes_cubes_inférieurs_a` (voir TODO).

In [6]:

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>

/**
 * \brief Calculer la somme des cubes des entiers naturels dont le cube est inférieur
 * ou égal à limite.
 * \param[in] limite la limite à ne pas dépasser pour les cubes
 * \return la sommes des cubes
 * \pre limite positive : limite > 0
 */
int sommes_cubes_inferieurs_a(int limite)
{
    assert(limite >= 0);
    int a=1;
    int s=0;
    while ((a*a*a)<=limite) {
        s=s+ a*a*a;
        a=a+1;
    }
    return s;
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                    NE PAS MODIFIER CE QUI SUIT...                      //
//                                                                        //
////////////////////////////////////////////////////////////////////////////


void test_sommes_cubes_inferieurs_a(void) {
```

```
    assert(1 == sommes_cubes_inferieurs_a(5));
    assert(1 == sommes_cubes_inferieurs_a(1));
    assert(1 == sommes_cubes_inferieurs_a(7));
    assert(9 == sommes_cubes_inferieurs_a(8));
    assert(9 == sommes_cubes_inferieurs_a(26));
    assert(36 == sommes_cubes_inferieurs_a(27));
    assert(36 == sommes_cubes_inferieurs_a(63));
    assert(100 == sommes_cubes_inferieurs_a(64));
    assert(100 == sommes_cubes_inferieurs_a(124));
    assert(225 == sommes_cubes_inferieurs_a(125));
    printf("%s", "sommes_cubes_inferieurs_a... ok\n");
}


int main(void) {
    test_sommes_cubes_inferieurs_a();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
sommes_cubes_inferieurs_a... ok
Bravo ! Tous les tests passent.
```

Exercice 12 - Ecrire un `Répéter ... Jusqu'à`

[12.1] Compléter et corriger le corps de la function `frequence` (voir TODO).

In [17]:

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

/**
 * \brief Obtenir la fréquence d'un chiffre dans un nombre.
 * Exemples : la fréquence de 5 dans 1515 est 2. La fréquence de 3 dans 123 est 1.
 * La fréquence de 0 dans 412 est 0.
 * \param[in] chiffre dont ont veut calculer la fréquence
 * \param[in] nombre pour lequel on veut calculer la fréquence de chiffre
 * \return la fréquence de chiffre dans nombre
 * \pre chiffre est un vrai chiffre : 0 <= chiffre <= 9
 */
int frequence(int nombre, int chiffre)
{
    assert(chiffre >= 0);
    assert(chiffre <= 9);

    int r=0;
    int occ=0;
    do{
        r=nombre%10;
        nombre=nombre/10;
        if (r==chiffre){
            occ++;
        }
    }
    while (nombre!=0);
    return occ;
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                    NE PAS MODIFIER CE QUI SUIT...                      //
//                                                                        //
////////////////////////////////////////////////////////////////////////////
```

```
void test_frequence(void) {
    assert(2 == frequence(1515, 5));
    assert(1 == frequence(123, 3));
    assert(0 == frequence(421, 0));
    assert(3 == frequence(444, 4));
    assert(1 == frequence(0, 0));
    printf("%s", "frequence... ok\n");
}


int main(void) {
    test_frequence();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
frequence... ok
Bravo ! Tous les tests passent.
```

Exercice 13 - Ecrire un `Pour`

[13.1] Compléter et corriger le corps de la function `frequence` (voir TODO).

In [26]:

```
// Consigne : compléter et corriger le corps des fonctions ci-dessous (voir TODO).

#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>
#include <assert.h>

/**
 * \brief Calculer la somme des cubes des entiers naturels de 1 à max.
 * \param[in] max un entier naturel
 * \return la sommes des cubes de 1 à max
 * \pre max positif : max >= 0
 */
int sommes_cubes(int max)
{
    assert(max >= 0);
    int s=0;
    for (int i = 1; i <= max; i++) {
        s += i*i*i;
    }
    return s;
}


////////////////////////////////////////////////////////////////////////////
//                                                                        //
//                    NE PAS MODIFIER CE QUI SUIT...                      //
//                                                                        //
////////////////////////////////////////////////////////////////////////////


void test_sommes_cubes(void) {
    assert(1 == sommes_cubes(1));
    assert(9 == sommes_cubes(2));
    assert(36 == sommes_cubes(3));
    assert(100 == sommes_cubes(4));
    assert(225 == sommes_cubes(5));
    assert(0 == sommes_cubes(0));
    printf("%s", "sommes_cubes... ok\n");
}


int main(void) {
```

```
    test_sommes_cubes();
    printf("%s", "Bravo ! Tous les tests passent.\n");
    return EXIT_SUCCESS;
}
```

```
sommes_cubes... ok
Bravo ! Tous les tests passent.
```

Exercice BILAN 1 : Conversion pouce/centimètres

[B.1] Traduire l'algorithme du listing en Langage C. Il permet de convertir en pouces et en centimètres une longueur saisie en pouces, centimètres ou mètres.

Listing 1 – Conversions en pouce et en centimètres

```
1  Algorithme pouce2cm
2
3      -- Afficher une longueur saisie au clavier en pouces et en centimètres.
4
5  Constantes
6      UN_POUCE = 2.54     -- valeur en centimètres d'un pouce
7
8  Variables
9      valeur: Réel        -- valeur de la longueur lue au clavier
10     unité: Caractère    -- unité de la longueur lue au clavier
11     lg_cm: Réel         -- longueur exprimée en centimètres
12     lg_p: Réel          -- longueur exprimée en pouces
13
14 Début
15     -- saisir la longueur (valeur + unité)
16     Écrire ("Entrer une longueur (valeur + unité) : ")
17     Lire (valeur)               -- saisir la valeur
18     Lire (unité)                -- saisir l'unité
19
20     -- calculer la longueur en pouces et en centimètres
21     Selon unité Dans
22         'p', 'P':       { la longueur a été saisie en pouces }
23             lg_p <- valeur
24             lg_cm <- lg_p * UN_POUCE
25
26         'c', 'C':       { la longueur a été saisie en centimètres }
27             lg_cm <- valeur
28             lg_p <- lg_cm / UN_POUCE
29
30         'm', 'M':       { la longueur a été saisie en mètres }
31             lg_cm <- valeur * 100
32             lg_p <- lg_cm / UN_POUCE
33
34     Sinon               { Unité non reconnue }
35         lg_p <- 0
36         lg_cm <- 0
37     FinSelon
38
39     -- afficher la longueur en pouces et en centimètres
40     ÉcrireLn (lg_p, " p = ", lg_cm, " cm")
41 Fin
```

/*

- Auteur : SAOUDI Younes
- Version : 1
- Objectif : Conversion pouces/centimètres **/

include <stdio.h>

include <stdlib.h>

define UN_POUCE 2.54

void clean_stdin(void) { int c;

```
    do {
        c = getchar();
    } while (c != '\n' && c != EOF);
```

} int main() { / *Saisir la longueur* / float valeur; char unite; float lg_cm; float lg_p; char choix='y'; do { printf("\nEntrer une longueur (valeur + unité) : "); scanf("%f",&valeur); unite=getchar(); if (unite==' '){ unite=getchar(); }

```
        /* Calculer la longueur en pouces et en centimètres */
        switch(unite) {
            case 'p':
                lg_p=valeur;
                lg_cm=lg_p * UN_POUCE;
                break;
            case 'P':
                lg_p=valeur;
                lg_cm=lg_p * UN_POUCE;;
                break;
            case 'c':
                lg_cm=valeur;
                lg_p=lg_cm / UN_POUCE;
                break;
            case 'C':
                lg_cm=valeur;
                lg_p=lg_cm / UN_POUCE;
                break;
            case 'm':
                lg_cm=valeur*100;
                lg_p=lg_cm/UN_POUCE;
                break;
            case 'M':
                lg_cm=valeur*100;
                lg_p=lg_cm/UN_POUCE;
                break;
            default:
                lg_cm=0.0;
                lg_p=0.0;
            }

        /* Afficher la longueur en pouces et en centimètres */
        printf("%1.2f p = %1.2f cm \n",lg_p,lg_cm);
        printf("Voulez-vous recommencer ? [y/n] : ");
        clean_stdin();
        choix=getchar();
```

} while (choix=='y'); printf("\nVous avez choisi de quitter le programme.\n"); return EXIT_SUCCESS; }

[B.2] Modifier le programme pour que l'utilisateur puisse mettre des espaces (des blancs) entre la valeur et l'unité de la longueur.

[B.3] Ajouter la possibilité de recommencer.

Les types utilisateurs

Les types utilisateurs permettent au programmeur de définir des types plus évolués. Les 3 types en C sont :

- Les types énumérés.
- Les enregistrements.
- Les tableaux.

Ces types se définissent au début d'un programme, avant la signature du programme principal `int main()`

Les types énumérés

Un type énuméré permet de définir un ensemble discret de valeurs possibles. L'exemple suivant déclare un type énuméré `enum Jour` :

```
enum Jour { LUNDI, MARDI, MERCREDI, JEUDI, VENDREDI, SAMEDI, DIMANCHE};
```

Les constantes `LUNDI`, `MARDI`, sont des constantes entières qui vallent respectivemet 0, 1, 2, etc. On peut donc les comparer.

Une variable de type `enum Jour` ne peut prendre que ces valeurs.

```
enum Jour mon_jour = LUNDI;  //declaration d'une variable initialsée à LUNDI
```

Il est conseillé de créer un alias au type `enum Jour` à l'aide de l'instruction `typedef`

`typedef enum Jour Jour` Ici on a créé l'alias (le synonyme) `Jour`.

Les types enregistrement

Un type enregistrement permet de déclarer une variable qui regroupe plusieurs données hétérogènes (i.e. de type différent). En C, on le définit de la sorte :

```c
struct Date {
    int jour;
    Mois mois;
    int annee;
};
```

Note : ne pas oublier le ; après la dernière parenthèse.

Le type `struct Date` est un 3-uplet qui regroupe un jour, un mois et une année.

```c
struct Date d1, d2;  // déclaration de deux dates
    // initialisation champ par champ
    d1.jour = 30;
    d1.mois = AVRIL; //ici mois est un type énuméré
    d1.annee = 1997;
    // initialisation directe des 3 champs
    d2 = {31, DECEMBRE, 2012};
```

Il est aussi possible de créer un alias au type `struct Date` à l'aide de l'instruction `typedef`

`typedef struct Date Date` Ici on a créé l'alias (le synonyme) `Date`.

Exercice 14 : definir et utiliser un type enregistrement

14.1) Définir un type Point qui regroupe deux réels (x et y)

[14.1] Définir un type Point qui regroupe deux coordonnées entières, X et Y.

[14.2] Ecrire un programme principal qui génère deux points ptA et ptB au coordonnées (0,0) et (10,10) respectives. Il calcule la distance entre ptA et ptB en norme Euclidienne.

In [3]:

```c
#include <stdlib.h>
#include <stdio.h>
#include <math.h>
#include <assert.h>

// Definition du type Point
struct Point{
    float x;
    float y;
};
typedef struct Point Point;
int main(){
    // Déclarer deux variables ptA et ptB de types Point
    Point ptA,ptB;

    // Initialiser ptA à (0,0)
    ptA.x=0.0;
    ptA.y=0.0;
    // Initialiser ptB à (10,10)
    ptB.x=10.0;
    ptB.y=10.0;
    // Calculer la distance entre ptA et ptB.
    float distance = 0;
    distance=sqrt((ptB.x-ptA.x)*(ptB.x-ptA.x) + (ptB.y-ptA.y)*(ptB.y-ptA.y));

    assert(distance == sqrt(200));

    return EXIT_SUCCESS;
}
```

```
/tmp/tmpwu5f5fk_.out: symbol lookup error: /tmp/tmpdbadfwy2.out: undefined symbol: sqrt
[C kernel] Executable exited with code 127
```

Tableaux

Les tableaux permettent d'enregistrer **un nombre fini de données de même type**.

Déclarer une variable tableau

On pourra par exemple definir une **variable tableau** capable d'enregistrer NB entiers.

> Attention : En C, **les indices varient entre 0 et NB-1**.

```c
#define NB 4
    // déclaration d'une variable tableau de NB entiers
    int tab[NB];
    // Initialisation de la 2e case :
    tab[1] = 20;

    // Si on initialise à la déclaration, on n'a pas besoin de donner la taille
    int tab_2[] = {1, 4, -1, 4};
```

Déclarer un type tableau

La déclaration d'un type tableau est réalisé avec `typedef` :

```c
// declaration du type t_tab
    typedef int t_tab[NB];
    // declaration de variables tableau de type t_tab
    t_tab tab1, tab2;
    // l'accès aux données de tab1 et tab2 se fait de ma même façon :
    tab1[0] = 20;
```

> **Attention !** `tab1 = tab2` **est interdit**.

Exercice 15

[15.1] Definir un type `t_tableau` de réels de capacité 20.

[15.2] Completer et corriger la fonction `initialiser` qui permet d'initialiser chaque élément d'un tableau de type `t_tableau` à `0.0`.

[15.3] Completer et corriger la fonction `est_vide` qui vérifie que tous les éléments sont bien initialisés à `0.0`.

In [12]:

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

#define CAPACITE 20
// Definition du type tableau
typedef float t_tableau[20];

/**
 * \brief Initialiser les éléments d'un tableau de réels avec 0.0
 * \param[out] tab tableau à initialiser
 * \param[in] taille nombre d'éléments du tableau
 * \pre taille <= CAPACITE
 */
void initialiser(t_tableau tab, int taille){
    assert(taille <= CAPACITE);
    for (int i = 0; i <= taille; i++) {
        tab[i]=0.0;
    }
}

/**
 * \brief le tableau est-il vide ?
 * \param[in out] tab tableau à tester
 * \param[in] taille nombre d'éléments du tableau
 * \pre taille <= CAPACITE
 */
bool est_vide(t_tableau tab, int taille){
    assert(taille <= CAPACITE);
    bool vide=true;
    for (int i = 0; i <= taille; i++) {
        if (tab[i]!=0.0) {
            vide=false;
        }
    }
    return vide;
}

int main(void){
    t_tableau T;
    //Initialiser les éléments d'une variable tableau à 0.0
    initialiser(T,10);
    //Vérifier avec assert que tous les éléments vallent bien 0.0
    assert(est_vide(T,10));

    return EXIT_SUCCESS;
}
```

Chaines de caractères en C

Les chaines de caractères sont des **tableaux de caractères** :

```
char[10] ma_chaine; // un tableau de caractères de taille 10.
    char[] mon_nom = "Jaffres-Runser"; // initialise le tableau avec une chaine constante
```

La bibliothèque `string.h` permet de manipuler les chaines de caractères :

- `strlen(s)` retourne le nombre de caractères de la chaine
- `strcpy(s1, s2)` recopie le contenu de `s1` dans `s2`. Attention, il faut que `s2` ait une capacité suffisante !
- `strcat` concatène deux chaines, etc.

Exécuter l'exemple suivant :

In [14]:

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

int main(void){
    char mon_nom[] = "Jaffres-Runser";
    printf("Longueur de '%s' : %lu caractères\n", mon_nom, strlen(mon_nom));
    printf("Taille du tableau : %lu éléments \n", sizeof(mon_nom));
    printf("dernier élément : '%i' \n", mon_nom[sizeof(mon_nom)-1]);
    return EXIT_SUCCESS;
}
```

```
Longueur de 'Jaffres-Runser' : 14 caractères
Taille du tableau : 15 éléments
```

Le dernier caractère d'une chaine est le caractère `\0` de code ascii 0.

> **Règle** : une chaine de caractère se termine toujours par le caractère `\0`.

Type pointeur et adresse mémoire

Adresse mémoire

En C, il est possible de connaitre l'adresse mémoire à laquelle est stockée une variable `var1` avec l'opérateur unaire `&` (éperluette) :

> &var1

Déclaration d'un pointeur

Il est possible d'enregistrer cette adresse dans une variable de type **pointeur**. Une variable de type pointeur est aussi communément appelée pointeur. Elle enregistre la référence (i.e. l'adresse) d'une variable ou d'une donnée.

Pour déclarer un pointeur en C, il faut connaitre le type de la donnée qui sera enregistrée à cette adresse. Par exemple, on peut déclarer un pointeur sur une variable de type `entier` ou un pointeur sur une variable de type `double`. Pour déclarer un pointeur, on met * devant le nom de la variable :

> `type_pointé* pointeur;`

Quelques exemples :

```
int* ptr_int; // déclaration du pointeur ptr_int sur un entier
```

```
int* ptr_int; // déclaration du pointeur ptr_int sur un entier
    double* ptr_dbl; // déclaration du pointeur ptr_dbl sur un double
    char* ptr_char; // un pointeur sur un caractère
    int** ptr_ptr_int; // un pointeur sur un pointeur, qui pointe sur un entier
```

> Note : il peut y avoir un espace avant l'opérateur *.

Initialisation d'un pointeur

Comme pour n'importe quelle déclaration de variable en C, le pointeur n'est pas initialisé à une valeur par défaut à sa déclaration. En d'autres termes, l'adresse enregistrée n'a aucun sens, elle est aléatoire.

On doit initialiser un pointeur soit avec :

- Le pointeur `NULL` (élément neutre des adresses possibles) :

```
double* ptr_d = NULL;
```

- Soit avec l'adresse mémoire d'une variable du bon type:

```
int var1 = 10;
  int* ptr_int = &var1; // initialisation avec l'adresse de la variable var1
```

- Ou avec la valeur d'un pointeur de même type

```
int* ptr_int_2 = ptr_int;
```

> **Si l'adresse est n'est pas connue au moment de la déclaration, il faut toujours initialiser le pointeur à `NULL`**

Accès à la donnée pointée

Pour accéder à la variable pointée, on utilise aussi l'opérateur * placé avant l'identificateur.

```
*ptr_int = 25
```

Quelques exemples :

```
int var1 = 10;
    int* ptr_int = &var1;
    *ptr_int = 20 // On modifie ici la variable var1, qui vaudra 20 par la suite.
    // Déclaration et initialisation d'un nouvel entier var2 avec la donnée référencée par le pointeur ptr_int
    int var2 = *ptr_int ;
    assert(var2 == 20);
```

Affectation de pointeurs

Affecter un pointeur `p1` à un pointeur `p2`, comme pour toute affectation, recopie l'adresse `p1` dans `p2`. Les deux pointeurs référencent alors la même zone mémoire.

Quelques exemples :

```
int var1 = 10;
    int* p1 = &var1;
    int* p2 = p1;
    // A cet instant, on peut modifier le contenu de var1 en passant par p1 ou par p2.
    *p1 = 100; //var1 vaut 100
    *p2 = 1000; //var1 vaut 1000 maintenant !
```

Exercice 16 : Manipulation de pointeurs

[16.1] Compiler le programme suivant.

- Qu'observez-vous pour le premier affichage ? Le résultat dépend du compilateur, du système, etc.
- Qu'observez-vous pour le second affichage ?

In [19]:

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

int main(){
    int d1 = 1;
    int d2 = 4;
    int* p_1 ;
    int* p_2 ;
    printf("*p_1 = %d, *p_2 = %d\n", *p_1, *p_2);
    printf("p_1 = %p, p_2 = %p", p_1, p_2);
    return EXIT_SUCCESS;
}
```

```
[C kernel] Executable exited with code -11
```

[16.2] Modifier ce programme pour que `p_1` et `p_2`pointent respectivement sur `d1` et `d2`.

In [20]:

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

int main(){
    int d1 = 1;
    int d2 = 4;
    int* p_1 =&d1;
    int* p_2 =&d2;
    printf("*p_1 = %d, *p_2 = %d", *p_1, *p_2);
    return EXIT_SUCCESS;
}
```

```
*p_1 = 1, *p_2 = 4
```

[16.3] Compléter le programme pour échanger les entiers pointés par `p1` et `p2`. Après initialisation des pointeurs, on n'accèdera aux entiers qu'à travers des pointeurs.

In [26]:

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

int main(){
    int d1 = 1;
    int d2 = 4;
    int* p_1=&d1 ;
    int* p_2=&d2 ;
    printf("Avant échange : *p_1 = %d, *p_2 = %d\n", *p_1, *p_2);

    int* tmp=p_1;
    p_1=p_2;
    p_2=tmp;

    printf("Après échange : *p_1 = %d, *p_2 = %d", *p_1, *p_2);
    return EXIT_SUCCESS;
}
```

```
Avant échange : *p_1 = 1, *p_2 = 4
Après échange : *p_1 = 4, *p_2 = 1
```

[16.4] Qu'en est-il des données enregistrées dans `d1` et `d2` ? Ont-elles changé ?

Enregistrement et pointeurs

On suppose le type enregistrement `point` suivant :

```c
struct point {
        int x;
        int y;
    };
    typedef struct point point;
```

Un pointeur sur un enregistrement permet d'accéder au contenu de l'enregistrement de deux manières :

1. Avec les opérateurs *** et .**

```c
point pt1;
 struct point * ptr_point = &pt1;
 (*ptr_point).x = 12;
 (*ptr_point).y = 0;
```

1. Avec l'opérateur **->**

```c
point pt1;
 struct point * ptr_point = &pt1;
 ptr_point->x = 12;
 ptr_point->y = 0;
```

> **Règle** : **Il faut utiliser la notation ->**

Tableau et pointeurs

En C, le nom de la variable tableau est **l'identifiant d'un pointeur sur la première case** du tableau. On peut donc accéder au contenu de la première case par ce pointeur. Par exemple :

```c
int tab[] = {1, 4, 8, 16};
    // tab est un pointeur sur la case 0
    *tab = 20; // équivalent à tab[0] = 20
```

Il est possible d'accéder à la case suivante **en incrémentant de 1 le pointeur** (arithmétique des pointeurs) :

```c
*(tab+1) = 40; // équivalent à tab[1] = 40
    // déclaration d'un pointeur sur la 4e case du tableau
    int* ptr = tab+3;
    assert(*ptr == 16);
```

L'opérateur **-** permet de se déplacer vers la gauche dans le tableau :

```c
ptr = ptr-2;
    //ptr pointe sur la 2e case du tableau
    assert(*ptr == 40);
```

Exercice 17

[17.1] Ré-écrire la fonction `initialiser` de l'exercice 15 avec la notation pointeur du tableau et l'arithmétique associée.

In [1]:

```c
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>

#define CAPACITE 20
// Definition du type tableau
typedef int t_tableau[10];

/**
 * \brief Initialiser les éléments d'un tableau de réels avec 0.0
 * \param[out] tab tableau à initialiser
 * \param[in] taille nombre d'éléments du tableau
 * \pre taille <= CAPACITE
 */
void initialiser(t_tableau tab, int taille){
    assert(taille <= CAPACITE);
    for (int i = 0; i <= taille; i++) {
        *(tab+i)=0.0;
    }
}

int main(void){
    t_tableau T;
    //Initialiser les éléments d'une variable tableau à 0.0
    initialiser(T,5);

    return EXIT_SUCCESS;
}
```

Les sous-programmes

Le langage C ne permet pas de différencier les fonctions des procédures algorithmiques. Le seul sous-programme utilisable est la fonction :

```
type_retour identificateur_fonction ( type_param1 id_param1, type_param2 id_param2,
...)
```

Chaque paramètre formel est typé et **passé par valeur** (mode IN algorithmique). Ainsi, l'appel à une fonction sur des paramètres réels variables ne modifie pas la donnée. Par contre, les instructions de la fonction connaissent la donnée (valeur) et peuvent la manipuler pour fournir l'unique résulat retourné via le type retour.

Illustration du passage par valeur

Exécuter le programme suivant :

In [2]:

```c
#include <stdio.h>

// Definition d'une fonction f1
int f1(int valeur) {
    printf("   valeur au début de f1 : %i \n", valeur);
    valeur = 0;
    printf("   valeur à la fin de f1 : %i \n", valeur);
    return valeur;
}

int main(){
```

```
    int donnee = 20;
    printf("donnee dans main() avant f1 : %i  \n", donnee);
    int donnee_retournee = f1(donnee); // la fonction utilise la valeur de donnee
    printf("donnee dans main() après f1 : %i  \n", donnee);
    printf("donnee_retournee dans main() : %i \n", donnee_retournee);
}
```

```
donnee dans main() avant f1 : 20
   valeur au début de f1 : 20
   valeur à la fin de f1 : 0
donnee dans main() après f1 : 20
donnee_retournee dans main() : 0
```

Passage par adresse

Pour pouvoir modifier le contenu d'une variable passée en paramètre d'une fonction (mode OUT ou IN OUT algorithmique), on fournit à la fonction **l'adresse de la variable**. Connaissant l'adresse, la fonction pourra alors modifier sa valeur.

On passe l'adresse d'une variable à une fonction à l'aide d'un pointeur. Pour se faire, il faut déclarer la fonction avec des paramètres formels qui sont des pointeurs.

Petite illustration du **passage de paramètres par adresse** :

In [3]:

```
#include <stdio.h>

// Definition d'une fonction f1 avec un paramètre pointeur qui
// peut enregistrer d'adresse d'un entier
int f1(int* valeur) {
    *valeur += *valeur; // Accès à la variable au travers du pointeur.
    printf("valeur dans f1 après incrémentation : %i \n", *valeur);
    return *valeur;
}

int main(){
    int donnee = 20;
    // Pour utiliser la fonction, on donne l'adresse de la variable
    int nouvelle_donnee = f1(&donnee);
    printf("donnee dans main() après incrémentation : %i  \n", donnee);
    printf("nouvelle_donne dans main() : %i \n", nouvelle_donnee);
}
```

```
valeur dans f1 après incrémentation : 40
donnee dans main() après incrémentation : 40
nouvelle_donne dans main() : 40
```

Pour ce passage par adresse, il faut :

- utiliser des pointeurs pour définir les paramètres formels
- dans les instructions de la fonction, accéder à la donnée pointée avec l'opérateur *
- lors de l'appel du sous-programme, fournir une adresse valide d'une variable à modifier avec l'opérateur &.

Passage d'un paramètre de type tableau en C

Un tableau étant un pointeur, le passage par valeur d'un paramètre tableau offre naturellement un passage en mode `in out`. Ainsi :

- Il n'est pas nécessaire de passer un tableau par adresse si on souhaite in mode `in out`.
- Si on veut définir un mode `in`, il faut empêche la modification en utilisant `const` :

```
/*
 * \brief Affiche un tableau de taille éléments
 * \param[in] tab tableau à afficher
 * \param[in] taille nombre d'éléments du tableau
 * \pre taille <= CAPACITE
 */
void afficher_tab (const int[] tab, int taille)
```

Exercice 18 : passage par adresse

Compléter le programme suivant en répondant aux questions suivantes :

[18.1] Définir le type `t_note`, caractérisé par sa valeur et son coefficient. Par exemple, la note de 14 a été obtenue pour le BE d'algorithmique et programmation qui compte coefficient 1/4.

[18.2] Définir le type `t_tab_notes` qui permet d'enregistrer 5 notes.

[18.3] Compléter et corriger la fonction qui initialise une note à partir de sa valeur et de son coefficient.

[18.4] Compléter et corriger la fonction qui calcule la moyenne des notes d'un tableau de notes.

> **Attention** il faut respecter le mode `in` du paramètre tableau.

In [2]:

```
#include <stdlib.h>
#include <stdio.h>
#include <assert.h>
#include <stdbool.h>
//typedef struct {float note;float coeff} t_note;
// Definition du type t_note
struct t_note{
    float note;
    float coeff;
};
typedef struct t_note t_note;
// Definition d'un tableau de notes t_tab_notes de 5 éléments.
typedef t_note t_tab_notes[5];

/**
 * \brief Initialiser une note
 * \param[out] note note à initialiser
 * \param[in] valeur nombre de points
 * \param[in] coef coefficient
 * \pre valeur <= 20 && valeur >= 0
 * \pre coef <= 1 && coef >= 0
 */
void initialiser_note(t_note* note, float valeur, float coef){
    assert(valeur <= 20 && valeur >= 0);
    assert(coef <= 20 && coef >= 0);
    //(*note).note=valeur;
    //(*note).coeff=coef;

    note->note= valeur;
    note->coeff=coef;
}
/**
 * \brief Calculer la moyenne des notes du tableau
 * \param[in] tab_notes tableau de nodes
 * \param[in] nb_notes nombre de notes
 */
float moyenne(t_tab_notes tab_notes, int nb_notes){
    float moy;
    moy=0.0;
    for (int i=0;i<=nb_notes-1;i++){
    moy+=(tab_notes[i].note)*(tab_notes[i].coeff);
}
    return moy;
}


int main(void){
    t_tab_notes notes;
```

```
    //Initialiser les éléments d'une variable tableau à 0.0
    initialiser_note(&notes[0], 10, 0.2);
    initialiser_note(&notes[1], 1, 0.3);
    initialiser_note(&notes[2], 12, 0.5);

    //Calculer la moyenne des 3 notes
    float moy;
    moy= moyenne(notes, 3);
    printf("La moyenne est %f",moy);
    assert(moy == 8.3);
    assert(moy == 10*0.2 + 1*0.3 +12*0.5);

    return EXIT_SUCCESS;
}
```

```
La moyenne est 8.300000
```

```
tmppzmrxyzt.out: /tmp/tmpd88j5mz4.c:59: main: Assertion `moy == 8.3' failed.
[C kernel] Executable exited with code -6
```

Arguments de la ligne de commande

Il est possible de fournir des arguments pour paramétrer l'exécution d'un programme. On pourra par exemple personnaliser le message affiché à l'utilisateur dans le `premier_programme` en exécutant :

```
./premier_programme Michel
```

pour qu'il présente l'affichage suivant

********** **** Bienvenue au DU IoT Michel ***** **********

Pour se faire, il faut déclarer la signature du programme principal avec les paramètres `argc` et `argv[]` :

In [ ]:

```
#include <stdlib.h>

int main(int argc, char* argv[]){
    return EXIT_SUCCESS;
}
```

> `int argc` : nombre d'arguments.
>
> `char* argv[]` : tableau de chaines de caractères

La chaîne à l'indice 0 existe toujours et contient le nom de l'exécutable. Les autres éventuelles chaines listent les arguments dans l'ordre où ils sont présentés.

Exercice 19 : Lister les arguments de la ligne de commande

[19.1] Ecrire un programme qui permet d'afficher les arguments de la ligne de commande.

In [97]:

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

```c
#include <stdio.h>

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

    return EXIT_SUCCESS;
}
```

```
Les arguments sont :
```

Exercices BILAN 2

> **ATTENTION**
>
> - Les deux exercices bilan suivants **sont à rendre à votre intervenant** de TP sous SVN.

Exercice 1 : Portée et masquage des variables.

Le programme fourni suivant compile sans erreur, même avec l'option -Wall. On répondra aux questions suivantes dans **un fichier texte prévu à cet effet sous SVN**, sans compiler ni exécuter le programme.

```c
// Comprendre la portée des variables,
// et le masquage.

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

int main() {
    int n = 10;
    int *p = NULL;
    printf("&n = %p\n", (void *) &n);
    printf("p = %p\n", (void *) p);
    {
        int a = 5;
        p = &a;
        printf("&a = %p\n", (void *) &a);
        printf("p = %p\n", (void *) p);
        printf("*p = %d\n", *p);
    }
    printf("p = %p\n", (void *) p);
    printf("*p = %d\n", *p);
    {
        int n = 7;
        printf("n = %d\n", n);
        printf("&n = %p\n", (void *) &n);
    }
    printf("p = %p\n", (void *) p);
    printf("*p = %d\n", *p);
    printf("n = %d\n", n);
    {
        double r = 11;
        printf("r = %f\n", r);
        printf("&r = %p\n", (void *) &r);
    }
    printf("*p = %d\n", *p);

    return EXIT_SUCCESS;
}
```

[B1.1] Quelle est la portée de chaque variable déclarée ? Pour chaque variable, on donnera le numéro de ligne où commence et se termine sa portée.

[B1.2] Y a-t-il un exemple de masquage de variable dans ce programme ?

[B1.3] Peut-on savoir ce que devrait afficher l'exécution de ce programme ?

[B1.4] Même s'il compile sans erreur, ce programme est faux. Pourquoi ?

[B1.5] La valeur de `p` change-t-elle après l'initialisation de la ligne 14 ?

[B1.6] Que se passerait-il si on modifiait `*p` après la ligne 19 ?

Exercice 2 : Définition d'une monnaie.

Dans cet exercice nous nous intéressons à la notion de monnaie. Une monnaie est caractérisée par sa valeur et sa devise. Nous considérerons que la valeur est réelle et que la devise est représentée par un caractère. La valeur d'une monnaie doit toujours être positive. Par exemple, la monnaie «cinq euros» sera représentée par la valeur 5 et le caractère « e », « dix dollars » par la valeur 10 et le caractère « $ ».

Vos réponses sont attendues dans un fichier sous SVN prévu à cet effet.

[B2.1] Définir le type `monnaie`.

[B2.2] Écrire un sous-programme qui initialise une monnaie à partir d'une valeur et d'une devise. La valeur doit être strictement positive. On utilisera la programmation par contrat pour le spécifier.

[B2.3] Écrire un sous-programme qui permet d'ajouter à une monnaie la valeur d'une autre monnaie. Les deux monnaies doivent avoir même devise pour que l'opération soit possible. Par exemple, si on ajoute une monnaie `m1` qui vaut 5 euros à une monnaie `m2` qui vaut 7 euros alors `m2` vaut 12 euros après l'opération et `m1` est inchangée. Si les deux monnaies n'ont pas la même devise, l'opération n'aura pas lieu.

On utilisera la programmation défensive et un code d'erreur, ici un booléen (valeur retournée) indiquera si l'opération a été réalisée ou non.

[B2.4] Écrire des sous-programmes de test des sous-programmes définis sur le type `monnaie`.

[B2.5] Écrire un programme principal qui :

1. déclare un tableau de 5 monnaies appelé porte_monnaie (5 doit être une constante préprocesseur),
2. initialise chaque élément du tableau en demandant la valeur et la devise d'une monnaie à l'utilisateur,
3. affiche la somme de toutes les monnaies qui sont dans une devise demandée à l'utilisateur.

Il est bien entendu possible de créer des sous-programmes issus d'un raffinement du programme principal.