Programmation en langage C : compétences de base

A l'attention des étudiants de L2, unité d'enseignement « Programmation en C et C++ » de l'université de Bourgogne.

Par Olivier Bailleux, maître de conférences HDR en informatique.

`olivier.bailleux@u-bourgogne.fr`

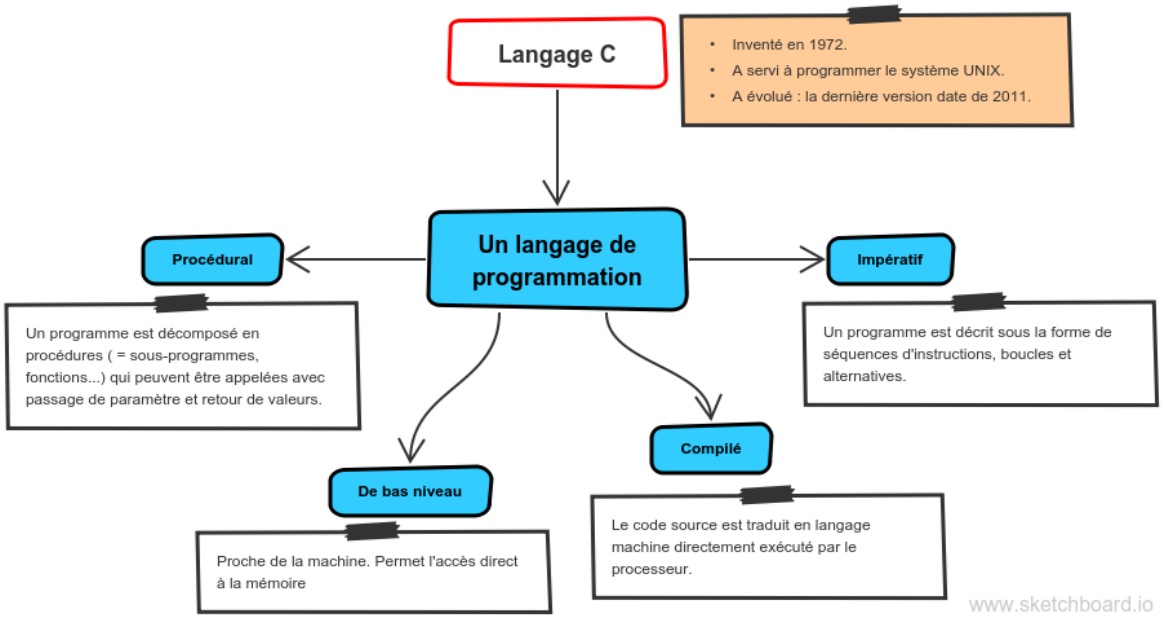
Version de décembre 2016.

Cette version est en cours de réalisation. Elle comporte quelques coquilles et aspérités.

Introduction

Qu'est-ce que le langage C ?

C'est un langage de programmation impératif, procédural et compilé conçu pour réaliser des programmes rapides et économes en ressources. Mais attention, programmer en C est une aventure pleine de pièges. Il faut beaucoup d'attention, de rigueur, et une certaine expérience pour éviter les erreurs liées notamment à une mauvaise gestion de la mémoire.



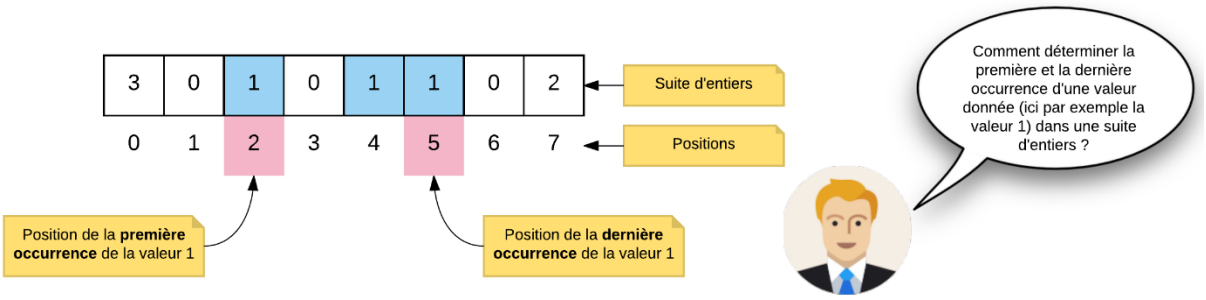
Prérequis

Pour suivre ce cours, vous devez avoir une première expérience de programmation avec un langage procédural ou orienté objet et être capable d'analyser un problème ou un besoin de manière à déterminer les structures de données et les algorithmes adaptés.

Exemple

Problème

Dans une suite de n entiers A_0, A_1, …, A_{n-1}, comment déterminer les positions de la première et de la dernière occurrence d'une valeur x donnée ?



Solution

Les entiers sont placés dans les cellules T[0] à T[n-1] d'un tableau T. La position de la première occurrence de la valeur x (si applicable) sera placée dans une variable A, et celle de la dernière occurrence de la valeur x (si applicable) sera placée dans une variable B.

Voilà, nous avons choisi une *structure de donnée* (en l'occurrence un tableau) pour représenter les informations à traiter. Maintenant, il faut réfléchir à un *algorithme* capable de faire le job. Un algorithme peut être formulé en langage naturel, comme ceci :

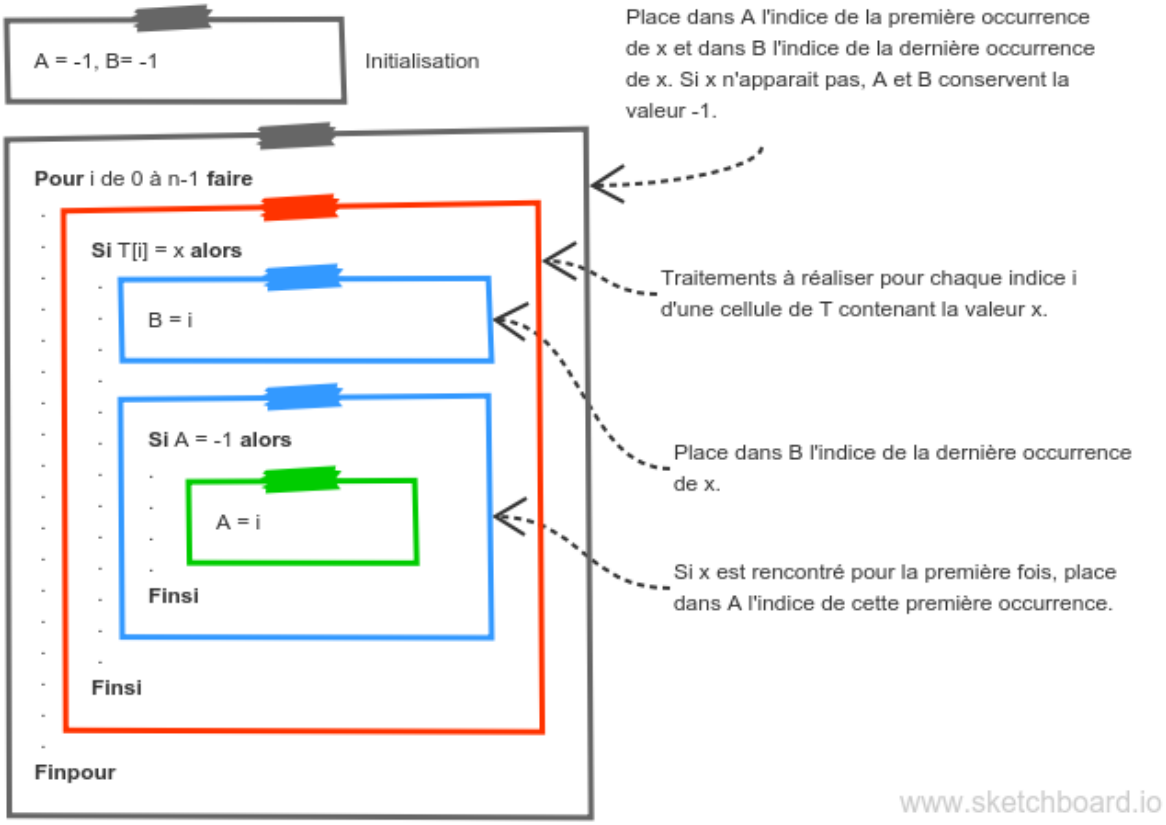
```
Les variables A et B sont initialisée à -1. Ensuite, on
parcourt le tableau en faisant varier une variable i entre
0 et n-1. À chaque fois que T[i] contient x, on place i
dans B et si A vaut -1, mais seulement dans ce cas, on
place également i dans A.
```

Il peut être également présenté de manière plus structurée, comme ceci :

```
A ← -1, B ← -1
Pour i de 0 à n-1 faire
    Si T[i] = x alors
        B ← i
        Si A = -1 alors
            A ← i
        Finsi
    Finsi
Finpour
```

Il n'y a pas de méthode miracle pour concevoir un algorithme. Chaque cas est particulier, mais il y a des principes généraux à respecter.

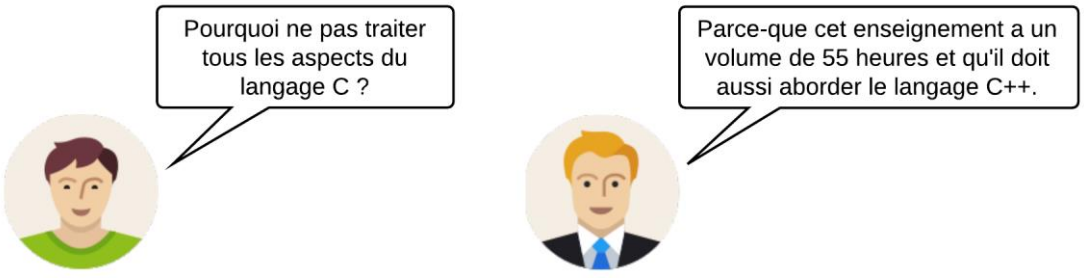
1. Je dois d'abord parfaitement comprendre ce qu'est censé faire l'algorithme avant de commencer à rechercher comment il va le faire.
2. Tout traitement qui n'est pas réalisable de manière immédiate et évidente doit être décomposé en parties plus simples.
3. Si j'ai du mal à faire cette décomposition, j'essaie de réaliser le traitement souhaité à la main, sur un exemple, en respectant la règle suivante : je ne peux voir ou modifier qu'un nombre limité de variable à la fois, généralement pas plus de trois. Par exemple, si les données à traiter sont dans un tableau, je n'ai pas le droit de regarder tout le tableau, mais seulement une ou deux valeurs en même temps.
4. Mon algorithme doit être constitué de blocs successifs et/ou imbriqués les un dans les autres, et chaque bloc doit avoir un rôle parfaitement déterminé.



Objectifs pédagogiques

L'objectif de l'enseignement en langage C que vous allez suivre est de vous permettre d'acquérir des compétences réparties en trois pools. Dans un deuxième temps, nous aborderons le langage C++, pour lequel il y aura deux pools de compétences supplémentaires à acquérir.

Attention, ces trois pools de compétences ne couvrent pas toutes les possibilités du langage C. De nombreuses notions, telles que par exemple la gestion des fichiers, les tableaux multidimensionnels natifs, les unions et énumérations, le préprocesseur, les caractères étendus, ne seront pas abordée ou ne seront que survolées.



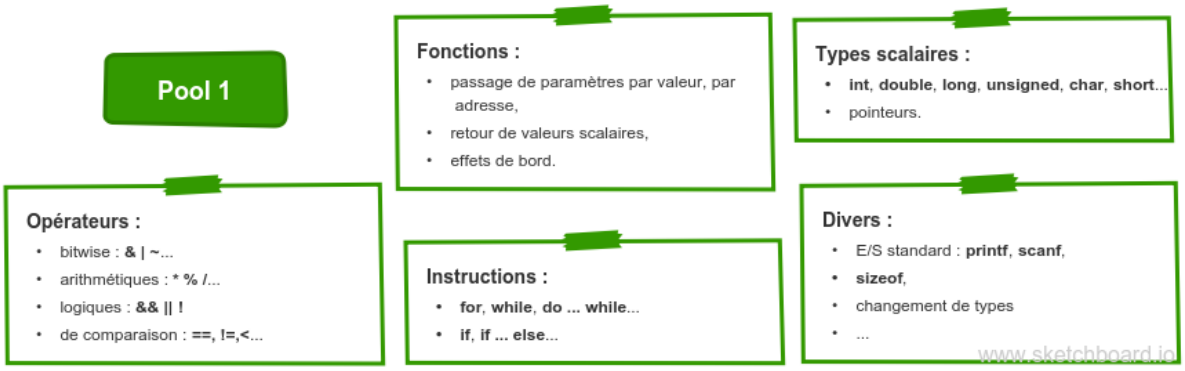
Mais si vous maîtrisez les bases qui vous seront donnée lors de cet enseignement, alors vous serez parfaitement capable d'apprendre par vous-même les notions que nous n'aurons pas abordées.

Pool1 : Programmation avec des types scalaires

Vous devrez maîtriser les représentations en mémoire des différentes données scalaires et pointeurs et être capable d'utiliser à bon escient les types correspondants pour réaliser des programmes ou

parties de programmes, décomposés si applicable en différentes fonctions, réalisant des traitements spécifiés par des énoncé en langage naturel.

Vous devez aussi être capable de prévoir le comportement et le résultat de l'exécution d'une partie de programme écrite en langage C utilisant les notions décrites ci-dessus, et de dessiner les représentations en mémoire des données traitées en matérialisant par des flèches les liens entre pointeurs et données pointées.



Pool2 : Tableaux et chaînes de caractères

Vous devrez maîtriser la déclaration et l'utilisation de tableaux et de chaînes de caractères, être capable de réaliser des fonctions acceptant en paramètres des adresses de tableaux et de chaînes, capables notamment de modifier les tableaux où chaînes pointés par ces adresses. Vous devrez savoir utiliser à bon escient quelques fonctions standards réalisant des traitements sur des chaînes de caractères (copie, concaténation, comparaison…) et être capables de réaliser des fonctions qui créent ou modifient des chaînes de caractères.

Vous devez aussi être capable de prévoir le comportement et le résultat de l'exécution d'une partie de programme écrite en langage C utilisant les notions décrites ci-dessus, et de dessiner les représentations en mémoire des données traitées en matérialisant par des flèches les liens entre pointeurs et données pointées.

Pool3 : Variables dynamiques et structures

Vous devez être capables de définir des structures contenant différents types de champs, y compris des tableaux, des structures et des pointeurs sur des structures, ainsi que des fonctions acceptant en paramètres des structures ou des pointeurs sur des structures et retournant des structures.

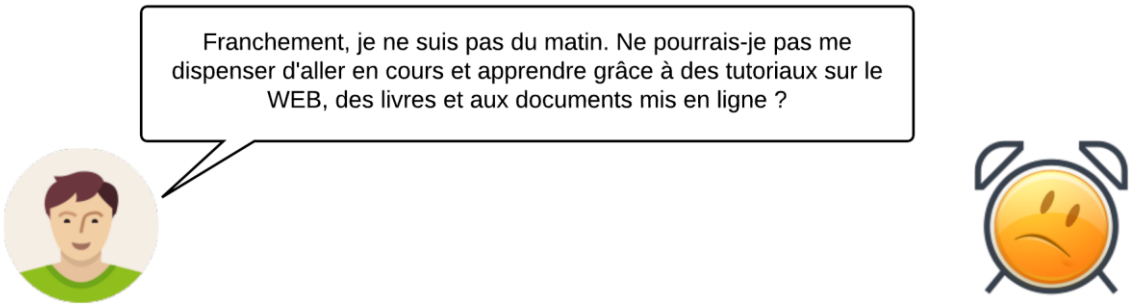
Vous devez maîtriser la création, l'utilisation et la destruction de variables dynamiques de types structure ou tableau (à une ou deux dimensions) représentées par des blocs mémoire situé dans le tas, et savoir réaliser des fonctions qui acceptent en paramètres et/ou retournent des adresses mémoire

de telles variables. Vous devez être capable de mettre en œuvre de telles fonction en réalisant des programmes ne provoquant pas de fuite mémoire ni d'accident mémoire.

Vous devez être capable de prévoir le comportement et le résultat de tout programme ou partie de programme utilisant des variables dynamiques, notamment de type tableau et structures, et de dessiner une représentation exacte et précise des données situées en mémoire dans la pile, dans le tas, et dans la zone des variables globale à n'importe quel moment de l'exécution.

Vous devez être en mesure de déterminer si un programme ou une partie de programme risque de provoquer, lors de son exécution, des fuites mémoire ou des accidents mémoire.

Auto apprentissage



Attention. Il est possible d'acquérir les compétences susmentionnées de manière autodidacte, en particulier si vous maîtrisez déjà certaines d'entre-elles…

MAIS

Il est très fortement déconseillé de sécher les cours pour plusieurs raisons :

1. Cet enseignement est conçu pour être suivi en mode présentiel, et les activités proposées en cours sont essentielles à une bonne assimilation des notions abordées.
2. Même des programmeurs ayant une certaine expérience du langage C peuvent se faire piéger par certaines situations qui seront présentées et expliquées en cours.
3. L'enseignant qui fait le cours a une grande expérience de la programmation en langage C.
4. Cet enseignant réalise aussi le sujet d'examen. En assistant au cours, vous pourrez cerner précisément les éléments considérés comme importants, voire essentiels, par cet enseignant, et qu'il faut donc parfaitement maîtriser pour avoir une bonne note à l'examen.
5. Beaucoup d'informations, de conseils dispensés en cours n'apparaissent pas dans les documents mis en ligne.

6. En apprenant par vous-même, vous mettrez plus de temps à maîtriser l'ensemble des compétences à assimiler, parce qu'il vous manquera quelques explications, quelques « clés » qui facilitent considérablement la compréhension de certaines notions.
7. En cours, vous pouvez poser directement des questions à l'enseignant.
8. Le cours est le seul moment où l'enseignant responsable peut s'adresser à l'ensemble des étudiants suivant une unité d'enseignement. En y assistant, vous êtes sûr de ne manquer aucune information importante sur l'organisation des enseignements et les modalités d'évaluation, notamment.
9. L'enseignant est un être humain. Sa performance, son efficacité dépendent de sa motivation, et sa motivation dépend de l'assiduité des étudiants et de leur comportement.
10. Beaucoup de gens triment pour payer des impôts qui vous permettent d'accéder aux études supérieures à faible coût (par comparaison avec d'autres pays). C'est aussi une marque de respect envers eux et leur travail que de faire votre possible pour réussir en assistant à tous les enseignements. Dans les pays où les études sont très chères, il y a très peu d'absentéisme en cours. Un enseignement a-t-il moins de valeur s'il est gratuit ?

Pool 1

Ce pool de compétences inclus les base de la programmation en langage C avec des données scalaires, c'est-à-dire des entiers, des nombres en virgule flottante, des Booléens, des caractères, des pointeurs, en fait toutes les données élémentaires.

Les types scalaires

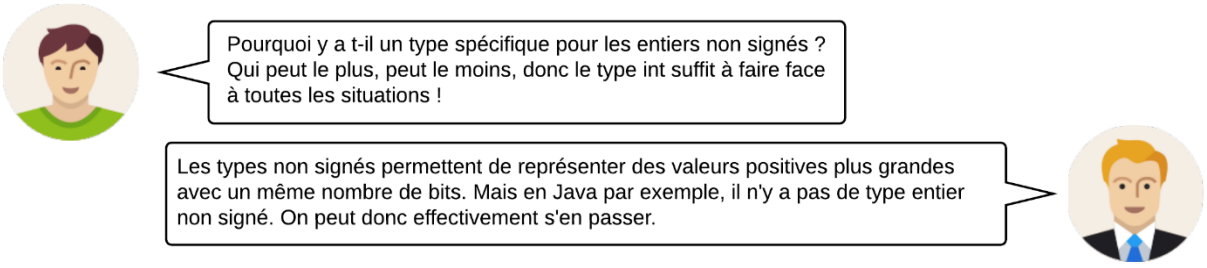
Les entiers

En langage C, les entiers sont utilisés pour représenter non seulement des nombres positifs ou négatifs, mais aussi des caractères et des valeurs Booléennes. Il existe de nombreux types pour représenter les entiers, qui diffèrent selon deux critères : la taille de leur représentation en mémoire et le fait qu'ils sont signés ou non signés.

Les entiers standards

Les types de base pour représenter les entiers sont le type `int`, et le type `unsigned int`. Ces deux types ont des représentations en mémoire de mêmes tailles, mais cette taille peut changer en fonction du compilateur et de l'architecture matérielle de la machine utilisée.

- Une valeur de type `int` ou `unsigned int` est représentée en mémoire par au moins deux octets. Les compilateurs récents produisant du code pour architectures 32 ou 64 bits utilisent 4 octets.
- L'instruction `sizeof` permet de connaître la taille en octets de la représentation en mémoire d'un type de donnée. Par exemple, dans un programme destiné à être exécuté en environnement Windows, Linux, ou Mac OS, `sizeof(int)` retourne la valeur 4.

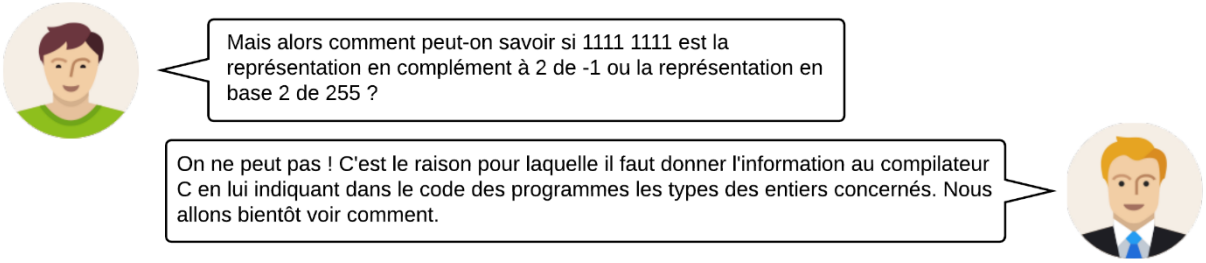


Un entier signé est représenté en mémoire en complément à 2, alors qu'un entier signé est représenté en base 2. Prenons un exemple sur 4 bits. En base 2, la valeur 12 s'écrit 1100 (8 +4). En complément à 2, seules les valeurs comprises entre -8 et 7 peuvent être représentées en complément à 2. On ne peut pas représenter la valeur 12. Le mot binaire 1100, qui représente 12 en base 2, représente la valeur négative -4 en complément à 2.

Ces notions sont censées être connues par les étudiants auxquels ce cours est destiné, voici juste un rappel de quelques propriétés utiles :

- Sur n bits, on peut représenter en base deux les valeurs comprises entre 0 et 2^n-1, et en complément à 2 les valeurs comprises entre -2^{n-1} et 2^{n-1}-1.
- La représentation en complémentation à deux un entier positif est la même que sa représentation en base deux.
- La représentation en complément à deux sur n bits d'une entier négatif x est la même que la représentation en base deux sur n bits de l'entier 2^n+x.

- Pour obtenir rapidement une complémentation à deux, on conserve tous les bits en partant de la droite jusqu'au premier 1 inclus, et on complémente tous les autres bits. Par exemple avec 1100 on obtient 0100.
- En complément à deux, dans toutes les représentations des entiers négatifs le bit le plus à gauche est à 1, alors qu'il est à 0 dans toutes les représentations des entiers positifs ou nul.



Justement, examinons un premier programme qui calcule la somme deux entiers saisis au clavier et affiche le résultat.

```c
#include <stdio.h>

int main()
{
    int a = 12;

    scanf("%d", &a);

    printf("%d\n",2*a);

    return 0;
}
```

Fichier d'entête nécessaire pour utiliser printf et scanf

Déclaration d'une variable de type int.

Saisie clavier d'un entier en base 10, qui est placé dans a

Affichage du résultat de 2 * a en base 10

Beaucoup de choses dans ce programme ne seront totalement élucidées que plus tard. La première ligne, par exemple, est nécessaire pour que ce code puisse être compilé. Le reste du programme définit une fonction principale appelée `main`. C'est là que se trouvent les premières instructions exécutées par le programme, et dans ce cas particulier les seules.

C'est la première ligne de la fonction `main` qui est importante dans cet exemple. Elle déclare une variable qui va contenir un entier signé, et place dans cette variable une valeur initiale.

Les lignes suivantes permettent de saisir une valeur au clavier, de la placer dans la variable `a`, puis de la multiplier par 2 et l'afficher à l'écran. Les détails seront expliqués plus bas. Quant à la dernière ligne, elle a pour but de dire au système d'exploitation que l'exécution du programme s'est déroulée sans incident (code d'erreur 0).

Avant de passer à la suite, sachez que le type `int` peut aussi s'écrire `signed` ou `signed int` et que le type `unsigned int` peut aussi s'écrire juste `unsigned`.

Les entiers longs

Les types `long` et `unsigned long` (aussi notés `long int`, `signed long`, `signed long int` et `unsigned long int` respectivement) permettent d'utiliser des entiers qui *avec certaines architectures*, ont une représentation en mémoire plus grande que celle des `int` et `unsigned int`. Mais avec les architectures à 32 bits et même certaines architectures 64 bits, les `int` et `long` ont les mêmes représentations et le type `long` fait double emploi. On sait juste que les entiers de type `long` ont une taille mémoire d'au moins 4 octets, soit 32 bits.

Il existe également des types `long long` et `unsigned long long` qui garantissent une représentation des entiers sur au moins 8 octets, soient 64 bits.

Les entiers courts

Pour consommer moins de mémoire, et lorsque les valeurs à représenter sont suffisamment petites, on peut utiliser les types `short` et `unsigned short` (aussi notés `short int`, `signed short`, `signed short int` et `unsigned short int` respectivement). Les entiers ayant ces types sont représentés en mémoire sur au moins 16 bits, et généralement sur exactement 16 bits.

Les caractères

En langage C, les types `char` et `unsigned char` ont deux usages. Ils représentent des entiers pouvant être codés sur un seul octet, mais aussi des caractères imprimables identifiés par leurs code ASCII. Ainsi, les lignes de code ci-dessous sont parfaitement correctes.

```
char c1 = 'a';

char c2 = c1 + 1;

char c3 = -33;

unsigned char c4 = 255;
```

Le code ascii de ‘a’ est placé dans c1.

Le code ascii de ‘b’ est placé dans c2.

Si les caractères sont considérés comme des entiers, alors on peut additionner deux caractères en écrivant par exemple 'a' + 'z' ?

Tout à fait. Mais le résultat n'a pas vraiment de sens. Par contre il est parfois utile de **soustraire** des caractères. Par exemple que représente 'a' - 'A' + 'h' ?

Les constantes numériques entières

Il est possible de renseigner le type exact d’une constante numérique, et dans certains cas sa représentation (décimale, hexadécimale…). Voici quelques exemples :

`0x200`	Entier hexadécimal (valeur décimale : 512)
`0200`	Entier octal (base 8) (valeur décimale : 128)
`0b11110000`	Entier en base 2
`45u`	Entier au format unsigned
`45L`	Entier au format long

Les conversions entre types entiers

Que se passe-t-il si on place un caractère dans une variable de type `int`, ou un `unsigned int` dans une variable de type `int`, ou si on additionne un `int` et un `unsigned short`, par exemple ? Il y aurait de quoi faire un cours complet, donc nous nous contenterons de traiter seulement certains cas.

Si un `unsigned` et un `int` interviennent dans une même opération (par exemple une addition), la valeur de type `unsigned` est convertie en `int` et le résultat sera donc de type `int`. Cette conversion peut occasionner un résultat incohérent si la valeur `unsigned` n’est pas représentable avec la taille d’un `int` sous forme signée.

```
int a = -1;
unsigned int b = 5;
printf("%d\n", a+b);
```

Affiche -4

Si on fait intervenir dans une même opération des types différents représentables en `int` sans perte d'information (i.e., `int`, `char`, `unsigned char`, `short`, `unsigned short`), les types qui ne sont pas des `int` sont convertis en `int` avant le calcul et le résultat est de type `int`. Ce principe est appelé « règle de promotion des entiers ».

```
int a = 10;
char b = -25;
printf("%d\n",a+b);
```

Affiche -15

Si on place une valeur entière v dans une variable x ayant un type pouvant représenter cette valeur, la conversion n'engendre aucune perte d'information. Dans le cas contraire, si la taille de la représentation en mémoire de x est plus grande que celle de a, la représentation de x est tronquée et v recevra une valeur différente de x.

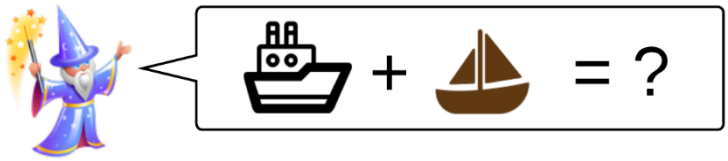
```
unsigned int a = 256;

char b = a;

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

Cette valeur est trop grande pour le type char.

La variable b reçoit la valeur 0.

Les flottants

Par « flottant », on entend nombre représenté en virgule flottante, c'est-à-dire sous la forme d'une mantisse et d'un exposant eux-mêmes codés en binaire. Nous ne rentrerons pas dans le détail de leur représentation en mémoire.



Les types flottants

Le langage C comporte trois types dédiés à la représentation de nombres en virgule flottante : `float` (simple précision), `double` (double précision) et `long double` (précision étendue). La taille en octets des données de chacun de ces types peut être obtenue, comme pour tous les types du langage C, avec l'instruction `sizeof`. Par exemple, avec un compilateur 32 bit sous Windows, la ligne de programme suivante affiche 4 8 12, ce qui indique qu'une valeur de type `float` occupe 4 octets en mémoire, une valeur de type `double` 8 octets, etc.

```
printf("%d %d %d\n",sizeof(float),sizeof(double),sizeof(long double));
```

Le premier argument de `printf` précise que les trois arguments suivants sont des entiers devant être afficher en base 10. Nous y reviendrons un peu plus tard.

Les constantes numériques

Quelques exemples de notations :

`12.4`	Nombre au format `double`
`12.4f`	Nombre au format `float`
`12.4l`	Nombre au format `long double`
`0.56e-4`	Notation scientifique

Les conversions de types

Les conversions entre types flottants tentent de préserver les valeurs lorsque c'est possible, éventuellement avec perte de précision. Par exemple, certaines valeurs de type `double` sont trop grandes pour être représentables en `float`. Le simple bon sens vous conduira à utiliser toujours le même type, sauf cas particuliers qui sortent du cadre de cet enseignement.

Les types flottants sont convertis en entiers en conservant la partie entière, lorsque le type de destination le permet, sinon le résultat n'est pas cohérent. La conversion des entiers en flottants tente de préserver la valeur de départ, mais occasionne parfois une perte de précision.

Voici deux exemples qui illustrent deux types de pièges liés à la conversion entre types entiers et flottants.

```
float x = 5000000000.0;
int y = x;
printf("%d\n",y);

y=2000000001;
x = y;
printf("%f\n",x);
```

Affiche -2147483648

Affiche 2000000000.000000

Les constantes symboliques

Il existe deux manières de définir des constantes en leur attribuant un identificateur (nom). En voici une illustration.

```
#define TIMEOUT 3600

const int MAX = 100;
```

Constante gérée par le préprocesseur.

Constante gérée par le compilateur.

La première ligne est prise en compte par un programme appelé *préprocesseur*, qui est lancé automatiquement avant toute compilation d'un fichier source C, et qui réalise des substitutions ou inclusions dans le code source. Dans l'exemple il remplace chaque occurrence du mot « TIMEOUT » par « 3600 ». Attention, il ne faut pas mettre de point-virgule à la fin de la ligne, sinon TIMEOUT serait remplacé par « 3600 ; ». Une erreur classique qui a conduit bien des débutants en langage C à s'arracher les cheveux.

La deuxième ligne consiste à déclarer une variable globale, avec le modificateur `const`, qui interdit toute modification en la transformant de fait en constante. La déclaration soit se faire à l'extérieur de toute définition de fonction.

Dans les deux cas, les déclarations de constantes doivent être situées dans le fichier source avant leur utilisation.

Les pointeurs

Un pointeur est une adresse mémoire. Pour l'instant, tous les pointeurs que nous utiliserons seront typés. Le type d'un pointeur nous renseigne (et renseigne le compilateur C) sur le type de la donnée pointée, c'est-à-dire de la donnée située dans la mémoire à l'adresse désignée par le pointeur. Voici un petit exemple introductif.

```
int x =10;
int* p = &x;
*p = *p + 1;
printf("%d\n",x);
```

L'adresse de la variable x est placée dans la variable p, qui est de type pointeur sur int.

On ajoute 1 à la valeur pointée par x.

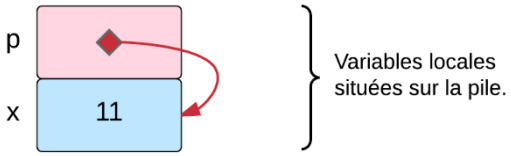
A l'affichage, on constate que x vaut 11.

Il y a plusieurs choses importantes dans cet exemple :

- La manière de désigner un type pointeur, avec une étoile *.
- La manière de récupérer l'adresse d'une variable avec une esperluette &.
- La manière d'accéder à une valeur pointée par un pointeur avec une étoile, mais utilisée différemment. On parle parfois d'opérateur de « dépointage » ou de déréférencement.

Voici une représentation des informations présentes en mémoire après exécution des trois premières lignes de cet exemple.

```
int x =10;
int* p = &x;
*p = *p + 1;
```



On suppose que les lignes de programme sont placées dans la fonction `main`, ce qui implique que toutes les variables sont situées en mémoire dans un emplacement particulier appelé *pile*. La variable p contient l'adresse de x. On dit qu'elle *pointe* la variable x, ce qui est matérialisé sur le dessin par la flèche rouge.

Les entrées-sorties standards

Affichage à l'écran

Pour réaliser des affichages à l'écran, nous utiliserons la fonction standard `printf`. Cette fonction accepte au moins un premier paramètre appelé chaine de formatage. Ce paramètre indique le cadre et le format dans lequel les valeurs des paramètres suivants, si applicable, seront affichés.

Voici quelques formats d'affichage que nous serons amenés à utiliser :

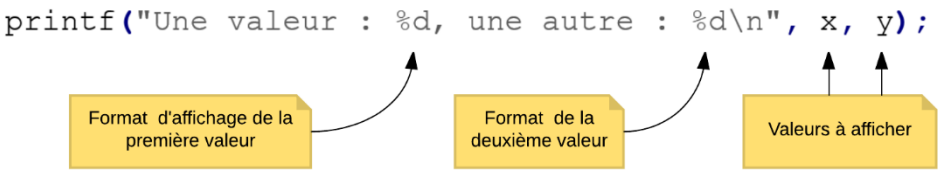
`%d`	Entier affiché en base 10
`%u`	Entier non signé affiché en base 10
`%x`	Entier affiché en hexadécimal
`%04x`	Entier en hexadécimal sur 4 chiffres, avec 0 à gauche si nécessaire
`%c`	Caractère
`%p`	Pointeur (adresse mémoire) en hexadécimal
`%s`	Chaine de caractères
`%f`	Flottant double précision
`%.02f`	Flottant double précision avec 2 chiffres après la virgule
`%e`	Flottant double précision en notation scientifique

Ceci n'est qu'un échantillon. Il est possible de faire un cours complet rien que sur ce sujet, mais n'oublions pas que nous disposons de seulement 55 heures en classe, et environ autant de travail personnel, pour aborder deux langages !

Examinons le comportement de cette fonction d'affichage sur des exemples. On considère une partie de programme dans laquelle les variables suivantes sont déclarées et initialisées :

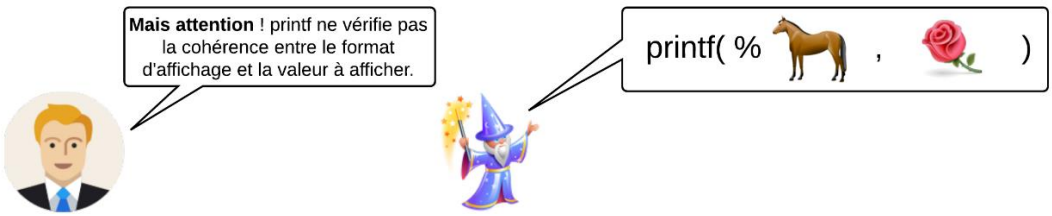
```
int x = 10;
int y = -5;
char c = 'A';
double z = 2.56;
```

Voici comment on peut afficher les valeurs de x et y, avec du texte autour, en une seule ligne de code.



Le caractère `\n` en fin de chaîne de formatage représente un retour à la ligne. Le résultat affiché est :

```
Une valeur : 10, une autre : -5
```



Voyons les résultats de quelques affichages réalisés à partir des variables déclarées plus haut.

`printf("%c %d\n", c, c);`	A 65
`printf("%u\n", y);`	4294967291
`printf("%d\n", z);`	1202590843
`printf("%.04f\n", z);`	2.5600
`printf("%p %x\n",&c,c);`	0060FEFF 41

Vous pouvez constater que certaines valeurs affichées (dans les case grisées), ne sont pas celles qui ont été transmises à la fonction `printf`. En fait, leurs codes binaires ont été interprétés par `printf` comme s'il représentait une donnée au format d'affichage demandé, d'où le résultat peu probant.

Saisie au clavier

La fonction standard `scanf` permet de saisir des données au clavier, de les convertir dans un format spécifié par une chaîne de formatage, et de les placer dans des variables dont on donne les *adresses* en paramètres.

Voici un exemple de saisie d'un entier et d'un caractère.

```
int a;
char b;
scanf("%x %c",&a, &b);
printf("Valeurs saisies : %d, %c\n",a,b);
```

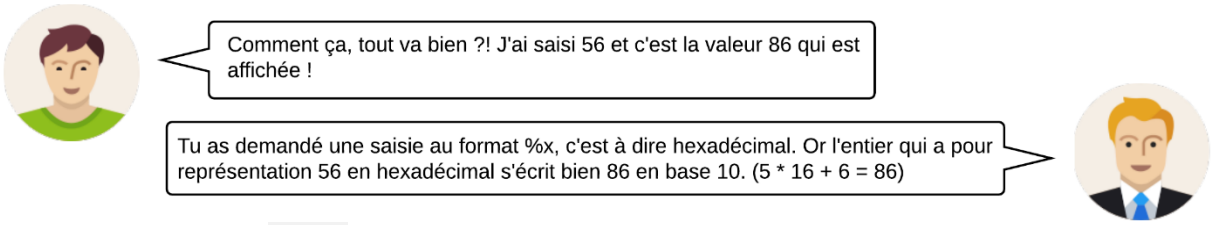
À l'exécution, rien ne s'affiche dans un premier temps. La fonction `scanf` ne fait pas d'affichage. Supposons que l'utilisateur saisisse au clavier les valeurs suivantes (il faut presser la touche <entrée> après chaque valeur) :

```
56 <entrée> t <entrée>
```

On constate l'affichage suivant :

```
Valeurs saisies : 86, t
```

Donc tout va bien.



Attention, la fonction `scanf` ne fait pas de vérification de cohérence entre la valeur saisie et le type de destination. Par exemple, si on saisit :

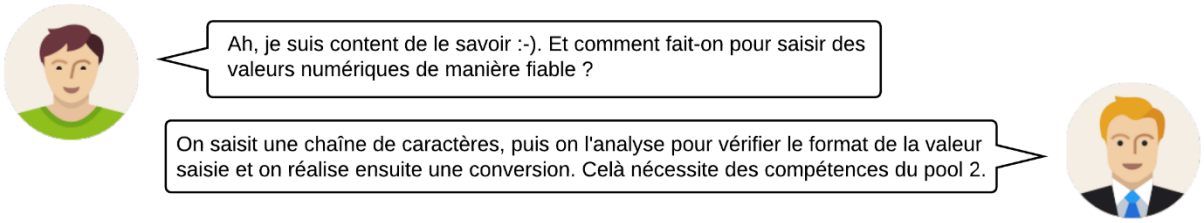
```
Donald <entrée>
```

On constate l'affichage suivant dès la première pression sur la touche entrée, sans avoir la possibilité de saisir une deuxième valeur :

```
Valeurs saisies : 13, o
```

Ces valeurs ne sont pas pertinentes parce-que le fait de ne pas saisir une valeur hexadécimale a complètement dérouté la fonction `scanf`.

DONC la fonction `scanf` n'est pas un moyen fiable de saisir des valeurs numériques. Elle peut juste être utilisée pour faire des essais, dans le cadre d'exercices de TP par exemple.



Les opérateurs

Opérateurs arithmétiques

On ne va pas en faire tout un plat. On retrouve les opérateurs arithmétiques du langage C dans de nombreux autres langages, y compris Java. Les opérations arithmétiques de bases sont réalisées par les opérateurs `+`, `-`, `*`, `/`, utilisables avec des entiers ou des flottants. Dans le cas où un des opérandes est un entier et l'autre un flottant, l'entier est d'abord converti en flottant, et le résultat est évidemment un nombre en virgule flottante.

L'opérateur `%` ne fonctionne qu'avec des opérandes entiers. Le résultat de `a % b` est le reste de la division entière de a par b.

Opérateurs de comparaison

On retrouve là encore des opérateurs présents dans d'autres langage. Plus précisément les opérateurs de comparaison de nombreux langages modernes (`==`, `!=`, `<`, `>`, `<=`, `>=`) sont hérités du langage C. Mais il y a une particularité qu'on ne retrouve plus dans les langages modernes : **Les valeurs**

booléennes vrai et faux, qui expriment notamment le résultat d'une comparaison, **sont représentées par des entiers**.

La convention est la suivante :

vrai	Tout entier différent de 0
faux	0

Donc par exemple le code suivant est compilé et exécuté sans qu'un compilateur C n'y trouve rien à redire :

```
int x = 10;
x = x + (x>5);
printf("%d\n",x);
```

Affiche 11

Avec les compilateurs les plus récents qui supportent la version C99 du langage, l'opérateur de comparaison retourne l'entier 0 pour faux ou 1 pour vrai. De plus, cette norme introduit le type `bool` et les constantes `true` et `false`, définis dans un fichier d'entête `stdbool.h`. Mais `bool` est juste l'équivalent d'un type entier, et les constantes `true` et `false` sont les valeurs entières 1 et 0 respectivement.

Opérateurs Booléens

Les opérateurs logiques du langage C sont notés `&&` (et), `||` (ou), `!` (non), comme en Java par exemple. Ils combinent deux valeurs Booléennes (donc en fait des entiers) et produisent comme résultat un entier qui représente une valeur Booléenne.

Opérateurs bitwises

Les opérateurs logiques

Ces opérateurs réalisent des fonctions logiques, comme leurs homologue Booléens, mais entre les bits de mêmes rangs des représentations binaires de leurs opérandes. Examinons par exemple le résultat de l'opération 15 & 28 en nous limitant au 8 bits les plus à droite (les autres sont tous à 0). Le principe est le même pour les autres opérateurs disponibles, à savoir `|` (ou), `~` (non), et `^` (ou exclusif).

15	0	0	0	0	1	1	1	1
28	0	0	0	1	1	1	0	0
15 & 28	0	0	0	0	1	1	0	0

Un bit est à 1 dans le résultat si et seulement si les bits de même rang dans les opérandes sont tous les deux à 1

Le résultat de 15 & 28 est donc 12.

Ces opérateurs bitwises sont très utiles pour tester la valeur d'un ou plusieurs bits ou pour changer la valeur d'un ou plusieurs bits sans modifier les autres.

Les décalages

Les opérateurs << et >> permettent de décaler vers la gauche ou la droite tous les bits de la représentation binaire d'un entier. Par exemple `128 << 2` vaut 512, parce que le décalage à gauche de tous les bits de la représentation en base 2 de 128, avec introduction de 0 à droite, a pour résultat la représentation binaire de 512. Le décalage bitwise est d'ailleurs parfois utilisé pour réaliser des

multiplications par 2, 4, 8, et autres puissances de deux. À l'inverse, `128 >> 3` a pour résultat 16, ce qui revient à diviser 128 par 8.

Maintenant, faisons une expérience.

```
printf("%d\n",-128 >> 3);
```

Affiche -16

Challenge :
Essayez de comprendre par quel prodige la division par 8 réalisée avec 3 décalages binaires a parfaitement fonctiionné avec un entier négatif.

Quelques exemples d'application

Mettre un bit à 1

Le programme suivant montre comment assigner la valeur 1 au 4ième bit en partant de la droite de la représentation binaire d'un entier.

```
unsigned int masque = 1 << 3;
unsigned data = 0xF0;
data = data | masque;
printf("%x\n", data);
```

Voici comment écrire une constante entière directement en hexadécimal.

Notez au passage la manière d'écrire une valeur directement en hexadécimal. Vous devez être en mesure de déterminer par vous-même, sans exécuter ce morceau de programme, la valeur qui sera affichée à l'écran. Si vous avez le moindre doute, essayez ! Si la valeur affichée n'est pas celle que vous aviez prévue, tentez de comprendre pourquoi, si nécessaire en réalisant d'autres expériences. Si (et seulement si) vous ne comprenez toujours pas ce qui se passe, discutez-en avec d'autres étudiants et / ou avec un enseignant. Adoptez cette attitude pour l'ensemble des enseignements que vous suivez, et vous optimiserez vos chances de réussite.

Mettre un bit à 0

Donnez les lignes de code pour assigner à 0 le 4ième bit en partant de la droite de l'entier dont la représentation hexadécimale est FFFF.

Vous pouvez trouver la solution par vous-même, et en la trouvant vous-même, votre apprentissage sera plus efficace.

Indice : utilisez les opérateurs << et ~ pour créer un masque, puis & pour modifier la donnée.

Déterminer la valeur d'un bit

On peut déterminer la valeur d'un bit situé en position i dans la représentation binaire d'un entier x en créant un masque m ayant un seul bit à 1, en position i, et en réalisant l'opération `x & m`. Le résultat est non nul si et seulement si le bit concerné a la valeur 1.

Attention !
Ceci n'est pas une formule magique à apprendre par coeur. Il est important que vous compreniez parfaitement pourquoi et comment ça fonctionne. Au moindre doute, enquêtez, renseignez-vous, faites des essais, posez des questions....

≠ C

Les instructions de contrôle

Les alternatives

Les instructions `if` et `if else` du langage C ont été reprises dans de nombreux langages, notamment Java. Leur syntaxe vous est donc certainement déjà connue.

```
if(condition)
  {
  }
```
Code exécuté si condition ≠ 0 (true)

```
if(condition)
  {
  }
else
  {
  }
```
Code exécuté si condition ≠ 0 (true)

Code exécuté si condition = 0 (false)

La condition peut être n'importe quelle expression ayant pour résultat un entier, qui sera interprété comme un Booléen.

Il existe une instruction `switch case` qui permet de réaliser un aiguillage à plusieurs branches avec des conditions simplifiées. Elle n'est pas très propre et nous ne l'utiliserons pas dans le présent enseignement, mais vous pouvez vous documenter si vous êtes intéressé.

Les instructions itératives

Elles permettent de réaliser des « boucles », i.e., de répéter un traitement tant qu'une certaine condition est vérifiée. La condition est bien sûr une expression ayant pour valeur un entier.

```
while(condition)
  {
  }
```
Code exécuté tant que condition ≠ 0 (true)

```
do
  {
  }
  while(condition);
```
Code exécuté tant que condition ≠ 0 (true)

Le corps d'une boucle `do while` est exécuté au moins une première fois avant le test de la condition, alors que celui d'une boucle `while` peut, si la condition est fausse dès le départ, ne pas être exécuté.

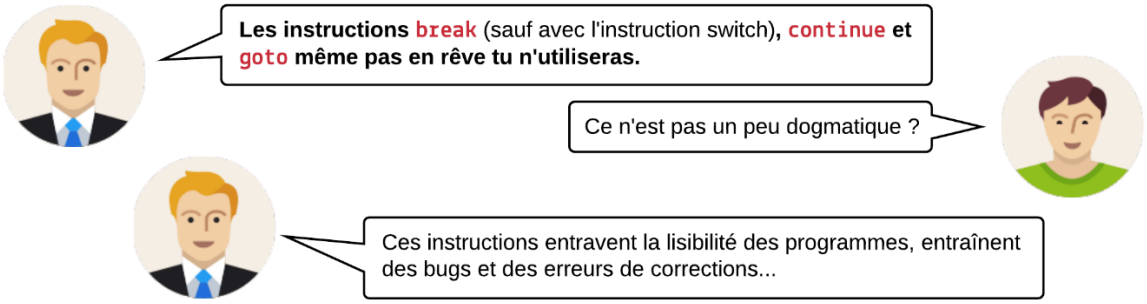
La boucle `for` est une reformulation de la boucle `while` dans laquelle des instructions d'initialisation, la condition d'itération et des instructions exécutées à la fin de chaque itération sont regroupées sur une seule ligne. Voici deux morceaux de programmes strictement équivalents réalisés avec chacune des deux syntaxes.

```c
for( i=0,j=10; i<=j; i++,j-- )
{
    printf("%d ",i*j);
}
```

```c
i=0; j=10;
while(i<=j)
{
    printf("%d ",i*j);
    i++; j--;
}
```

Une bonne pratique de programmation consiste à ne jamais modifier dans le corps d'une boucle `for` (la partie entre accolades) les variables de contrôle qui sont mises à jour dans l'emplacement identifié par le cadre bleu dans l'exemple ci-dessus (le troisième cadre si vous lisez ce document en noir et blanc).

Et les instruction break, continue, goto… ?



Les fonctions

En langage C, tous les sous programmes sont appelés fonctions, même ceux qui ne retournent pas de valeur et sont en fait des procédures. Une fonction peut donc :

- Accepter ou pas des paramètres.
- Retourner ou pas une valeur.

Pour le moment, c'est-à-dire dans le pool 1 de compétences, nous ne considérons que les paramètres et valeurs de retour de types scalaires ou pointeurs sur scalaires.

Définitions, appel, gestion de la mémoire

Voici un exemple introductif de fonction `dist` qui accepte en paramètres deux entiers et retourne la valeur absolue de leur différence, et d'une fonction `main` qui appelle la fonction `dist`.

```c
int dist(int a, int b)
{
    if(a<b)
        return b-a;
    else
        return a-b;
}
```

```c
int main()
{
    printf("%d\n", dist(-5, 15));
}
```

Affiche 20

Les deux paramètres a et b de la fonction `dist` sont des variables locales, qui n'existent que pendant l'exécution de cette fonction, qui sont initialisées par les valeurs passées en paramètres lors de l'appel de cette fonction, et qui placées en mémoire dans la pile.

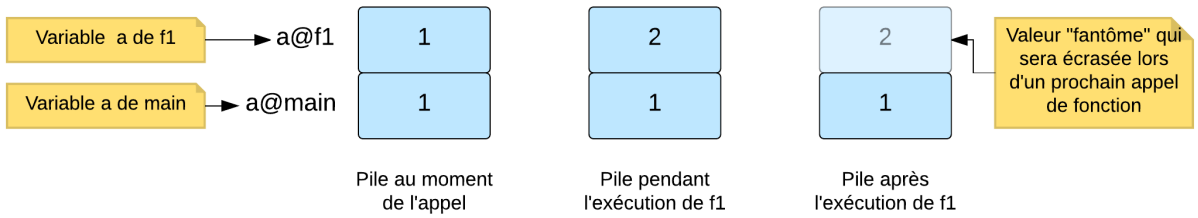
Il est important de bien retenir qu'une fonction peut modifier ses paramètres, mais sans que cela puisse avoir le moindre effet à l'extérieur de cette fonction. Voici un exemple :

```c
void f1(int a)
{
    a = a+1;
}
```

```c
int main()
{
    int a = 1;
    f1(a);
    printf("%d", a);
}
```

Affiche 1

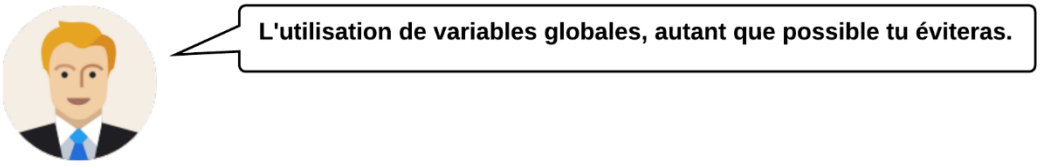
Dans ce code, il y a deux variables a, une dans `main` et une dans `f1`. Ce sont deux variables différentes. Voici ce qui se passe en mémoire pendant l'exécution de `f1` appelée par `main`.



On voit qu'au moment de l'appel, la valeur de la variable a de `main` est recopiée dans la variable paramètre a de `f1`. La fonction `f1` modifie cette copie, sans effet sur l'originale. Après l'appel, il reste dans le pile une trace de cette opération, mais qui sera écrasée lors d'un prochain appel de fonction, si applicable.

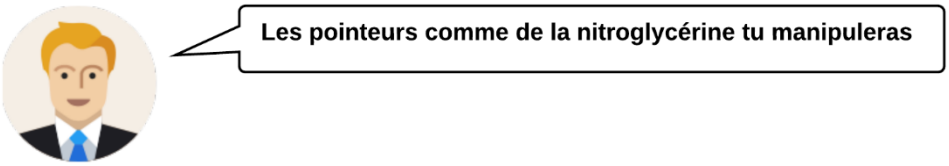
Passage par adresse et effet de bord

On parle « d'effet de bord » lorsqu'une fonction modifie une (ou plusieurs) variable(s) qui ne sont pas locales à cette fonction. Cela peut arriver quand on utilise des variables globales, qui sont définies à l'extérieur de toute fonction (y compris de la fonction main), mais…



L'utilisation de variables globales est une cause reconnue de manque de fiabilité des applications, voire comme facteur de risque très important pour des applications critiques.

Par contre on s'autorise le passage d'adresses en paramètre. (Sauf dans certaines applications très critiques, telles que la commande des gouvernes d'un avion…). Cela permet à une fonction de modifier les variables pointées par les adresses qu'elle reçoit en paramètres. En cas d'erreur de programmation, les risques de corruption de la mémoire exploitée par l'application sont considérables.



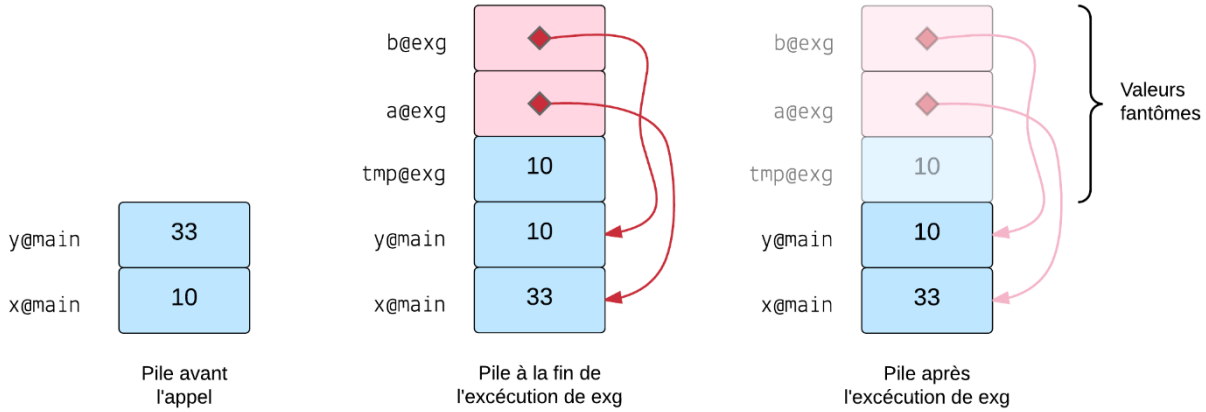
Ce conseil est valable pour les applications en production, mais bien sûr vous pouvez et même devez faire des essais, des erreurs, des expériences avec les pointeurs. Jusqu'à plus ample informé, vous ne risquez pas de détruire la machine sur laquelle vous travaillez.

Voici un exemple de fonction qui accepte en paramètres deux pointeurs a et b sur des entiers et qui échange les valeurs pointées.

```c
void exg(int* a, int* b)
{
    int tmp = *a;
    *a = *b;
    *b = tmp;
}
```

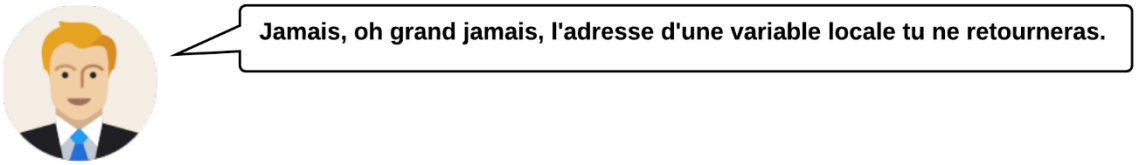
```c
int main()
{
    int x=10, y=33;
    exg(&x,&y);
    printf("%d %d\n", x, y);
}
```

Comme ce sont les adresses des variables x et y qui sont passées en paramètre à la fonction, elle peut non seulement accéder au contenu de ces variables, mais aussi les modifier. Observons ce qui se passe en mémoire.



On constate que les valeurs des variables x et y de `main` ont bien été échangées. La fonction a produit un résultat par effet de bord, alors qu'elle ne retourne pas de valeur. Mais il est possible de combiner les deux manières de fournir un résultat.

Un conseil très important au passage :



Ici encore, cela n'a rien de dogmatique. Les variables locales n'existent plus après l'exécution de la fonction dans laquelle elles ont été définie. Elles ne sont que des « fantômes » qui seront écrasés par de nouvelles variables et / ou paramètres lors des prochains appels de fonctions. Donc exploiter ces adresses mémoires non sécurisées ou leur contenu est un moyen idéal pour produire des applications au comportement imprévisible.

Définition versus déclaration

Définir une fonction consiste à donner son code complet. La déclarer consiste à donner juste sa *signature*, aussi appelée *prototype*, c'est-à-dire le types de sa valeur de retour (si applicable), le nom de la fonction et les types de ses paramètres, si applicable. Ci-dessous, un exemple de déclaration de la fonction exg de l'exemple précédent.

```c
void exg(int* a, int* b);
```

Toute fonction doit être définie une fois et peut être en outre déclarée plusieurs fois. Les principales situations où une déclaration est nécessaire sont les suivantes :

- La fonction est appelée avant sa définition, dans un même fichier source.
- La fonction est appelée dans un autre fichier source que celui où elle a été définie.
- La fonction a été compilée séparément et son code exécutable est situé dans un fichier binaire appelé « bibliothèque ». C'est le cas des fonctions standards telles que `printf`, `scanf`,…

Dans les deux dernier cas, et d'une manière générale à chaque fois qu'une fonction est susceptible d'être appelée depuis un autre fichier que celui où elle a été définie, on place sa déclaration dans un fichier d'entête avec l'extension .h. Nous y reviendrons plus tard, en abordant les principes de la programmation modulaire en langage C.

Annexes

La notation hexadécimale

Pourquoi

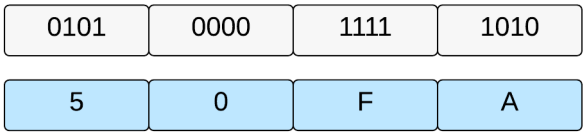
Le passage de base deux en base 16, et réciproquement, est très simple et peut se faire sans calculatrice, et même de tête avec un tout petit peu d'entraînement. La notation hexadécimale est donc une manière concise de représenter des mots binaires.

Comment ?

Chaque groupe de 4 bits correspond à un chiffre ou une lettre, avec la correspondance suivante :

0000	0	1000	8
0001	1	1001	9
0010	2	1010	A
0011	3	1011	B
0100	4	1100	C
0101	5	1101	D
0110	6	1110	E
0111	7	1111	F

Voici un exemple :



Le préprocesseur

C'est une application qui effectue un prétraitement du code source avant compilation. Elle exécute des directives qui commencent par le caractère #. Nous en avons déjà utilisé deux :

- `#include`, qui inclut dans le code le contenu d'un autre fichier, généralement un fichier d'entête ayant l'extension .h.
- `#define`, qui permet notamment de définir des constantes, mais aussi des macro-instructions, notion que nous n'aborderons pas dans le présent enseignement.

Il existe d'autres directives du préprocesseur, permettant notamment de spécifier des compilations conditionnelles, par exemple en incluant ou pas certaines parties de code selon le système

d'exploitation sur lequel le programme devra s'exécuter, ou selon que le programme est une version de développement, avec affichage d'informations de mise au point, ou bien de production. On pourrait faire un cours complet rien que sur ce sujet, mais nous n'aborderons que les notions strictement nécessaires aux objectifs pédagogiques du présent enseignement, quand elles seront nécessaires. Pour plus d'information, vous pouvez vous référer à une des sources documentaires données plus bas.

Compiler les programmes

On peut procéder en « ligne de commande », en utilisant par exemple, sous Linux, le compilateur gcc. Pour plus d'information, tapez « `man gcc` » dans une console.

On peut aussi utiliser un environnement de développement intégré (IDE) qui facilite le travail en permettant l'édition d'un code source, sa compilation et son exécution. Je vous conseille d'utiliser CodeBlock, gratuit, facilement disponible sur différents systèmes (Windows, Mac, Linux).

Se documenter

Sous forme papier, le livre « Langage C » de Claude Delanoy, aux éditions Eyrolles, est très complet et proposé à un prix abordable. Son contenu dépasse largement ce que nous aborderons dans ce cours et pourra vous être utile si vous souhaitez aller plus loin. Il existe de nombreux autres ouvrages disponibles à la vente et dans les bibliothèques universitaires.

De nombreuses sources documentaires sont également accessibles gratuitement sur le WEB, parmi lesquelles le Wikilivre « Programmation C », à l'url suivante :

https://fr.wikibooks.org/wiki/Programmation_C

Pool 2

Dans le pool précédent, nous avons acquis les compétences nous permettant de réaliser à peu près n'importe quel programme utilisant exclusivement des types scalaires. Mais beaucoup de programmes pratiques ont vocation à traiter des données représentées par des suites de valeurs. La notion de tableau correspond à la manière la plus simple et la plus « proche de la machine » d'aborder ce sujet.

Tableaux à une dimension

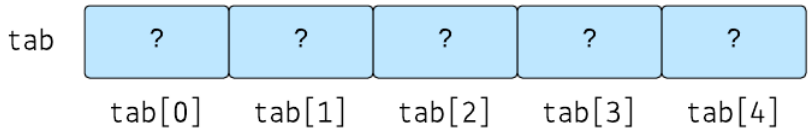
Un tableau est une structure de données représentant plusieurs valeurs de même type stockées en mémoire de manière contigüe.

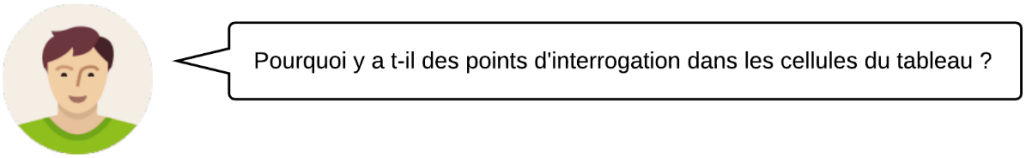
Déclarer un tableau

Voici un exemple de déclaration d'un tableau d'entiers.

```
int tab[5];
```

Ce tableau contient 5 cellules, et chacune de ces cellules contient un entier. Chaque cellule est une variable de type `int` et peut donc être utilisée de la même manière que toute autre variable de type `int`. Les cellules sont désignées dans le code par `tab[0]`, `tab[1]`, `tab[2]`, `tab[3]` et `tab[4]`.





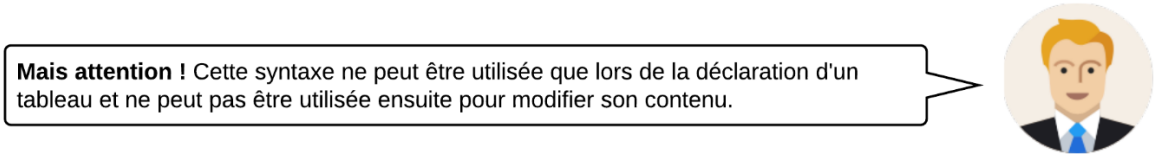
Dans ce deuxième pool de compétences, on distingue deux sortes de tableaux :

- Un tableau est dit *local* s'il est déclaré dans les limites du code définissant une fonction, c'est-à-dire au même endroit que les variables locales de cette fonction. Il est alors stocké dans la pile et s'il n'est pas explicitement initialisé, il contient des valeurs imprévisibles qui dépendent du contenu de la zone mémoire concernée.
- Un tableau est dit *global* s'il est déclaré à l'extérieur de toute définition de fonction. Il est alors placé dans la zone mémoire dédiée au stockage des variables globales et, sauf initialisation explicite, toutes ses cellules sont remplies avec une valeur par défaut, en l'occurrence la valeur 0 pour un tableau d'entier ou d'un type similaire (`char`, `short`, `long`, `unsigned`…).

On peut initialiser explicitement un tableau de la manière suivante :

```
int tab[5] = {10, 20, 30};
```

Dans cet exemple, les trois premières cellules du tableau sont initialisées avec les valeurs données entre accolades, *et les valeurs suivantes avec des 0*.



La taille d'un tableau local ou global est fixée une fois pour toutes lors de sa déclaration. Il faudra attendre le pool 3 pour pouvoir créer des tableaux de tailles variables.

Pointeurs et tableaux

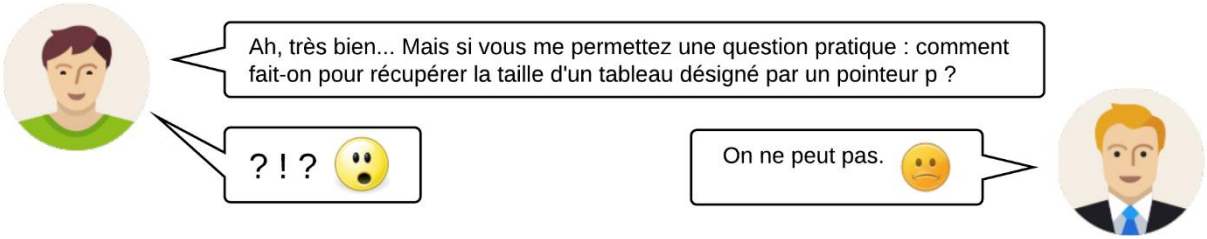
En langage C, le nom d'un tableau désigne aussi l'adresse mémoire de sa première cellule, et est par conséquent considéré comme un pointeur. Réciproquement, tout pointeur peut être utilisé comme un identificateur d'un tableau, même s'il ne pointe pas véritablement un tableau, ce qui constitue une source inépuisable de bugs et de catastrophes. Examinons un exemple.

```c
int tab[5] = {10, 20, 30};
int* p = tab;
p[1] = 55;
tab[2] = 88;
printf("%d %d %d %d %d", tab[0], tab[1], tab[2], sizeof(tab), sizeof(p));
```

L'affichage obtenu lors de l'exécution de ces lignes de codes est le suivant :

```
10 55 88 20 4
```

On constate donc qu'il a été possible de modifier la cellule `tab[1]` en utilisant la variable `p` qui pointe le début du tableau `tab`. Le langage C ne fait de différence entre le nom du tableau et son adresse. Par contre, les deux dernières valeurs affichées, `sizeof(tab)` et `sizeof(p)` sont différentes ! C'est parce-que `sizeof` est une méta-instruction qui est évaluée lors de la compilation du programme et dans le cas où son argument est une variable, elle donne la quantité de mémoire occupée par le contenu de cette variable. Donc `sizeof(tab)` donne la quantité de mémoire occupée par le tableau `tab`, à savoir 5 fois la taille d'un entier, soit 5x4 = 20 octets. Alors que `sizeof(p)` donne la taille occupée en mémoire par (le contenu de) la variable `p`, soit la taille d'une adresse mémoire. L'exemple ayant été compilé avec un compilateur 32 bits, les adresses sont codées sur 4 octets.



La taille d'un tableau n'est pas stockée en mémoire, donc c'est au programmeur de gérer cette information et de l'exploiter à chaque fois qu'il souhaite, par exemple, réaliser une opération avec tous les éléments d'un tableau, comme par exemple les afficher !

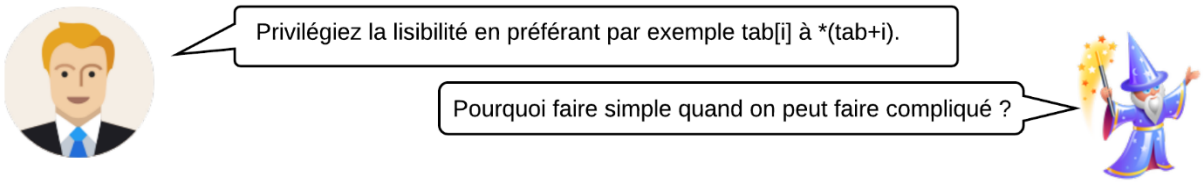
Arithmétique des pointeurs

On peut réaliser certaines opérations sur les pointeurs, à savoir :

- *Ajouter* ou soustraire un entier n à un pointeur p. L'adresse `p + n` représente la valeur `p + n * sizeof(*p)`, où `*p` est la donnée pointée par p. Donc ajouter un entier n à un pointeur p revient à lui ajouter un déplacement qui représente n fois la taille en mémoire de la valeur pointée par p. **Attention**, le programmeur ne doit pas écrire `p + n * sizeof(*p)` car le compilateur C fait déjà le calcul en exploitant le type du pointeur p. Il y a toujours quelques étudiants qui perdent bêtement un demi-point à l'examen en faisant cette erreur. Bien sûr, soustraire un entier à un pointeur revient à lui ajouter la valeur opposée.
- *Soustraire* deux pointeurs de même type. Si p et q sont des pointeurs sur des entiers, par exemple, `p - q` représente `(p - q) / sizeof(int)`, c'est-à-dire le nombre d'entiers qu'il est possible de placer en mémoire entre les deux adresses. Si p représente une adresse située avant q (donc plus petite), on obtient une valeur négative, mais dont la valeur absolue représente bien le nombre d'entiers qui peuvent être placés en mémoire entre les deux adresses concernées. Cette règle de calcul se généralise à tous type de pointeurs.
- *Comparer* deux pointeurs. Si p et q sont des pointeurs de même type, on peut écrire des expression telles que `p == q`, `p != q`, `p < q`,… qui retournent un booléen (représenté par un entier) résultat de la comparaison des adresses concernées.

Ces opérations sur les pointeurs peuvent être utiles lorsque les pointeurs concernés désignent des tableaux. Voici quelques notations strictement équivalentes basées sur le tableau `tab` de l'exemple précédent.

`tab[i]`	est équivalent à	`*(tab+i)`
`&(tab[i])`	est équivalent à	`tab+i`
`*tab`	est équivalent à	`tab[0]`
`tab`	est équivalent à	`&(tab[0])`



Fonctions et tableaux

Pour mémoire, lorsqu'on passe en paramètre une valeur scalaire à une fonction, c'est une *copie* de cette valeur qui est transmise à la fonction lors de l'appel. Cette copie est placée dans une variable locale faisant office de paramètre.

Or il n'est pas possible de passer en paramètre à une fonction la copie d'un tableau. On peut seulement transmettre à une fonction l'adresse d'un tableau, ou les adresses de plusieurs tableaux. De même, une fonction ne peut pas retourner un tableau, tout au plus l'adresse d'un tableau qui ne doit pas être local à cette fonction.

Concrètement, pour qu'une fonction puisse accéder aux données stockées dans un tableau, il faut lui passer en paramètre l'adresse de la première cellule de ce tableau, ainsi que la taille de ce tableau, sous la forme d'un paramètre de type entier. La fonction peut alors non seulement lires les valeurs situées dans le tableau, mais aussi les modifier, sauf si le modificateur `const` est utilisé, et encore, il est toujours possible de « tricher ».

Commençons par examiner un exemple simple, à savoir une fonction qui affiche un tableau de valeurs de type `double`. Attention, il y a deux manières de déclarer le paramètre désignant le tableau : l'une montre explicitement qu'il s'agit d'un pointeur, l'autre que c'est l'adresse d'un tableau qui devra être

passée en paramètre, mais le compilateur ne fait aucune différence entre les deux syntaxes. La taille du tableau est transmise à la fonction via le paramètre n.

```
void print_tab_double(const double t[], int n)
{
    for(int i=0; i<n; i++)
    {
        printf("%.02f ",t[i]);
    }
    printf("\n");
}
```

Peut aussi s'écrire `const double* t`

Voici un exemple de fonction permettant de tester la fonction d'affichage ci-dessus.

```
void exp_tab2()
{
    double tab[] = {1.0, 2.2, 3.3};
    print_tab_double(tab,3);
}
```

Les opérations arithmétiques sur les pointeurs vues précédemment et l'équivalence entre pointeurs et tableaux autorisent des variations telles que celle-ci-dessous, qui a pour effet l'affichage des deux dernières valeurs du tableau.

```
    print_tab_double(tab+1,2);
```

Mais attention, il est aussi possible d'écrire du code qui va lire ou écrire des valeurs en dehors des limites du tableau, avec des conséquences aussi fâcheuses qu'imprévisibles : le langage C privilégie l'efficacité plutôt que la sécurité.

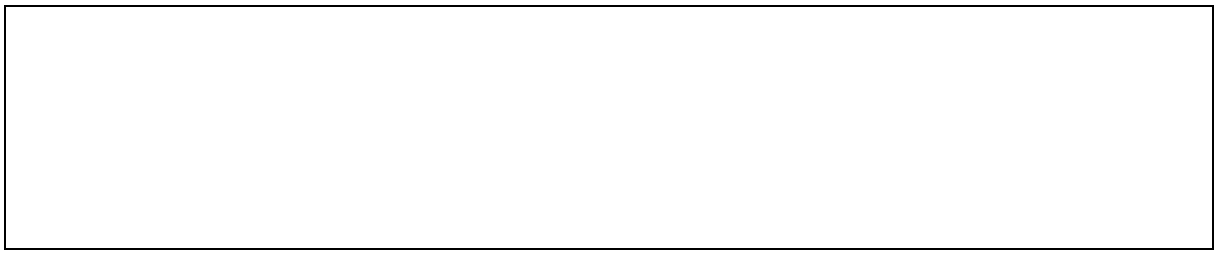
Voici un exemple de fonction qui recopie les n premier éléments d'un tableau de double dans un autre, en supposant que chacun des deux tableaux, source et destination, aient une taille au moins égale à n. La vérification de ce prérequis ne *peut pas* être réalisé par la fonction elle-même et doit donc être faite, si applicable, *avant* son appel.

```
void copy_tab_double(const double* src, double* dest, int n)
{
    for(int i=0; i<n; i++)
    {
        dest[i] = src[i];
    }
}
```

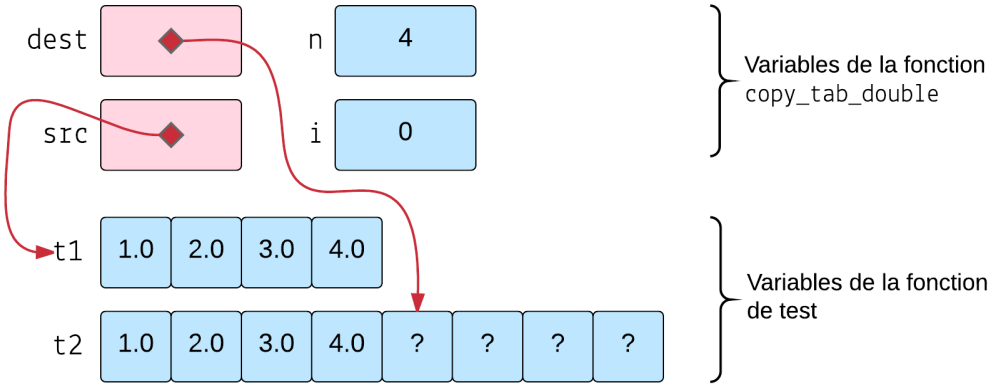
Maintenant, à vous de jouer !

Écrivez les lignes de codes permettant :

- de créer un tableau `t1` de 4 cellules contenant des nombre de type `double`, initialisé avec les valeurs 1.0, 2.0, 3.0, 4.0,
- ainsi qu'un tableau `t2` non initialisé de 8 cellules de type `double`,
- puis d'appeler deux fois la fonction `copy_tab_double` pour recopier le contenu de `t1` dans les 4 premières cellules de `t2` et dans les 4 cellules suivantes.



Voici l'état de la mémoire juste au début de l'exécution du deuxième appel de la fonction `copy_tab_double`. Toutes les variables et tableaux représentés sont dans la pile.

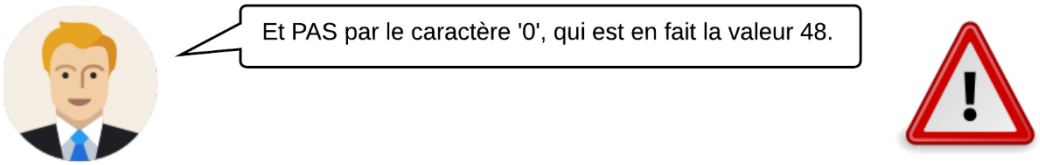


Chaînes de caractères

Représentation des chaînes

Les chaînes de caractères sont représentées par des tableaux de caractères (valeurs de type `char` codées sur 8 bits). Il n'y a donc pas de type spécifique tel que le « `string` » du C++ par exemple. Il y a juste une convention :

La fin d'une chaîne est marquée par la valeur 0, parfois notée `'\0'`.



La longueur d'une chaîne n'est donc *pas* nécessairement celle du tableau qui la contient, mais le nombre de cellules de ce tableau situées avant la valeur 0.

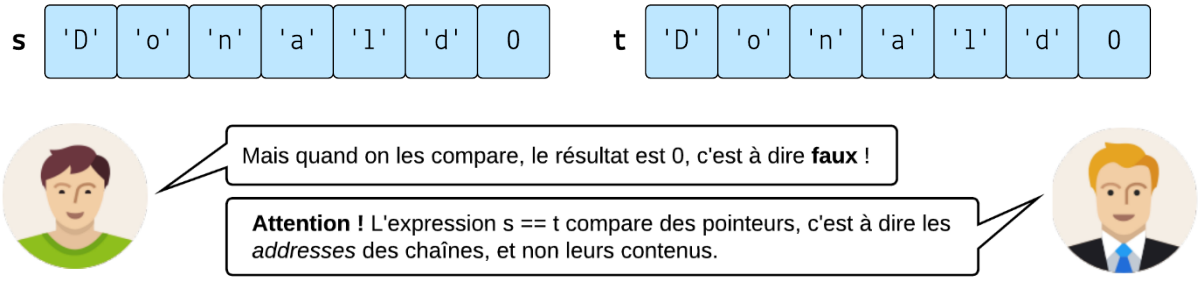
Il existe toutefois certaines facilités pour manipuler les chaînes, en particulier une notation entre doubles quottes (guillemets) reconnue et traduite par le compilateur en une suite de caractères terminée par la valeur 0, i.e., le *marqueur de fin de chaîne*. Voici deux manières légales de déclarer et d'initialiser un tableau représentant une chaîne de caractères.

```c
char s[7] = "Donald";
char t[] = {'D', 'o', 'n', 'a', 'l', 'd', 0};
printf("%s --- %s --- %d\n",s,t,s==t);
```

L'exécution de ces lignes de codes produit l'affichage suivant :

```
Donald --- Donald --- 0
```

Le tableau t est dimensionné automatiquement par le compilateur à une taille de 7 cellules, c'est-à-dire juste le nombre nécessaire pour placer les 6 caractères de la chaîne et le marqueur de fin. La taille du tableau s est donnée de manière explicite. Il est tout à fait possible de donner une taille plus grande dans la perspective de stocker plus tard une chaîne plus longue. Les deux chaînes sont représentées exactement de la même manière en mémoire.



N'oubliez pas que les chaînes sont désignées par des pointeurs (ou des noms de tableaux qui représentent donc aussi des adresses), et que si vous utilisez des opérateurs tels que `==` (comparaison), `=` (assignation), ou `+`, ce sont des opérations sur les adresses des chaînes qui seront réalisées, et non sur leurs contenus. Cette confusion coute chaque année des points à l'examen à de nombreux étudiants.
Pour réaliser des opérations sur les chaînes, il faut agir directement sur le contenu des tableaux qui les représentent, et il existe pour cela certaines fonctions standards très pratiques.

Fonctions et chaînes

Fonctions standards

Le langage C dispose de fonctions spécialisées dans la gestion des chaînes de caractères qui sont déclarées dans le fichier d'entête `string.h`. En voici quelques-unes parmi les plus utilisées :

```c
char* strcpy(char* dest, const char* src);
```

Copie le contenu de la chaîne désignée par `src` (y compris le marqueur de fin) à l'adresse de destination `dest`. Il faut que `dest` pointe un emplacement mémoire valide pouvant contenir la chaîne source.

```c
char* strncpy(char* dest, const char* src, size_t n);
```

Copie le contenu de la chaîne désignée par `src` (y compris le marqueur de fin) à l'adresse de destination `dest`, mais en copiant au plus `n` caractères. Il faut que `dest` pointe un emplacement mémoire valide pouvant contenir au moins `n` caratères. Si la chaîne copiée comporte plus de `n` caractères, aucun marqueur de fin n'est ajouté.

```c
char* strcat(char* dest, const char* src);
```

Copie le contenu de la chaîne désignée par `src` (y compris le marqueur de fin) à la fin de la chaîne désignée par `dest` de manière à ce que la chaîne de destination soit le résultat de la concaténation

de son ancien contenu et de la chaîne source. Il faut que `dest` pointe un emplacement mémoire valide pouvant contenir le résultat de cette concaténation.

```
char* strncat(char* dest, const char* src, size_t n);
```
Version sécurisée de `strcat` qui limite à n le nombre de caractère ajouté à la chaîne de destination.

```
size_t strlen(const char* src);
```
Retourne la longueur de la chaine désignée par `src`, c'est-à-dire son nombre de caractères sans compter le marqueur de fin.

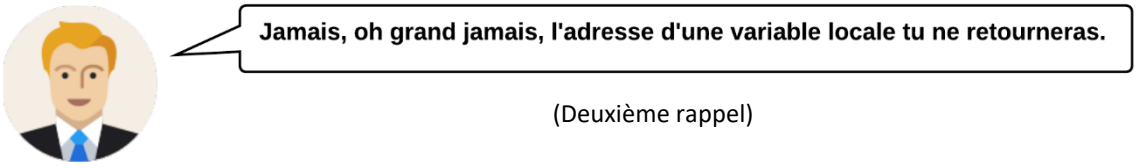
```
int strcmp(const char* str1, const char* str2)
```
Compare les chaînes désignées par str1 et str2 d'après l'ordre lexicographique basé sur les codes ascii des caractères. Retourne 0 si les deux chaines sont identiques, une valeur négative si la première est inférieure à la seconde, sinon une valeur positive.

Le type `size_t` qui apparait dans certaines déclarations est un type entier non signé définit dans le fichier d'entête standard `stddef.h`. Certaines des fonctions présentées ci-dessus retournent un char*. La valeur retournée est un pointeur sur la chaîne résultat ou destination de l'opération réalisée, qui l'un des paramètres. En pratique, cette valeur de retour est donc inutile et elle n'est généralement pas exploitée.

Réaliser ses propres fonctions de traitement de chaînes

Il existe bien d'autres fonctions de traitement de chaînes de caractères disponibles, et il est bien sûr possible de coder des fonctions acceptant en paramètres des chaînes de caractères pour répondre à tout besoin spécifique. Les chaînes étant représentées par des tableaux de caractères, tout ce qui a été dit à propos des fonctions avec paramètres désignant des tableaux s'applique au passage de chaînes (ou plus précisément d'adresses de chaînes) en paramètres.

Voici à titre d'exemple une fonction qui produit une chaîne constituée de n occurrences d'un même caractère c. Au stade où nous en sommes, nous ne pouvons réaliser une fonction retournant une chaîne de caractères qu'elle aurait complètement créée sous forme d'un tableau n'existant pas lors de l'appel à cette fonction car une fonction ne peut pas retourner un tableau, tout au plus l'adresse d'un tableau mais…



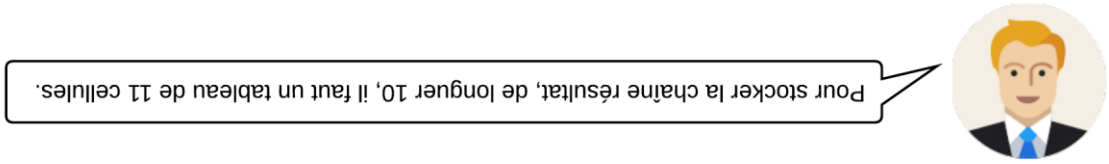
(Deuxième rappel)

Donc nous devons passer en paramètre à notre fonction l'adresse d'un tableau existant au moment de l'appel, et de taille suffisante pour contenir la chaîne crée, marqueur de fin inclus. Voici la définition d'une telle fonction.

```
void fillStr(char* dest, char c, unsigned n)
{
    for(unsigned i=0; i<n; i++)
    {
        dest[i] = c;
    }
    dest[n] = 0;
}
```

Voici un exemple d'appel de cette fonction *qui comporte une erreur*. Essayez de trouver cette erreur avant de lire la suite.

```
char s[10];
fillStr(s,'A',10);
```



Une telle erreur est n'est pas détectée à la compilation et peut très bien n'avoir aucun effet visible à l'exécution, tout comme elle peut provoquer un plantage du programme. Tout dépend de la manière dont est utilisée la zone mémoire (ici un seul octet) située juste après le tableau de destination, qui reçoit inopinément la valeur 0. Un des effets possibles est le changement de valeur d'une autre variable.

Pool3

Dans ce pool, nous abordons deux notions importantes : les structures et la gestion dynamique de la mémoire.

Les structures

Notions de base

Une structure permet de regrouper plusieurs données qui ne sont pas nécessairement du même type. Chacun de ses membres, qui peuvent être des scalaires, des tableaux, ou des structures, est appelé *champ* et est identifié par un nom.

Voici un exemple de définition d'une structure décrivant un point à l'aide de deux coordonnées.

```
typedef struct
{
    double x;
    double y;
} point;
```

Syntaxe la plus simple

Cette déclaration définit un type `point` représentant une structure ayant deux champs nommés x et y. Ce type pourra être utilisé pour déclarer des variables, des paramètres de fonctions, des valeurs de retour de fonctions…

Il est aussi possible d'utiliser la syntaxe suivante :

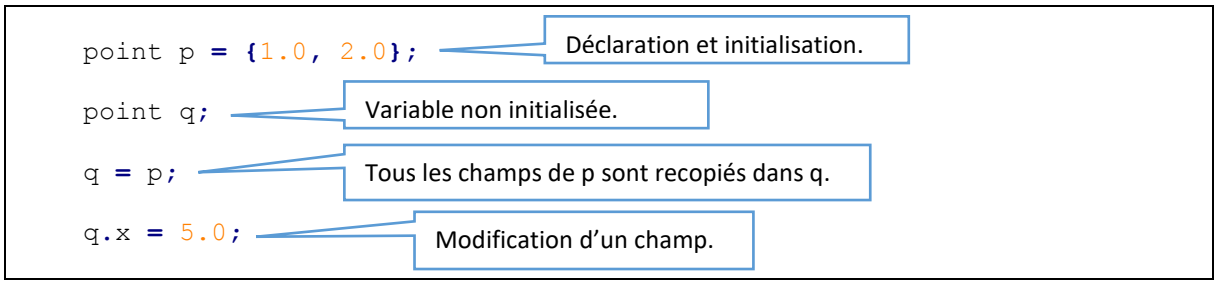
```
struct point
{
    double x;
    double y;
};
```

Syntaxe à éviter, donnée pour information

Mais attention, dans ce cas le type définit se nomme `struct point` et non `point`. Cette syntaxe plus lourde est à éviter, sauf dans le cas de structures auto référentes, c'est-à-dire dont l'un des champs est de type pointeur sur la structure elle-même, mais nous n'utiliserons pas ce type de structure cette année, faute de temps. Nous utiliserons par contre dans le pool 5 des classes auto référentes en C++. Il est aussi possible d'utiliser simultanément les deux syntaxes, mais c'est sans intérêt pour nos besoins actuels.

Les lignes de code ci-dessous montrent comment déclarer une variable de type structure, l'initialiser et modifier les valeurs de ses champs.

Attention, l'initialisation des valeurs n'est possible qu'une seule fois lors de la déclaration d'une variable. Par contre il est possible d'assigner une variable de type structure avec le contenu d'une autre variable de même type, ou par exemple avec la valeur de retour d'une fonction retournant une structure de même type, mais nous y reviendrons plus tard.



Le fait de pouvoir assigner directement une variable de type structure est un gros avantage des structures par rapport aux tableaux. Rappelez-vous que pour modifier les valeurs d'un tableau, il faut procéder case par case (sauf à utiliser des fonctions de transfert mémoire qui sont hors programme).

Fonctions et structures

Il y a deux manières de passer une structure en paramètre à une fonction :

1. **Par valeur**. Dans ce cas, le contenu de la structure passée en paramètre lors de l'appel de la fonction est *recopié* dans la variable locale faisant office de paramètre.
2. **Par adresse** (pointeur). Dans ce cas, seul l'adresse d'une structure, et non son contenu, est transmise en paramètre lors de l'appel de la fonction. Si le pointeur n'est pas déclaré avec le modificateur `const`, la fonction peut modifier la structure pointée.

D'autre part, une fonction peut retourner une valeur de type structure. Elle peut aussi retourner l'adresse d'une structure, mais attention, en aucun cas l'adresse d'une variable locale de la fonction concernée.

Voici un premier exemple de fonction qui accepte deux points a et b en paramètres et retourne le point situé au milieu du segment [a,b].

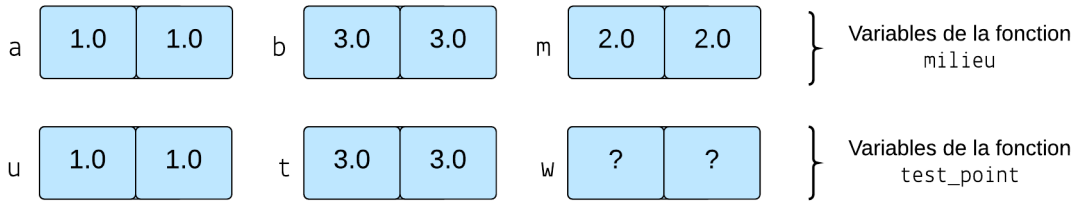
```
point milieu(point a, point b)
{
    point m;
    m.x = (a.x + b.x) / 2.0;
    m.y = (a.y + b.y) / 2.0;
    return m;
}
```

Dans cet exemple, la fonction reçoit en paramètres deux copies d'*instances* de point et retourne une *instance* de point. Dans la suite, on appellera instance la donnée contenue dans une variable de type structure ou pouvant être placée dans une telle variable (par exemple après avoir été retournée par une fonction).

Voici un exemple d'appel de cette fonction.

```
void test_point()
{
    point u = {1.0,1.0}; point t = {3.0,3.0};
    point w = milieu(u,t);
    printf("(%.02f , %.02f)\n",w.x, w.y);
}
```

Il est intéressant et instructif de regarder ce qui se passe dans la mémoire, précisément dans la pile, juste avant la fin de l'exécution de la fonction `milieu` appelée par la fonction `test_point`.

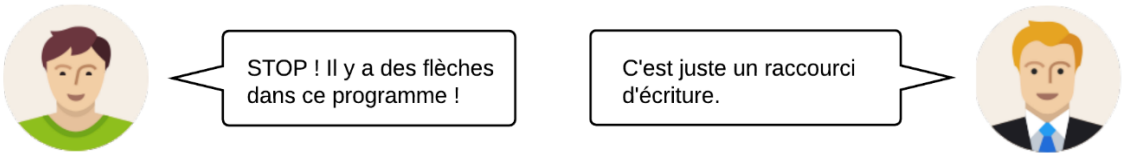


On voit bien sur le schéma que les valeurs des structures u et t de la fonction `test_point` ont été *recopiées* dans les paramètres a et b de la fonction `milieu`, qui a placé le résultat de son calcul dans sa variable locale m. Cette valeur sera elle-même recopiée dans la variable w de la fonction `test_point` au moment de l'exécution de l'instruction `return` de la fonction `milieu`.

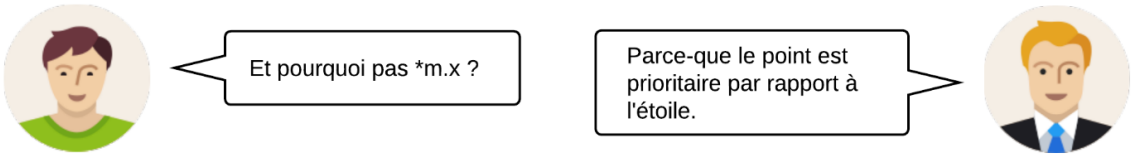
Ces recopies des valeurs des champs des structures évitent tout risque d'effet de bord, mais impactent l'efficacité du programme en termes de temps d'exécution et de consommation de mémoire. Ce n'est pas grave si le programme est destiné à être exécuté sur une machine puissante de type ordinateur de bureau, mais peut être gênant dans le cas d'une application embarquée exécutée sur un système à ressources limitées.

Voici maintenant une version optimisée avec passage des paramètres par adresses et « livraison » du résultat par effet de bord.

```
void milieu(const point* a, const point* b, point* m)
{
    m->x = (a->x + b->x) / 2.0;
    m->y = (a->y + b->y) / 2.0;
}
```



Si m désigne l'adresse d'une structure et x un des champs de cette structure, alors la notation `m->x` est équivalente à `(*m).x`, c'est-à-dire le champ x de la structure *pointée* par m. Cette notation est non seulement plus courte, mais aussi plus lisible. Merci de l'utiliser.

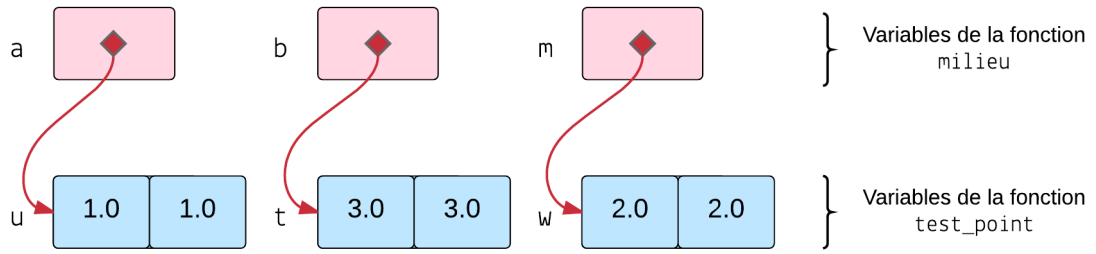


Autrement dit, `*m.x` est équivalent à `*(m.x)` et pas à `(*m).x`.

Voici maintenant la nouvelle version de la fonction `test_point` qui appelle la fonction `milieu` en lui passant en paramètres les adresses des deux points de références et l'adresse de la structure dans laquelle le résultat du calcul sera placé.

```c
void test_point()
{
    point u = {1.0,1.0}; point t = {3.0,3.0};
    point w;
    milieu(&u, &t, &w);
    printf("(%.02f , %.02f)\n",w.x, w.y);
}
```

Examinons à présent ce qui se passe dans la mémoire juste à la fin de l'exécution de la fonction `milieu` appelée par la fonction `test_point`.



La différence en termes de quantité de mémoire utilisée et de données copiées n'est pas énorme parce-que notre structure `point` ne comporte que deux champs, mais dans le cas d'utilisation de structures de taille importante, le gain pourrait être considérable.

Aller plus loin

Les exemples donnés ci-dessus ne font qu'effleurer les possibilités offertes par les structures. On peut faire des structures dont les champs sont des tableaux, des structures, des pointeurs, des tableaux de structures… et il est bien sur possible de réaliser des fonctions acceptant à la fois des paramètres de types structures et d'autres de types pointeurs sur des structures.

Les structures permettent par exemple « d'encapsuler » des tableaux de manière à permettre leur passage en paramètre par valeur et / ou le retour d'un tableau par une fonction. Vous aurez l'occasion d'explorer certaines de ces possibilités en lors des travaux pratiques et travaux dirigés. N'hésitez pas à compléter ces explorations en réalisant vos propres expérimentations.

La gestion dynamique de la mémoire

Jusqu'à maintenant, toutes les variables que nous avons utilisées, y compris de types structures et tableaux, étaient stockées dans la pile. Nous avons aussi évoqué les variables globales, stockées dans une zone mémoire spécifique.

Mais il existe une troisième zone dans laquelle il est possible de stocker des données de tout type pour une durée qui n'est pas limitée à l'exécution d'une fonction, mais qui n'est pas non plus nécessairement celle de l'exécution du programme. On parle de *variables dynamiques*.

Toutes les variables dynamiques sont stockées dans une zone mémoire appelée le *tas*. Elles sont créées et détruites à l'aide d'une panoplie de 4 fonctions dont voici les descriptions.

Les 4 fonctions de gestion de la mémoire

```c
void* malloc(size_t size)
```

Réserve dans le tas un bloc mémoire de la taille indiquée. Retourne l'adresse du bloc réservé ou la valeur NULL dans le cas où la réservation a échoué.

Il y a deux nouveautés qui apparaissent ici :

1. Cette fonction retourne un pointeur de type `void*`. C'est nouveau car jusqu'à maintenant nous n'avons vu que des pointeurs typés. Un pointeur de type `void*` pointe une adresse mémoire dont l'usage n'est pas défini. L'arithmétique des pointeurs, l'opérateur de déréférencement *, la possibilité d'utiliser un pointeur comme tableau ne fonctionnent pas avec les `void*`.
2. Cette fonction peut retourner la valeur `NULL`. Cette valeur n'est pas une adresse valide (en fait c'est l'adresse 0, à laquelle il ne peut jamais y avoir de données exploitables par un programme). C'est une valeur spéciale qui peut être utilisée par exemple pour indiquer une erreur.

```
void *calloc(size_t nitems, size_t size)
```

Réserve dans le tas un bloc mémoire d'une taille égale à `nitems * size`. Retourne l'adresse du bloc réservé ou la valeur NULL dans le cas où la réservation a échoué. Le bloc réservé est initialisé avec des octets ayant la valeur 0.

Cette fonction, fait un travail assez similaire à celui réalisé par `malloc`, mais s'exécute un peu plus lentement car elle remplit avec des 0 le bloc réservé alors que `malloc` ne fait aucune initialisation. La fonction `calloc` est plutôt utilisée pour la réservation de tableaux dynamiques, d'où les noms de ses deux paramètres représentant la taille (en nombre de cellules) du tableau à réserver et la taille (en octets) des cellules de ce tableau.

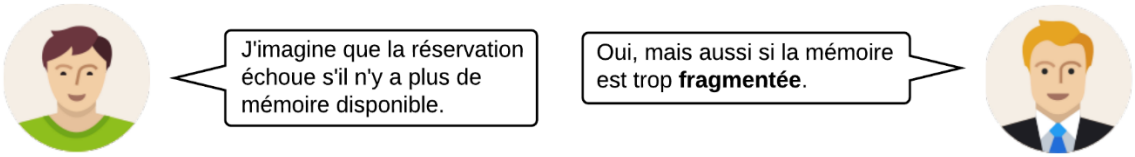
```
void *realloc(void *ptr, size_t size)
```

Si le paramètre `ptr` a la valeur `NULL`, cette fonction est équivalente à `malloc(size)`. Sinon elle redimensionne le bloc situé à l'adresse `ptr` (qui doit déjà avoir été réservé préalablement). Ce redimensionnement peut occasionner un déplacement du bloc dans la mémoire. Dans tous les cas, la valeur retournée est celle du nouveau bloc, qu'il ait ou non la même adresse que l'ancien. Si le bloc est déplacé, les valeurs contenues dans l'ancien bloc sont recopiées dans le nouveau. Les valeurs suivantes (si applicable) ne sont pas initialisées. Comme `malloc`, `realloc` retourne `NULL` si la réservation ou le redimensionnement échoue.

Typiquement, `realloc` est utiliser pour modifier la taille d'un tableau dynamique, donc soit pour l'agrandir (en conservant les valeurs des cellules qui existaient avant le redimensionnement), soit pour le rétrécir, auquel cas les valeurs des cellules situées au-delà de l'ancienne limites sont bien sûr perdues.

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

Libère un bloc mémoire ayant été préalablement réservé. Le paramètre `ptr` est l'adresse du bloc à libérer. Si cette adresse ne pointe pas un bloc préalablement réservé, le comportement de la fonction est imprévisible et peut conduire à un crash de l'application.



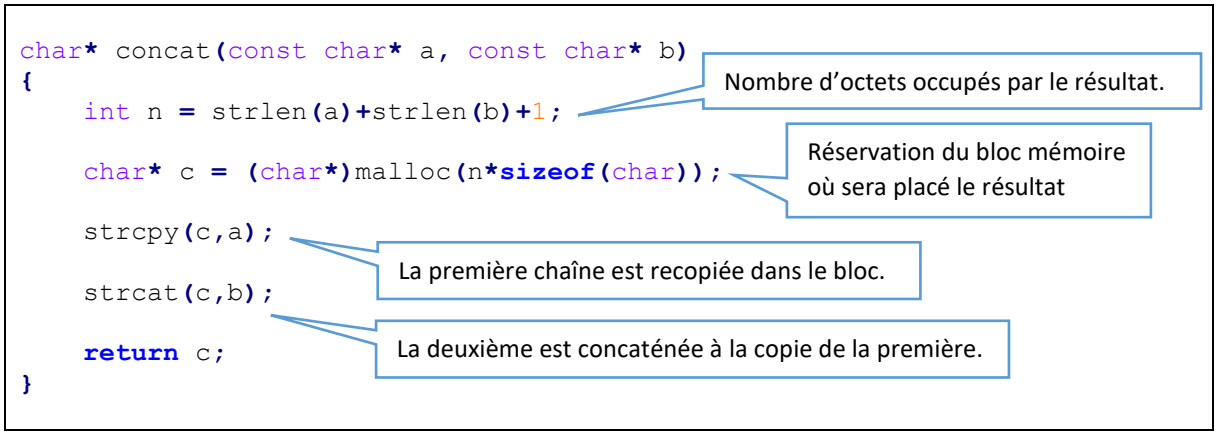
En effet, si une application réserve et libère des blocs de tailles différentes, il peut très bien arriver que de nombreux blocs réservés soient dispersés dans la mémoire et qu'il n'y ait plus de zone suffisamment

grande pour répondre à une nouvelle demande, même si la quantité de mémoire libre totale est très supérieure à la taille du bloc demandé. Comme il n'y a aucun système de défragmentation automatique de la mémoire utilisée par le tas d'une application réalisée en langage C, le seul moyen d'éviter le blocage est de libérer certains blocs réservés, mais il n'existe aucun moyen rapide d'identifier ces blocs.

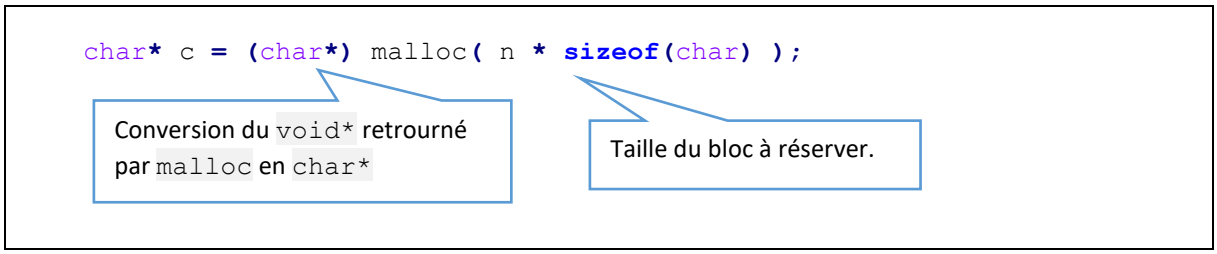
Dans des applications extrêmement critiques (spatial, aéronautique, automobile, médical, monétique…), on évitera les variables dynamiques ou on fera en sorte que l'ordre et les tailles des blocs réservés et libérés rendent impossible une telle situation.

Un exemple d'utilisation

La fonction définie ci-dessous crée une nouvelle chaîne de caractère contenant le résultat de la concaténation de deux chaînes existantes désignées par les paramètres a et b. La nouvelle chaîne crée est stockée dans le tas.



Observons attentivement la manière dont la fonction `malloc` est utilisée.



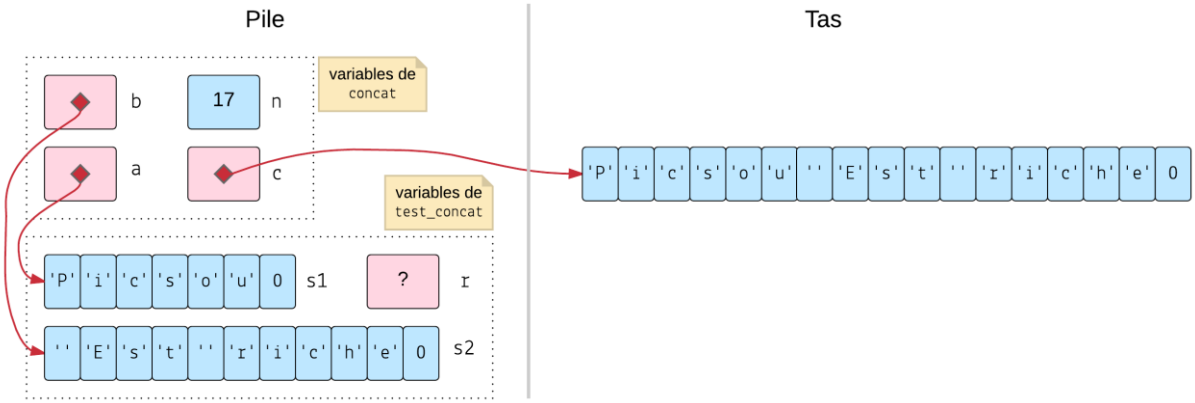
On retrouvera toujours ce même schéma : la taille du bloc à réservé est calculée à l'aide de la commande `sizeof` (il se trouve d'ailleurs que `sizeof(char)` vaut 1, mais c'est un cas particulier), et la valeur de retour est « castée » avec le type approprié car le compilateur C émet une erreur ou un avertissement en cas de tentative de mettre une adresse de type `void*` dans une variable de type pointeur typée.

Examinons maintenant un exemple d'utilisation de cette fonction `concat`.

```c
void test_concat()
{
    char s1[] = "Picsou";
    char s2[] = " Est riche";
    char* r = concat(s1,s2);
    printf("%s\n",r);
    free(r);
}
```

Les deux premières lignes de la fonction créent chacune une chaîne dans la pile. Comme les tailles des tableaux représentant ces chaînes ne sont pas précisées, le compilateur réserve juste ce qui est nécessaire, donc par exemple 7 caractères pour `s1` (incluant le marqueur de fin). Ensuite la concaténation est réalisée par un appel à `concat` qui retourne l'adresse du résultat, situé dans le tas. La chaîne résultat est affichée pour vérification, puis détruite par un appel à `free`. D'une manière générale, dans un programme bien conçu, chaque bloc réservé par `malloc` (ou `calloc` ou `realloc`) doit être libéré par `free`, mais pas nécessairement dans la même fonction. C'est justement l'intérêt des variables dynamiques : leur durée de vie est indépendante de celle de la fonction qui les a créées.

Examinons ce qui se passe dans la mémoire juste avant la fin de l'exécution de la fonction `concat` appelée par `test_concat`.



Immédiatement après, c'est évidemment la variable `r` de `test_concat` qui pointera la chaîne résultat dans le tas.

Cet exemple présentait une application avec des chaînes, mais on peut créer des variables dynamiques contenant n'importe quel autre type de tableau, de structure, et même des valeurs de types scalaires.

Fuites et accidents mémoires

Les fuites et accidents mémoire sont les deux fléaux liés à l'utilisation des variables dynamiques.

On parle de *fuite mémoire* lorsque certains blocs réservés ne sont plus pointés par aucune variable (incluant des variables dynamiques) et n'ont pas été détruit par un appel à `free`.

Dans le cadre de ce cours, le terme accident mémoire désigne essentiellement deux types de situations fâcheuses :

1. le cas où un bloc de mémoire réservé a été libéré plusieurs fois,
2. toute tentative d'écriture en mémoire dans une zone non réservée et/ou non autorisée.

Les cas les plus courants seront traités en TDs et TPs.