Langage C

Langage C avancé : décomposition et pointeurs

Thomas Guyet

AGROCAMPUS-OUEST, Centre de Rennes

Version 2.0, Dernière modification : 18 avril 2012

Table des matières

CHAPITRE

1 Décomposition d'un programme C – récursivité

1.1 Approche descendante

On se replace ici sur le problème de conception algorithmique d'un programme qui doit réaliser une tâche complexe. Pour trouver une solution algorithmique à ce problème, l'approche descendante invite à décomposer cette tâche, qu'on va dire de premier niveau, en tâches de second niveau. Cette première étape permet d'avoir un algorithme abstrait (dans le sens où il ne peut être exécuté par la machine). Dans un second temps, l'analyste programmeur s'intéresse aux tâches de second niveau pour en donner des algorithmes utilisant des tâches de troisième niveau etc. jusqu'à décrire les tâches à partir d'instructions élémentaires.

La devise de l'approche descendante est "diviser pour régner" : le problème est divisé est sous-partie qu'on sait bien faire (*régner*), et qu'on sait combiner en utilisant les structures de contrôle usuelles.

L'approche descendante est une démarche intellectuelle pour aider à construire des algorithmes. Il est intéressant de conserver des traces de cette démarche conservant une trace des tâches de second niveau, troisième niveau ...

- ceci va rendre le programme beaucoup plus lisible que si il était entièrement écris dans une unique fonction **main**(), et donc ce sera beaucoup plus facile d'y trouver des erreurs ou, pour autre programmeur, de comprendre ce programme.
- les briques intermédiaires sont réutilisables pour d'autres programmes. Si ces briques sont bien conçues, elles sont suffisamment génériques pour pouvoir être utilisées à plusieurs endroits dans vos programmes. Plutôt que de refaire à chaque fois le même bout de programme, mieux vaut le faire une fois bien, et le réutiliser ensuite [1] !
- le travail est plus facilement partageable entre plusieurs programmeurs. Si deux programmeurs se sont bien mis d'accord sur les **spécifications** des fonctions, peut importe lequel des programmeurs a effectivement réalisé la fonction, ils pourront utiliser le travail de l'autre sans se soucier de comment c'est fait.

En programmation, les sous-tâches peuvent s'écrire sous la forme de fonctions. Chaque fonction réalise une sous-tâche. Les fichiers `.h` et `.c` permettent ensuite de regrouper des ensembles de fonctions qui sont cohérentes entre elles de sorte à constituer des sortes de modules (on parle de programmation modulaire).

Un programme sera désormais composé comme un ensemble de fonctions [2] qui peuvent s'appeler les unes les autres. La fonction **main**() est une fonction particulière, puisque le compilateur va savoir que c'est par cette fonction que le programme doit commencer ! Pour le **main**(), on parle parfois de *programme principal*. Il serait plus judicieux de parler de *fonction principale*.

1. Il semble ici opportun de rappeler que les fainéants ont bonne presse chez les informaticiens. En effet, mieux vaut utiliser quelque chose qui est déjà fait et qui marche que de le refaire moins bien soit même !

2. En programmation, on distingue parfois les fonctions, des procédures. En C, tout est fonction. Je n'insisterai pas sur la différence.

1.2 Fonctions

Une fonction est une sorte de boîte noire :
- **à l'extérieure**, elle est vue **comme une instruction** qui réalise une tâche de traitement de données (peu importe comment !),
- **à l'intérieur**, c'est **un (mini-)programme** qui implémente la sous-tâche de traitement de l'information.

Exemple 1 - Fonction Somme

Le programme ci-dessous illustre la définition et l'utilisation d'une fonction.

```
1  #include <stdio.h>
2  /*
3  * int somme(int a, int b)
4  *  @brief Calcul de la somme de deux entiers a et b
5  *  @param a, b~: entiers operandes de la somme
6  *  @return la somme des deux entiers a et b.
7  */
8  int somme(int a, int b) {
9    int s=0;
10   printf("a: %d, b: %d\n",a,b);
11   s=a+b;
12   return a+b;
13 }
14
15 int main(void) {
16   int x =2, y =3, z =0;
17   z=somme(x,y);
18   printf("%d\n", z);
19 }
```

Le résultat de l'exécution de ce programme est le suivant :

```
...> ./fonctions
a: 2, b: 3
5
```

À la ligne 17, **somme** *est utilisé comme une instruction dans le programme principal. Et les lignes 9 à 12 décrivent ce que fait cette sous-tâche à partir d'instructions plus élémentaires. Les lignes 2 à 7 sont des commentaires qui décrivent ce que fait la fonction. Ce doit être ici des informations précises et concises qui vont permettre à l'utilisateur de la fonction de savoir exactement ce qui va se passer lors de l'**appel de la fonction** (par exemple, à la ligne 17) sans avoir à lire le contenu de la fonction. On parle de **cartouche**. Il contient les **spécifications de la fonction**.*

1.2.1 Déclaration d'une fonction

Une fonction est un (sous-)programme qui prend des paramètres et **retourne** une valeur. Une fonction est caractérisée par :
- son **profil** : une sorte de nom de la fonction
- son **corps** : c'est l'implémentation de ce que fait la fonction

En C, une fonction se définie de la sorte :

```
type_retour nom_fonction(type1 parametre1, type2 parametre2, ...)
{
  //Corps de la fonction
  ...
}
```

Exemple 2 - Exemple d'une fonction somme

On reprend ici la même fonction que l'exemple 1.2.

```
int somme(int a, int b) // profil de la fonction
{
  //Corps de la fonction
  int s=0;
  printf("a: %d, b: %d\n",a,b);
  s=a+b;
  return a+b;
}
```

Il faut faire le rapprochement entre la définition d'une fonction telle que présentée ci-dessus et une fonction mathématique. Voici la déclaration usuelle d'une fonction mathématique :

$$f : \mathbb{N} \times \mathbb{R} \to \mathbb{R}$$
$$n, x \mapsto 2.x.e^n$$

La première ligne donne, en quelque sorte, le profil de la fonction f : la fonction a deux paramètres (un entier n et un réel x), et que la fonction "retourne" une valeur réelle (à droite de la flèche).
Sur la seconde ligne, vous avez l'implémentation de la fonction : c'est-à-dire la description de ce quelle fait. Ici, cette fonction calcule la valeur $2.x.e^n$.

Remarque 2 - Effet de bord

La fonction de l'exemple 1.2.1 réalise un calcul qui permet de retourner une *valeur utile*. On peut dire que c'est l'attente principale qu'à le programmeur lorsqu'il utilise cette fonction.
Tout ce qui se réalise en plus peut être vu comme un **effet de bord**. L'utilisation de la fonction de l'exemple aura aussi pour conséquence de faire un affichage à l'écran. C'est un effet de bord.
Ces effets doivent être décris dans le cartouche de la fonction, car il ne sont généralement pas attendus par l'utilisateur.

Profil de la fonction

Dans un profil de fonction :
- le contenu des parenthèses indique les **paramètres** de la fonction ; L'**ordre des paramètres** est très important. Ce qui compte pour la liste des paramètres d'une fonction, c'est uniquement la liste des types du paramètre (le nom servira pour le corps de la fonction).
- le type à gauche indique le **type de retour** de la fonction.

Exemple 3 - Exemple d'une fonction somme (suite)

Dans l'exemple de la fonction somme*, le profil indique que la fonction a deux paramètres (un premier paramètre de type entier et un second paramètre de type entier) et retourne une valeur entière.*

Remarque 3

Lorsqu'une fonction n'a pas de paramètre, on peut soit ne rien mettre entre les parenthèses (mais avec des parenthèses tout de même), soit indiquer explicitement qu'il n'a pas de paramètre en indiquant `void`.

Exemple 4 - Fonction et procédure sans paramètres

Profil d'une fonction sans paramètre : `int fonction(void)` *ou* `int fonction()`
Profil d'une procédure sans paramètres (il est nécessaire de mettre le `void` *de type de retour dans tous les cas) :* `void proc(void)` *ou* `void proc()`

Remarque 4

En C, deux fonctions ne peuvent avoir les mêmes noms ET les mêmes paramètres ! Il faut comprendre par "mêmes paramètres" que l'ordre et les *types* des paramètres sont les mêmes (les noms de paramètres n'interviennent pas). Ceci signifie que deux fonctions peuvent avoir le même nom à condition qu'elles n'est pas les mêmes paramètres.

Exemple 5

Exemple de fonctions compatibles :
- `void division ( int a, float b);`
- `void division ( int a);`
- `void division ( float a, int b);`
- `void division ( double a, double b);`

Exemple de fonctions qui ne peuvent pas coexister :
- `void division (int a, int b);`
- `int division (int c, int d);`

Corps de la fonction

Le corps de la fonction qui comporte la description de la tâche réalisée par la fonction est délimité par les accolades "{...}" succédant le profil de la fonction.
Pour le sous-programme qu'est une fonction les paramètres de la fonction correspondent à des **variables locales** qui sont initialisées par les valeurs données par l'appel de la fonction. Ces variables peuvent être utilisées sans être initialisées.
Une fonction doit obligatoirement retourner une valeur dont le type correspond au type de retour de la fonction. Pour indiquer quelle valeur retourner, il faut utiliser l'instruction `return`.
L'instruction `return` :
– finalise l'exécution de la fonction (les instructions suivantes ne seront pas traitées),
– indique la valeur qui sera retournée par la fonction.

Exemple 6

La fonction suivante retournera toujours 0 :

```
int fonction()
{
  int x;
  x=0;
  return x;
  //Partie du code inatteignable
  x=x+1;
  return x;
}
```

L'instruction `return` peut également s'utiliser avec des parenthèses, par exemple :

```
int fonction()
{
  int x;
  x=0;
  return(x);
}
```

Procédure : une fonction sans type de retour

En C, une **procédure** est une sorte de fonction qui ne retourne aucune valeur (par exemple la fonction `printf` fait quelque chose, mais elle ne retourne aucune valeur.). Dans la déclaration de la fonction cela se traduit en disant que le type de retour est `void`.
L'instruction `return` (sans paramètres) peut également être utilisée dans une procédure pour mettre fin à l'exécution de celle-ci.

Exemple 7 - Procédure

```
void maProcedure(int i, float f)
{
  if (i<0) {
```

```
4       return; //on sort de la fonction
5     }
6     printf("%d , %f\n", i,f);
7   }
```

Le cartouche

Un cartouche correspond à la documentation d'une fonction pour des utilisateurs de celle-ci. Il doit être :
– précis
– exhaustif
– concis
Il est très important de faire l'effort de mettre un cartouche sur **toutes** vos fonctions. C'est un effort qui paye à long terme (pour la qualité du code et également pour la formation des programmeurs débutants). Rédiger un cartouche est une étape difficile, mais cela vous amènera à vous poser tout un tas de bonnes questions sur votre implémentation (avez vous bien pensez à tous les cas possibles, quels valeurs de paramètres faut-il interdire, ...). Pour vous aider à la rédaction, *mettez vous à la place d'un programmeur qui veut utiliser votre fonction*, mais sans lire le code. Vous devez alors savoir comment utiliser la fonction et savoir exactement ce qu'elle fait !
Le format plutôt simple que je propose comporte trois champs à décrire dans un bloc de commentaire avant la fonction (format Doxygen [3]) :
– **@brief** : vous y mettrez la description détaillée [4] de ce que fait la fonction, avec également la description des limitations (cas qui ne sont pas pris en compte, ...) et des traitements particuliers.
– **@param** : description d'un paramètre de fonction (quels sont les contraintes pré-supposées dans l'implémentation de la fonction ...)
– **@return** : description de la valeur retournée (en fonction des entrées)

1.2.2 Utilisation d'une fonction

Appel de fonction

Tout l'intérêt d'avoir déclaré une fonction, c'est de l'utiliser dans un programme de niveau de décomposition supérieur. Dans une fonction on peut **faire appel** à d'autres fonctions en passant des valeurs qui servent d'arguments à la fonction.

Remarque 5

Lorsque je désigne ce qu'il y a entre parenthèses pour une fonction, je parle d'"arguments" (*arguments effectifs*) pour l'utilisation d'une fonction, alors que je parle de "paramètres" lorsqu'il s'agit des variables décrites dans le profil d'une fonction (*arguments formels*).

---

3. Doxygen est un générateur automatique de documentation à partir des annotations du code par le programmeur. *cf.* www.doxygen.org
4. Pour Doxygen, le champs **@brief** correspond à la description *courte*, mais peu importe ici ... ce n'est pas le but premier de ce cours !

Exemple 8

Dans l'exemple suivant, les arguments de la fonction somme sont 12 et 23, et le résultat de la fonction somme est affecté à la variable **d**.

```
1 int d;
2 d=somme(12, 23);
3 printf ("%d",d);
```

Lors de cet appel de la fonction somme, voilà ce qui se passe en interne dans la fonction :
- *le paramètre* **a** *prend la valeur 12 (*cf. *Exemple 1.2)*
- *le paramètre* **a** *prend la valeur 23*
- *les instructions de la fonction sont ensuite réalisées (* **printf** *et addition dans la variable* **s***)*
- *au* **return**, *c'est la fin de la fonction*

Le résultat de ce programme sera l'affichage suivant :

```
35
```

À la place des valeurs 12 et 23, on peut avoir envie de mettre directement des variables comme argument de la fonction :

```
1 int d, x=12, y=23;
2 d=somme(x, y);
3 printf("%d",d);
```

Le résultat de ce programme est strictement le même que précédemment. Pour bien comprendre cela, il faut comprendre comment les arguments sont transmis à la fonction.

Transmission des arguments "par valeurs"

En C, la transmission d'un argument à une fonction a toujours lieu **par valeur**. Ceci signifie que **lors d'un appel de fonction par un programme, les valeurs des arguments sont recopiés dans les emplacements mémoire des paramètres de la fonction**.
De même, pour le retour de la fonction, c'est *une valeur* qui est retournée et qui peut être affectée à une variable (du type correspondant au type de retour de la fonction).
Pour bien comprendre ce qui se passe, il faut *imaginer* qu'il y a une étape, avant de rentrer dans la fonction appelée, qui consiste à remplacer le contenu des arguments de la fonction appelée par la *valeur* de ces arguments de sorte qu'on se retrouve dans le cas de l'exemple précédent (**d**=**somme**(12, 23)). Sur l'exemple de programme 1.2.2, on peut imaginer les étapes suivantes lors de l'appel de la fonction à la ligne 9 :

Exemple 9 - Illustration des passages de paramètres

```
1 #include <stdio.h>
2 float division (int a, int b) {
3   return (float )a/(float )b;
4 }
5
6 void main() {
```

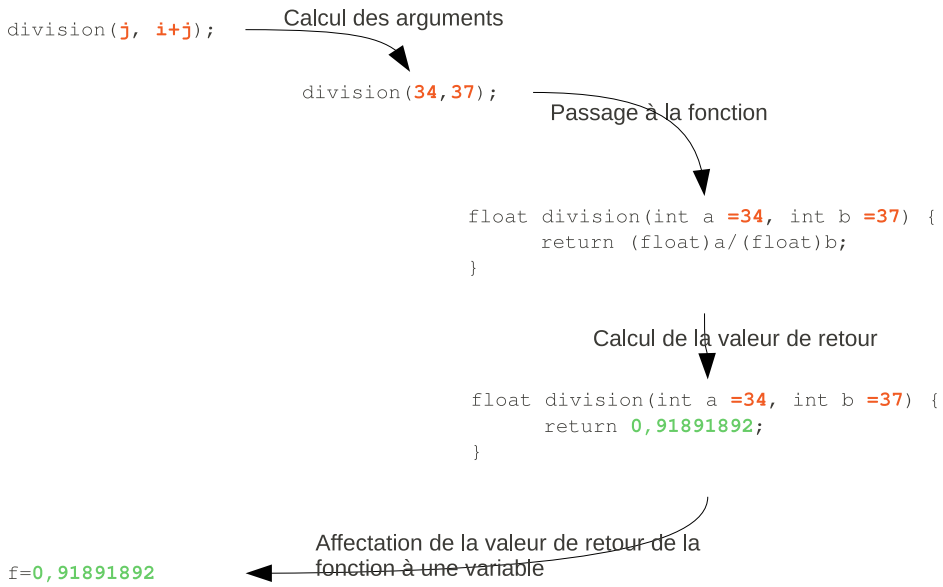


FIGURE 1.1 – Illustration du passage de paramètres par valeurs et du retour de la fonction (appel de fonction : **f**=**division**(**j**,**i**+**j**) avec **i**=3 et **j**=34)

```
7   int i=3, j=34;
8   float f;
9   f=division(j, i+j);
10  printf("result: %f\n",f);
11 }
```

La conséquence du passage par valeur est la suivante : **une fonction ne peut pas modifier la valeur d'un argument**. L'argument n'est pas visible à l'intérieur de la fonction. Toute modification de la valeur d'un paramètre d'une fonction ne sera effectuée que sur la copie, et n'aura aucune incidence sur la valeur de l'argument effectif.

Exemple 10

Dans l'exemple ci-dessous, la fonction n'a aucun effet et l'affichage sera : **n**=2 **p**=5

```
1  #include <stdio.h>
2  void Echange(int a, int b)
3  {
4    int temp;
5    temp = a; a = b; b = temp;
6  }
7
8  int main(void)
9  {
10   int n=2, p=5;
11   Echange(n, p);
12   printf("\n=%d \t p=%d \n",n,p);
13   return 0;
14 }
```

Exercice 1 (Définition d'une fonction) *Pour chacune des questions suivantes, il est demandé d'écrire la fonction et d'écrire une fonction* **main***() qui fera appel à cette fonction.*

Question a) Écrire une fonction qui calcule l'IMC à partir du poids et de la taille d'une personne :

$$IMC = \frac{poids}{taille^2}$$

Question b) Écrire une fonction qui calcule la longueur L d'un câble entre deux pylônes, grâce à la formule :

$$L = a\left(1 + \frac{2}{3}\left(\frac{2f}{a}\right)^2\right)$$

où a est la distance entre les pylônes et **f** *la flèche mesurée perpendiculairement au milieu du câble. Ces deux paramètres seront passés en paramètres de la fonction.*

Question c) Dans le même esprit, écrire une fonction qui calcule le capital A produit par x euros, placés au taux r au bout de n années, avec :

$$A = x\,(1 + r)^n$$

Exercice 2 (Appel de fonction)

Question a) Saisie utilisateur *Modifier le* **main***() de la fonction pour faire en sorte que la valeur d'une des fonctions (au choix) soit saisie par l'utilisateur et que le programme affiche le résultat.*

Question b) Choix de la fonction à utiliser *En réutilisant les fonctions de l'exercice précédent, écrire un programme qui demande si l'utilisateur veut calculer son IMC, la longueur de son fil à linge ou son capital financier. Ensuite, il sera proposé à un utilisateur de donner les valeurs nécessaire au calcul pour finalement afficher le résultat*

1.2.3 Visibilité et durée de vie des variables

Visibilité des variables dans un bloc d'instructions

Definition 1 - Visibilité/Portée

La **visibilité** (ou la **portée**) d'une variable déclarée désigne les endroits du programme où cette variable peut être utilisée.

On voit dans l'exemple 1.2.3 que les variables **i**, **r** et **d** ont été définies et utilisées à plusieurs endroits. Dans le principe, c'est tout à fait possible. Le problème est alors de savoir si, lors de leur invocation, les variables sont bien définies, et de savoir de quelle "version" il s'agit.
Pour cela, il y a un ensemble de règles qui régissent la visibilité des variables. Ces règles sont bâties sur la notion de bloc d'instructions. On retrouve cette notion de bloc pour les fonctions et les structures de contrôles.

Definition 2 - Bloc d'instructions

Un **bloc d'instructions** est un ensemble d'instructions délimitées par des accolades (une ouvrante et une fermante).

Et maintenant, voici les règles en question :
- Une variable est définie dans un bloc *après sa déclaration*,
- Elle est visible dans le bloc et tout les sous-blocs succédant la déclaration mais pas dans son sur-bloc,
- Une variable peut être redéfinie dans un sous-bloc mais pas dans un même bloc,
- La variable redéfinie dans un sous-bloc *masque* l'autre variable dans tous les sous-blocs (règle de proximité supérieure).

Exemple 11 - Visibilité des variables

```
1  #include <stdio.h>
2  int i=4;
3
4  int fonction(int d)
5  {
6    int r=4;
7    printf("dans la fonction %d %d\n", d, r);
8  }
9
10 int main(void)
11 {
12   fonction(i);
13   printf("avant dcl %d\n", i);
14   int i=3;
15   fonction(i);
16   {
17     int i=2;
18     printf("dans le sous-bloc du main %d\n", i);
19   }
20   printf("dans le main: %d %d\n", i, r);
21   return 0;
22 }
```

Tout d'abord, il y aura une erreur à la ligne 20 puisque `r` *n'est pas défini à cet endroit. Sinon, le résultat de l'exécution de ce programme sera l'affichage suivant :*

```
dans la fonction 4 4
avant dcl 4
dans la fonction 3 4
dans le sous-bloc du main 2
dans le main 3
```

Definition 3 - Variable globale

> Une variable globale est une variable définie pour tout le programme. Elle est déclarée en dehors de toute fonction et est accessible depuis toutes les fonctions.

La ligne 2 du code ci-dessus illustre la déclaration de la variable `i` comme variable globale.

Durée de vie d'une variable

La visibilité des variables à des conséquences sur l'utilisation de la mémoire par le programme. Ce qu'il faut retenir en simplifiant les choses c'est que **les variables qui ne sont pas visibles sont supprimées de la mémoire** (leur emplacement mémoire est rendu disponible pour une autre variable).
Les variables globales sont toujours statiques, c'est-à-dire permanentes : elles existent pendant toute la durée de l'exécution. Le système d'exploitation se charge, immédiatement avant l'activation du programme, de les allouer dans un espace mémoire de taille adéquate, éventuellement garni de valeurs initiales.
À l'opposé, les variables locales et les arguments des fonctions sont "automatiques" : l'espace correspondant est alloué lors de l'activation de la fonction ou du bloc d'instructions en question et il est rendu au système lorsque le contrôle quitte cette fonction ou ce bloc.

Exercice 3 (Jouons avec la visibilité des variables (⋆))

Question a) Écrire une fonction qui se contente de comptabiliser le nombre de fois où elle a été appelée en affichant seulement un message "de temps en temps", à savoir :
– *au premier appel : *** appel 1 fois ****
– *au dixième appel : *** appel 10 fois ****
– *au vingtième appel : *** appel 20 fois ****
et ainsi de suite ...
C'est dans la fonction qu'il faut compter le nombre de fois où elle est appelée ! Sinon, c'est triché ! On supposera que le nombre maximal d'appels ne peut dépasser la capacité d'un `long`.

Question b) Écrire le programme qui teste la fonction
Aide : Il faut une variable qui soit capable de se souvenir, en dehors de la fonction, le nombre de fois où le programme est passé dans la fonction.

1.2.4 Séparer la déclaration de l'implémentation

Il est parfois nécessaire de séparer la déclaration de la fonction (c'est-à-dire annoncer au compilateur d'une fonction existe en donnant son profil complet) et son implémentation (c'est-à-dire décrire au compilateur ce qui compose cette fonction).

Exemple 12 - Dépendances circulaires entre fonctions

Dans l'exemple ci-dessous, la fonction `A()` *fait appel à* `B()` *et la fonction* `B()` *fait appel à* `A()`*, si bien que le programmeur ne saurait quelle fonction déclarer en premier (*`A` *ou* `B` *? ?) de sorte que le compilateur, qui lit linéairement le fichier, le comprenne. Si* `A` *était déclaré avant* `B`*, alors* `B` *ne serait pas connus au moment de son invocation, et réciproquement.*

La solution ci-dessous indique au début du fichier qu'une fonction **A** *existe (et donc que le compilo peut l'utiliser) et ensuite* **B** *peut être déclaré en utilisant un appel à la fonction* **A**. *Ensuite, à la fin du fichier, on finit par décrire la fonction* **A**. *Du fait que les profils soient les mêmes, le compilateur va faire la correspondance entre la ligne 1 et la ligne 14.*

```
1  float A(int a, int b); //Declaration de la fonction
2  void B() {
3     ...
4    textbf{A(x,y);}
5     ...
6  }
7
8  void main() {
9    B();
10    printf("Ok\n");
11 }
12
13 //Definition de la fonction A !!
14 float A(int a, int b) {
15   B();
16   return ...;
17 }
```

Remarque 6

Le compilateur vérifie l'existence des fonctions dont on a déclaré le profil lors de l'étape d'édition de liens.

1.2.5 Exercices

Exercice 4 (Fonction simple) *Quel sera l'affichage du programme ci-dessous ?*

```
1  int mafontion(double a, double b) {
2    return (int)(a-b);
3  }
4
5  int main() {
6    double x=2.3, y=4.7;
7    int r=1;
8
9    r=mafontion(x,y);
10    printf("%d\n", r);
11 }
```

Exercice 5 (Passage de paramètres) *Quel sera l'affichage de ce programme ?*
Aide : rappelez vous des mécanismes de passage de paramètres à une fonction en C.

```c
int mafonction(int a) {
  printf("a:%d, ", a);
  a=1;
  return a;
}

int main() {
  int x=0, y=0;
  y = mafonction(x);
  printf("x:%d, y:%d\n", x, y);
  return 0;
}
```

Exercice 6 (Effet du `return` dans une fonction) *Indiquer ce que le programme ci-dessous affichera ?*

```c
int mafonction(int a) {
  int i;
  for(i=0; i<100; i++) {
    if( i>a ) {
      return 0;
    }
    printf("%d\n", i);
  }
  return 1;
}

int main() {
  int ret=mafonction(5);
  printf("%d\n",ret);
  int ret=mafonction(105);
  printf("%d\n",ret);
  return 0;
}
```

1.3 Récursivité

Definition 4 - Fonction récursive

Une **fonction récursive** est une fonction qui s'appelle elle-même soit directement, soit indirectement.

1.3.1 Récursivité simple

Exemple 13

Prenons un exemple de calcul d'une factorielle. Pour rappel, la factorielle de n (notée n!) correspond à la formule suivante :

$$n! = n.(n-1).(n-2)...3.2.1$$

Récursivement, la factorielle peut s'écrire très simplement ainsi :

$$n! = \begin{cases} 1 & si\ n = 1 \\ n.(n-1)! & sinon \end{cases}$$

La fonction correspondante peut alors s'écrire ainsi :

```
#include <stdio.h>
int Factorielle (int n) {
  int Fact;
  if (n>1) {
    Fact = n * Factorielle (n-1);
  } else {
    Fact=1;
  }
  return Fact;
}

int main(void) {
  int n, fn;
  printf ("Donnez un entier positif: ");
  scanf("%d",\&n);
  fn = Factorielle (n);
  printf ("La factorielle de %d est %d\n",n,fn);
  return 0;
}
```

Definition 5 - Cas d'arrêt

Dans une fonction récursive, il doit exister une condition pour laquelle on arrête la récursion (sinon la récursion serait infinie). Cette condition définit le **cas terminal** ou **cas d'arrêt**.

Dans l'exemple, le cas $n = 1$ est appelé le **cas terminal**.
Le tableau suivant représente la séquences des appels de fonctions lorsque l'utilisateur a rentré le chiffre 4 :

Fonction	Ligne de code	n	Fact	fn
main	17 (avant affectation)	4		???
Factorielle(4)	5	4	???	
Factorielle(4)	6 (avant affectation)	4	???	
Factorielle(3)	5	3	???	
Factorielle(3)	6 (avant affectation)	3	???	

Factorielle(2)	5	2	???	
Factorielle(2)	6 (avant affectation)	2	???	
Factorielle(1)	5	1	???	
Factorielle(1)	8	1	1	
Factorielle(1)	9	1	1	
Factorielle(2)	6 (après affectation)	2	2	
Factorielle(2)	9	2	2	
Factorielle(3)	6 (après affectation)	3	6	
Factorielle(3)	9	3	6	
Factorielle(4)	6 (après affectation)	4	24	
Factorielle(4)	9	4	24	
main	17 (après affectation)	4		24

Prenons un second exemple avec la fonction puissance $r : x \to x^n$. Cette fonction peut être définie récursivement :

$$x^{n-1} : \begin{cases} 1 & si\ n = 0 \\ x.x^{n-1} & sinon \end{cases}$$

La fonction correspondant s'écrit ainsi [5] :

```
1 double xn(double x, int n)
2 {
3   if (n==1) return 1;
4   else return x * xn(x, n-1);
5 }
```

1.3.2 Autres formes de récursivité

Récursivité multiple

Une définition récursive peut contenir plus d'un appel récursif. Nous voulons calculer ici les combinaisons C_n^p en se servant de la relation de Pascal :

$$C_n^p = \begin{cases} 1 & si\ p = 0\ ou\ p = n, \\ C_{n-1}^p + C_{n-1}^{p-1} & sinon. \end{cases}$$

La fonction s'écrit alors ainsi :

```
1 int combinaison(int n, int p)
2 {
3   if ( p==0 || p==n ) {
4     return 1;
5   } else {
6     return combinaison(n-1, p) + combinaison(n-1, p-1);
7   }
8 }
```

Les appels récursifs des fonctions devient alors assez complexe. Ici, tout ce passe bien parce qu'on sait qu'on ne tombera jamais deux fois sur le même calcul de combinaison, mais parfois ce peut être le cas. Auquel cas, les mêmes calculs sont répétés plusieurs fois.

5. J'utilise ici une notation très condensée. Ceci rend parfois le C assez illisible pour les novices, mais c'est très pratique, car rapide à écrire !

Récursivité mutuelle

Des définitions sont dites mutuellement récursives si elles dépendent les unes des autres. Ça peut être le cas pour la définition de la parité :

$$pair(n) = \begin{cases} \text{vrai} & \text{si } n = 0 \\ impair(n-1) & \text{sinon} \end{cases}$$

et

$$impair(n) = \begin{cases} \text{vrai} & \text{si } n = 1 \\ pair(n-1) & \text{sinon} \end{cases}$$

Les fonctions vont alors de soit ...

Exercice 7 (Qui pair gagne ...) *Écrire en C les fonctions qui permettent de déterminer récursivement si un nombre est pair ou impair.*

Récursivité imbriquée

La fonction d'*Akermann* est définie comme suit :

$$A(m,n) = \begin{cases} n+1 & si\ m = 0 \\ A(m-1,1) & si\ m > 0\ et\ n = 0 \\ A(m-1,A(m,n-1)) & sinon \end{cases}$$

Dans ce cas, selon les mêmes principes de traduction que précédemment, on peut définir une fonction avec une récursion imbriquée ... c'est presque sans intérêt ! Mais ça a le mérite de montrer que la récursion peut se définir à toutes les sauces et que ça fonctionne très bien en informatique !

1.3.3 Principes et dangers de la récursivité

Principe et intérêt : ce sont les mêmes que ceux de la démonstration par récurrence en mathématiques. On doit avoir :
- un certain nombre de cas dont la résolution est connue, ces "cas simples" formeront les cas d'arrêt de la récursion ;
- un moyen de se ramener d'un cas "compliqué" à un cas "plus simple".

La récursivité permet d'écrire des algorithmes concis et élégants. Il faut noter que tout programme itératif peut être traduit sous une forme récursive.
Difficultés :
- la définition peut être dénuée de sens
- il faut être sûr que l'on retombera toujours sur un cas connu, c'est-à-dire sur un cas d'arrêt ; il nous faut nous assurer que la fonction est complètement définie, c'est-à-dire, qu'elle est définie sur tout son domaine d'applications.

Moyen : existence d'un ordre strict tel que la suite des valeurs successives des arguments invoqués par la définition soit strictement monotone et finit toujours par atteindre une valeur pour laquelle la solution est explicitement définie.
L'algorithme ci-dessous teste si a est un diviseur de b.

```
1 int diviseur(int a, int b)
2 {
3   if(a<=0) {
```

```
4       return -1; //erreur
5     } else {
6       if( a>=b ) return a==b;
7       else return diviseur(a, b-a);
8     }
9   }
```

La suite des valeurs b, $b-a$, $b-2a$, ... est strictement décroissante, car a est strictement positif, et on finit toujours par aboutir à un couple d'arguments (a,b) tel que $b-a$ soit négatif, car défini explicitement (cas d'arrêt). Cette méthode ne permet pas de traiter tous les cas :

```
1   int syracuse(int n)
2   {
3     if( n==0 || n==1 )
4       return 1
5     else {
6       if( (n%2) == 0) {
7         syracuse(n/2);
8       } else {
9         syracuse(3*n+1);
10      }
11    }
12  }
```

Remarque 7

Le résultat de cette fonction est toujours 1 pour tous les entiers (problème ouvert...)

1.3.4 Non-décidabilité de la terminaison

Cette section commence à être un peu plus poussée en terme d'algorithmique, puisqu'on se demande ici si il existe des moyens de déterminer automatiquement si un programme donné termine quand il est exécuté sur un jeu de données.
Et bien, la réponse est ... NON !
Qu'est ce que ça veut dire ?? Ça veut dire que vous ne pouvez pas démontrer (pour tous les algorithmes) si le programme s'arrête ou pas : vous ne pouvez pas montrer qu'il va s'arrêter, mais vous ne pouvez pas non plus démonter qu'il va tourner à l'infini ! Vous n'en saurez rien ... c'est indécidable [6] !
La démonstration de ce résultat étonnant, est une démonstration par l'absurde utilisant un raisonnement dit diagonal (proposé par Godël).
Supposons que la terminaison soit décidable, alors il existe un programme, nommé `termine`, qui vérifie la terminaison d'un programme, sur des données.
À partir de ce programme on conçoit le programme Q suivant (en pseudo-code) :

```
bool Q()
{
  resultat = termine(Q, {})
  while( résultat==vrai ) {
    attendre 1 seconde
  }
  return(resultat);
}
```

6. Notez que la classe des problèmes indécidables est "majoritaire" dans l'ensemble des problèmes "informatisables" !

Supposons que le programme `Q`, qui ne prend pas d'arguments, termine. Donc `termine(Q, {})` renvoie vrai, la deuxième instruction de `Q` boucle indéfiniment et `Q` ne termine pas. Il y a donc contradiction et le programme `Q` ne termine pas.
Donc, `termine(Q, {})` renvoie faux, la deuxième instruction de `Q` ne boucle pas, et le programme `Q` termine normalement. Il y a une nouvelle contradiction : par conséquent, il n'existe pas de programme tel que termine, et donc le problème de la terminaison n'est pas décidable.

1.3.5 Exercices

Exercice 8 (Nombre d'or) *Pour $v_1 = 2$, la suite récurrence suivante converge vers le nombre d'or :*

$$v_n = 1 + \frac{1}{v_{n-1}}.$$

Pour rappel, le nombre d'or vaut $\Phi = \frac{1+\sqrt{5}}{2}$. Il est censé être le rapport de distance "le plus esthétique" pour un cadre rectangulaire (en autres rapport de distances) ...
Construire une fonction qui calcule récursivement les valeurs successives de la suite et qui s'arrête lorsque la distance entre deux valeurs successives est inférieure à ϵ ($|v_{n+1} - v_n| < \epsilon$).

Exercice 9 (Suite de Fibonacci) *Possédant au départ un couple de lapins, combien de couples de lapins obtient-on en douze mois si chaque couple engendre tous les mois un nouveau couple à compter du second mois de son existence ?*
La suite de Fibonacci répond à cette question :

$$u_0 = u_1 = 1$$
$$\forall n \in \mathbb{N}, u_{n+2} = u_n + u_{n+1}$$

Écrire une fonction récursive en C qui calcule la suite de Fibonacci jusqu'à rang k et qui enregistre les valeurs de $(u_n)_{n\in[0,k]}$ dans un fichier pour être affiché avec `gnuplot`.

1.4 Travailler avec plusieurs fichiers `.h` et `.c`

Revenons dans un peu plus de cambouis du langage C ... après avoir réfléchis, ça va détendre !

1.4.1 Séparer les déclarations des fonctions de leurs implémentations

La déclaration et l'implémentation des fonctions peuvent être séparées dans deux types de fichiers :
– fichiers **.h** : il contient uniquement les profils des fonctions (les entêtes ou *headers*), c'est-à-dire la déclaration d'une fonction.
– fichiers **.c** : il contient les corps des fonctions, c'est-à-dire leurs implémentations.
La primitive du préprocesseur `#include` permet d'inclure des fichiers `.h` dans un fichier `.c`. Tout ce passe alors comme le cas de la déclaration préalable d'un profil de fonction (*cf.* section 1.2.4) : les fonctions peuvent être utilisées, et le compilateur fera le lien entre la déclaration et l'implémentation uniquement lors de l'édition de liens.

Exemple 14 - Séparation des fichiers

fonctions.h :

```
#ifndef FONCTIONS_H
#define FONCTIONS_H

int somme(int a, int b);
int soustraction(int a, int b);

#endif
```

main.c :

```
#include "fonctions.h"
#include <stdio.h>
void main() {
   int a;
   int b;
   int sum = somme(a,b);
   int sub = soustraction(b,a);
   printf("%d %d\n", sum, sub);
}
```

fonctions.c :

```
#include "fonction.h"
int somme(int a, int b) {
  return a+b;
}

int soustraction(int a, int b) {
  return a-b;
}
```

Remarque 8

Consulter le chapitre sur la compilation pour savoir comment utiliser la compilation séparée avec `gcc`, et des outils de compilations tels que `make` pour vous aider à enchaîner correctement les compilations.

1.4.2 Programmation modulaire

Maintenant que vous êtes capables de regrouper des fonctions dans des fichiers, il est intéressant de les regrouper intelligemment. Ce qui est intéressant, c'est de prévoir des regroupements qui vont permettre une réutilisation des fichiers dans d'autres développements.
Ces regroupements portent le nom de "module" ... d'où le nom de programmation modulaire !
Pour chaque module, il y a :

– des objets globaux (constantes et définitions de types) et des interfaces de fonctions dans un fichier d'en tête (`.h`) ;

– des objets locaux (variables) et des implémentations de fonctions dans un fichier source (`.c`) ou mieux dans un fichier compilé (`.o`).

Si vous fournissez uniquement le fichier `fonctions.h` et le fichier `fonctions.o`, alors vous permettez aux autres programmeurs d'utiliser votre module tout en conservant le secret sur l'implémentation des fonctions que vous proposez. C'est exactement ce que permettent les librairies usuelles (*e.g.* `libXML` qui offre des fonctions pour manipuler des fichiers `xml`) puisque vous disposez des fichiers d'entêtes (*e.g.* `SAX.h`, `xmlIO.h`, ...) et des librairies qui sont compilées comme des librairies partagées avec des extensions `.so` (*e.g.* `libxml2.so`).

Notez que pour les librairies standard, c'est un peu différent puisqu'elle font maintenant quasiment partie intégrante du C.

On comprend alors maintenant mieux l'importance de bien documenter les fonctions qui ont été écrites. Car dans le cas de modules, on n'a pas l'implémentation pour comprendre ce qui se fait dans une fonction.

1.5 Correction des exercices

Solution à l'exercice 3 Solution aux deux questions :

```
1  int nbappel; /// Dclaration d'une variable globale
2
3  void mafonction() {
4    nbappel++;
5    printf("*** appel %d fois ***\n", nbappel);
6  }
7
8  void main() {
9    nbappel = 0; // Initialisation de la variable globale !
10   for(int i=2; i<14; i++) {
11     mafonction();
12   }
13 }
```

Deux remarques supplémentaires pour la correction :

– Attention de ne pas oublier d'initialiser les variables globales au début du programme. Ceci étant une instruction, elle doit être réalisée dans une fonction. La place qui lui convient donc est au début du main.

– On parle ici d'**effet de bord** : la fonction **mafonction** a un effet de bord sur la variable **nbappel**. L'appel de cette fonction modifie une variable qui n'est pas directement dans son contexte.

Solution à l'exercice 4 Le programme affiche `-2`. Dans la fonction, la différence entre a et b donne $2,3-4,7=-2,4$. Le transtypage du `double` en `int` réalise une troncature, d'où le résultat.

Solution à l'exercice 5 Le programme affiche `a:0, x:0, y:0`.

Solution à l'exercice 6 Le programme affiche :

```
1
2
3
4
5
```

```
0
1
2
3
(...)
98
99
1
```

Solution à l'exercice 7 Voici la solution. Les deux fonctions étant croisées, il est nécessaire d'en déclarer une avant de l'implémenter.

```
int impair(int n); //declaration de la fonction impair en attendant sa definition

int pair(int n) {
  if( n==0 ) {
    return 1;
  } else  {
    return impair(n-1);
  }
}

int impair(int n) {
  if( n==1 ) {
    return 1;
  } else  {
    return pair(n-1);
  }
}
```

Solution à l'exercice 8 On déduit directement de la relation de récursion établies dans l'énoncé la fonction suivante :

```
double nbor(int n) {
  if(n==1) {
    return 2.0;
  } else {
    return 1+1/nbor(n-1);
  }
}
```

Solution à l'exercice 9 Pour obtenir la solution, il faut renverser la formule de récursion et en déduire le code ci-dessous :

```
uint fibonacci(uint m)
{
  if (m <= 1)
    return 1;
  return fibonacci(m-2) + fibonacci(m-1) ;
}

void main() {
  uint val=30;
  uint f = fibonacci(val);
  printf("fibo(%d)=%d\n", val, f);
}
```

Voici une version non-récursive du calcul (c'est à dire iteratif !) de la suite de fibonnacci :

```
1  int fibonacci(int n) {
2    int a = 0;
3    int b = 1;
4    int sum;
5    int i;
6
7    for (i=0;i<n;i++) {
8      printf("%d\n",a);
9      sum = a + b;
10     a = b;
11     b = sum;
12   }
13   return b;
14 }
```

CHAPITRE

2 Les pointeurs et allocation dynamique

Enfin nous voilà à LA partie intéressante ! Celle pour laquelle on fait du C, ou bien qu'on déteste ce langage ... la manipulation des pointeurs et l'auto-gestion de la mémoire. Bientôt vous pourrez vous ranger dans l'une ou l'autre des catégories de programmeurs !

2.1 Les pointeurs

La notion de pointeur est difficile à acquérir. Il sera donc à peu près normal que cela vous bloque au début ... Leur usage est néanmoins fondamental en C ! Bien sûr, par mimétisme de bout de code, un grand nombre de programmeur amateur se débrouillent pour écrire du code en C sans en connaître toute la portée ! Espérons que vous puissiez dépasser cette étape lors de ce cours ...

2.1.1 Définition et déclaration

Definition 6 - Pointeur

Un pointeur est une variable qui contient l'adresse d'un emplacement mémoire.
Cet emplacement mémoire correspond à une variable du programme. Le **type du pointeur** donne le type de la variable pointée.
En mémoire, un pointeur est codé comme un entier positif sur un mot mémoire.

Rappelons nous qu'une variable désigne un emplacement dans la mémoire de l'ordinateur. Nous avons mentionné [1] que chaque emplacement peut être désigné par un indice (sa position sur la longue bande d'emplacements qu'est la mémoire) et par un taille (la place nécessaire pour coder une variable en mémoire qui dépend du type de la variable). Pour désigner un emplacement de la mémoire, il y a donc deux possibilités :
– donner le nom associé à cet emplacement, *i.e.* le nom de la variable
– donner l'adresse de cet emplacement, *i.e.* la valeur du pointeur qui pointe sur cet emplacement.
Ici, toute la subtilité ne sera perçue que si vous faite un claire différence entre "le nom" d'une variable et "la valeur" d'une variable.

Remarque 9

Lors qu'on parle d'un "pointeur", on devrait conserver en mémoire qu'il s'agit plutôt d'une "variable pointeur".

1. *cf.* cours sur l'architecture des machines, et en particulier sur l'organisation de la mémoire. Il serait intéressant de relire cette partie du cours si vous ne l'avez plus en mémoire.

Soit **i** une variable de type **int** et **pi** une variable pointeur qui pointe sur **i**, alors on peut représenter le contenu de la mémoire par le schéma illustratif ci-dessous. La variable pointeur désignée sous le nom **pi** contient la valeur `@106` (lu "adresse numéro 106"). Cette adresse correspond à l'emplacement mémoire de la variable **i**. **pi** pointe **donc** sur la variable **i**.

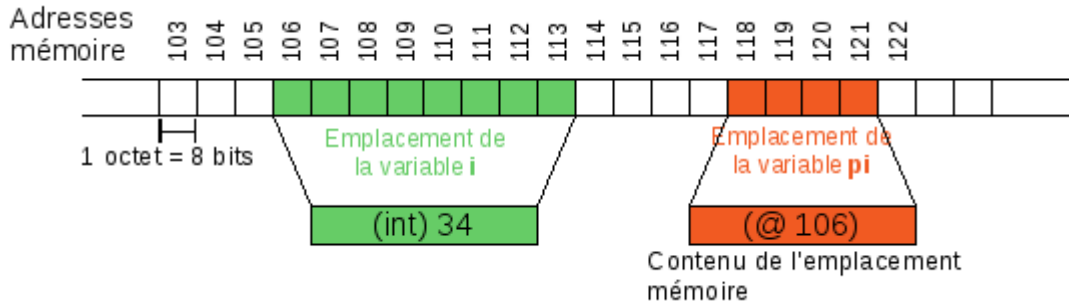


FIGURE 2.1 – Illustration de la représentation en mémoire d'un pointeur (**pi**) et de sa variable pointée (**i**).

Pour des raisons pratiques et mnémotechniques **les pointeurs sont typés**. De la sorte, le programmeur (et le compilateur) sait quel type de donnée contient l'adresse pointée par un pointeur et également (c'est très important) la taille de l'emplacement mémoire désigné.

Pour déclarer une variable pointeur, il faut utiliser le symbole "$*$" avant le nom de la variable. Le type de la variable sur laquelle pointe la variable pointeur est désigné par le type en début de ligne. Voici quelques exemples de déclaration de pointeur :

```
int *pi ; //Pointeur sur un int
double *pf0; //Pointeur sur un double
char *s ; //Pointeur sur un caractere
void *p; //Pointeur pur
```

Pour mieux comprendre comment fonctionne la déclaration d'un pointeur, voici, ci-dessous, un second exemple dans lequel on déclare **un entier i** et **deux pointeurs pj et pi sur des entiers**. On voit que l'étoile doit se mettre avant le nom de la variable (et non après le type !)

```
int *pi, i, *pj; //pi et pj sont des pointeurs, mais i est un entier !!
```

Allons directement dans le cambouis : dans la mesure où un pointeur est une variable, il également possible d'avoir un pointeur sur cette variable. On a alors un pointeur de pointeur ! Dans l'exemple ci-dessous, on a un pointeur sur un pointeur d'entier :

```
int **ppi;
```

! Attention !

Par convention personnelle, je donne des noms commençant par "**p**" aux pointeurs.

2.1.2 Manipulation des pointeurs sur variables

Référencement (opérateur &) et affectation d'une variable pointeur

Definition 7 - Référencement

L'opérateur de référencement, noté &, est un opérateur unaire qui donne l'adresse mémoire d'une variable. L'opérateur se place avant le nom de la variable :

&**variable** ou &(**variable**)

Par exemple, sur la figure 2.1 , on a :
– &**i** donne la valeur `@(106)`
– &**pi** donne la valeur `@(118)`
Cette opération est indispensable pour affecter notre variable pointeur avec l'adresse d'une autre variable.

Exemple 15

```
1 int i = 34;
2 int *pi;
3 pi = &i; // pi contient l'adresse de i
```

À la ligne 3, la variable **pi** *est affectée avec pour valeur, l'adresse de la variable* **i**. *On se retrouve alors exactement dans la cas de la figure 2.1.*

Déréférencement (opérateur *)

Definition 8 - Déréférencement

L'opérateur de déréférencement, noté $*$, est un opérateur unaire **sur un pointeur** qui désigne l'emplacement mémoire pointé par la variable pointeur **p**. L'opérateur se place avant le nom de la variable pointeur :

$*$**p** ou $*$(**p**)

Le type de la variable $*$**p** est donné en *enlevant une étoile* à la déclaration de **p**.

Exemple 16

Par exemple, si on déclare la variable **g** *suivante :*
`int ***g;`
alors $*$**g** *est de type* `int**`, *et* $**$**g** *est de type* `int*` *et enfin* $***$**g** *est de type* `int` *(il n'est pas possible n'aller plus loin !)*

Exemple 17

Reprenons maintenant l'exemple de la figure 2.1 : il est important de voir que ∗pi *désigne le* ***même*** *emplacement mémoire que* i.
Ainsi, si on poursuit l'exemple avec les instructions après la ligne 3 ci-dessous. à la ligne 4, (∗pi) *est une variable (de type* int*), qui désigne le même emplacement que* i*, et qui donc vaut 34. Donc,* j *se voit attribué la valeur 34 !*
À la ligne 6, (∗pi) *donc désigne le même emplacement que* i*, et donc si on affecte cette variable avec une nouvelle valeur, on change le contenu de l'emplacement commun avec* i*. Du coup, à la ligne 8, on affiche 0 !*

```
1 int i = 34, j=0;
2 int *pi;
3 pi = &i;
4 j = *pi;
5   //ICI j contient 34
6 *pi = 0;
7   //ICI i vaut maintenant 0
8 printf("%d\n",i);
```

Si vous avez bien compris cet exemple, vous êtes à même de poursuivre la lecture du cours ...
Quelques erreurs classiques :
– Utilisation d'un pointeur non-initialisé

```
1 int *pi;
2 *pi = 13;
```

Ici, pas de problème de type : à ligne 2, j'affecte bien un int (∗pi est un int) avec une valeur entière. Le compilo va être très content. Mais ça ne va pas marché lors de l'exécution (*segmentation fault*) ! Vous n'avez pas initialisé la variable pi !!! La conséquence est que vous ne savez pas sur quel emplacement mémoire le pointeur pointe ... Mais ça ne pose pas de problème de conscience au compilo ... Alors admettons que pi ait pour valeur latente 234 (du fait de la valeurs des bits existants précédemment) ! Alors lors de l'instruction de la ligne 2, il va modifier l'emplacement mémoire de l'adresse 234 sans se demander si il en a le droit ... et les conséquences seront nécessairement dramatiques [2].
Conclusion, **il faut s'assurer que vos pointeurs pointent sur quelque chose en mémoire avant de les utiliser.**
– Comparaison de pointeurs à la place de la comparaison de valeurs
Dans le cas ci-dessous, le programme n'affiche rien !

```
1 int i =13, j =13, *pi, *pj;
2 *pi = &i;
3 *pj = &j;
4 if( pi == pj ) {
5   printf("coucou");
6 }
```

2. Dramatiques, mais pas trop quand même sur vos ordinateurs. En effet, tout système d'exploitation qui se respecte dispose d'un MMU. Cette partie du système fait attention à ce qu'un programme n'écrive pas partout et n'importe comment en mémoire... les programmeurs savent que tous les programmeurs font se genre d'erreur ! Normalement, vous n'aller pas tout cracher ... Mais, on rencontre parfois des systèmes sans MMU, c'est le cas par exemple du système de la nintendo DS, ce qui explique pourquoi elle a été si facile à craquer !

Dans la condition, vous comparez deux pointeurs, c'est-à-dire deux adresses mémoires. Ici, **pi** et **pj** désignent respectivement les emplacements mémoire de **i** et de **j**. Donc la condition est fausse. Pour comparer les valeurs des variables désignées par les pointeurs, il faut faire quelque chose de la sorte :

```
if ( *pi == *pj )
```

Le pointeur NULL

Definition 9 - NULL

Le pointeur **NULL** (ou valant 0 en conversion en entiers) désigne un pointeur qui ne pointe sur rien.

Il est bien d'initialiser tous ces pointeurs à **NULL**, ou de remettre à **NULL** tous les pointeurs dont vous ne servez plus ! Par convention, la valeur NULL sert également de valeur de retour d'erreur pour certaines fonctions.
Les opérateurs de l'arithmétique entière pouvant être utilisés pour les pointeurs, il est possible d'écrire ce genre d'expression :

```
1 int *p=NULL;
2 ...
3 // Oprations sur p
4 ...
5 if( p==NULL ) {
6   printf("rien faire\n");
7 } else \{
8   printf("Valeur pointee par p~: %d\n", *p);
9 }
```

2.1.3 Pointeurs et tableaux

Un tableau désigne tout un ensemble d'emplacements mémoire intéressants pour celui qui l'a déclaré. Tout naturellement, les pointeurs vont permettre de désigner ces emplacements mémoire.
On commence cette section en introduisant des opérateurs sur les pointeurs. Ensuite, on fera le lien avec les tableaux.

Opérateurs arithmétiques

En interne, un pointeur est un nombre puisqu'il s'agit d'une adresse mémoire ! Alors, les opérateurs arithmétiques usuels fonctionnent très bien. En particulier, les opérateurs d'incrémentation et de décrémentation ($++$, $--$, $+=$ et $-=$), et les opérateurs d'addition et soustraction usuels.
L'exemple ci-dessous illustre la signification de ces opérateurs pour des pointeurs.

```
pa + 1; // dsigne le pointeur qui pointe sur l'espace mmoire suivant
pa + 2; // dsigne le pointeur qui pointe sur l'espace mmoire encore suivant ...
```

La figure 2.2 illustre ce que signifie exactement "l'espace mémoire suivant". Supposons que l'adresse en mémoire de **a** est `@(106)`. Si, dans le cas du haut, **a** est un **long**, (et **long** ***pa**), alors l'adresse pointée par (**pa**+2) est 106+16 = `@(132)`. En revanche, si **a** est de type **int** alors l'adresse pointée par (**pa**+2) sera 106+8 = `@(116)`.

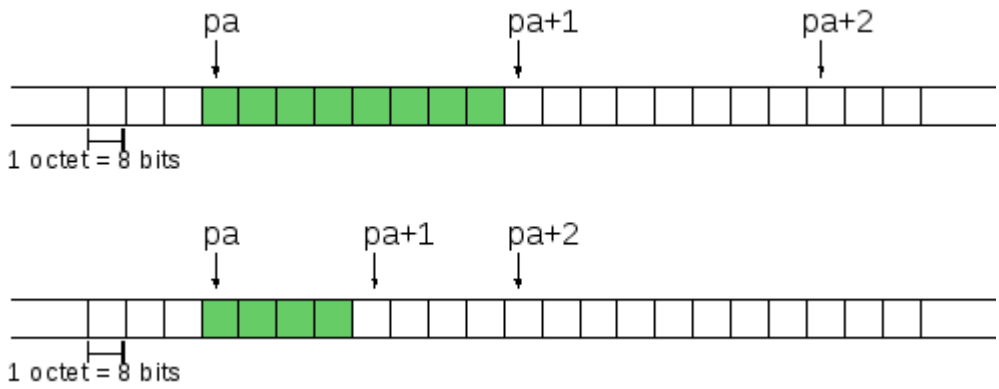


FIGURE 2.2 – Illustration de l'arithmétique des pointeurs. Ici, **pa** est un pointeur qui pointe avec l'emplacement mémoire de la variable **a**. Au dessus, **a** est de type **long**, et en dessous, **a** est de type **int**.

Il faut comprendre que l'adresse de (**pa**+2) est calculée par le compilateur en sachant la taille en mémoire du type de l'objet sur lequel pointe le pointeur (en fait, il utilise le type déclaré du pointeur).

Remarque 10

Ceci fonctionne non seulement avec les types de bases (**int**, **float**, **double**), mais avec toutes les structures de données, y compris celles que vous auriez pu définir vous même (*cf.* la partie du cours sur la définition de types complexes).

Remarque 11

Notez qu'il est également possible d'écrire des choses horribles comme des sommes ou des différences de pointeurs ... mais c'est là où les libertés qu'offre le C lui nuisent ! Mieux vaut ne jamais faire cela ...

Application aux tableaux

Comparons maintenant la Figure 2.2 ci-dessus et la Figure ci-dessous qui vous rappelle l'organisation en mémoire d'un tableau unidimensionnel.
On constate que si un pointeur **pa** désigne un élément du tableau, alors **pa**+1 va désigner l'élément suivant de ce tableau. Et donc, les opérateurs d'incrémentation ou de décrémentation permettent de faire des opérations naturelles sur les indices de tableaux.

Un pointeur va pouvoir servir d'indice sur un élément du tableau !

Maintenant, si on regarde de plus près le pointeur sur la variable **tab**, on constate que le début de l'emplacement mémoire désigné par ce pointeur est le même que le début de l'emplacement mémoire désigné par le premier élément du tableau. La seule différence, importante, est que le type de pointeur est différent : dans le premier cas, l'emplacement mémoire désigné s'étend sur tout le tableau (*e.g.* type `int [10]`), et dans un second l'emplacement mémoire ne désigne qu'un élément du tableau (*e.g.* type `int`) : le typage du pointeur à alors toute son importance !
Prenons maintenant un petit exemple pour voir les manipulations. Soit les déclarations suivantes :

```
int tab[10];
int *pi;
```

On peut initialiser le pointeur pour qu'il pointe sur un élément du tableau ainsi :

```
pi = &z[5]; // pi pointe sur z[5]
```

Ensuite, il est possible de faire des opérations sur le pointeur qui désignera alors d'autres éléments du tableau :
- **pi**++ : **pi** désignera alors le 6ème élément du tableau
- **pi**– : **pi** désignera alors le 4ème élément du tableau (si on repart de la ligne 3 de l'exemple !)
- **pi**+2 : **pi** désignera alors le 7ème élément du tableau
- **pi**+7 : **pi** désignera alors un élément hors du tableau (Attention ! !)

Maintenant, on peut très bien travailler sur les valeurs du tableau en utilisant l'opérateur de déréférencement :
- ∗(**pi**) : donne la valeur de l'élément du tableau pointé
- ∗(**pi**+1) : donne la valeur de l'élément pointé suivant dans le tableau
- Plus généralement, ∗(**pi**+**a**) donne la valeur de l'élément à **a** case décalées de l'élément pointé par **pi** !

! Attention !

Si **pi** est un pointeur sur le premier élément d'un tableau **tab**, et **i** un entier alors : **p**[**i**] est équivalent à ∗(**p**+**i**)
Attention, il faut également que le type du pointeur corresponde au type du tableau, sinon les décalages ne se feront pas correctement !

Il est nécessaire d'avoir une référence absolue (initialisée) telle que **pi** pour pourvoir désigner des éléments d'un tableau. En utilisant la seconde remarque tirée de l'observation des Figures précédentes, on peut facilement récupérer le pointeur sur le premier élément du tableau. Il y a juste à s'assurer du bon typage du pointeur.
En C un tableau est un pointeur (avec quelques contraintes sémantiques supplémentaires sur la longueur du tableau). Ceci permet d'écrire l'initialisation d'un pointeur à partir d'une variable tableau.
À la suite de l'exemple précédent, on peut ainsi écrire :

```
pi=tab;
```

pi sera alors un pointeur qui désigne le premier élément du tableau.

Remarque 12

Sur les tableaux à dimension multiple, ça ne marche pas aussi simplement à cause du typage de pointeur. L'exemple suivant illustre la façon de procéder pour récupérer un pointeur sur le premier élément d'un tableau d'entiers à 2 dimension. On se contente d'indiquer au compilateur qu'il faut utiliser le début de tableau de **tab** comme début d'emplacement mémoire, mais qu'il faut l'interpréter comme un emplacement qui contient un entier : ceci est entièrement de la responsabilité du programmeur, et donc un peu dangereux!

```
int tab [10][5], *pi;
pi=(int *)tab;
```

Exercice 10 (À vos crayons !) *Faire un dessin pour illustrant les emplacements mémoire utilisés dans le programme suivant, et donner la valeur de* **x**.

```
int a [10];
int *pa;
pa = &a[0];
int x=*pa;
```

Exemple 18 - Initialisation d'un tableau

---

*L'exemple ci-dessous illustre l'**initialisation d'un tableau en utilisant des pointeur** :*

```
1 int a [10];
2 int *pa=a; // pa pointe sur le premier element du tableau
3 for (int i=0; i<10;i++) {
4   // initialisation du i-eme element du tableau:
5   *(pa + i)= 1;
6 }
7 //ICI: pa pointe sur le premier element du tableau
```

l'exemple suivant est encore plus rapide :

```
1 pa = &a[0]; // pa pointe sur le premier element du tableau
2 for(int i=0; i<10; i++) \{
3   // initialisation du i-eme element du tableau:
4   *pa = 1;
5   pa++;
6 }
7 //ICI: pa pointe sur le dernier element du tableau
```

Ce second exemple est plus rapide que le premier grâce à l'utilisation de l'opérateur **pa**++*. Avec cet opérateur, il faut une seule instruction pour pointer sur l'indice suivant, alors que dans le premier cas, il faut d'une part faire l'incrémentation de* **i** *et, d'autre part, l'addition de* **i** *avec* **pa**. *Donc on a divisé par deux le nombre d'instructions nécessaire pour cette opération !*
Attention néanmoins à la différence en sortie de boucle : **pa** *ne pointe pas vers les mêmes objets dans les deux cas !*

Exercice 11 (Pointeurs) *Indiquer quels seront les affichages produits par ce programme :*

```c
#include <stdio.h>
int a[]={0,1,2,3,4};
int main()
{
  int i, *p;
  for( i=0; i<=4; i++) {
    printf("a[%d] = %d, ", i, a[i]);
  }
  printf("\n");
  for( p=&a[0]; p<=&a[4]; p++) {
    printf("*p = %d, ", *p);
  }
  printf("\n");
  for( p=&a[0], i=1; i<=5; i++) {
    printf("p[%d] = %d, ", i, p[i]);
  }
  printf("\n");
  for( p=a, i=0; p+i<=a+4; p++, i++) {
    printf("*(p+%d) = %d, ", i, *(p+i));
  }
  printf("\n");
}
```

Exercice 12 (Voisinage dans un tableau à deux dimensions) *On considère un tableau d'entier à 2 dimensions de taille $n \times m$ (`int tab[n][m]`). Soit `pi` un pointeur sur un élément de ce tableau (`int *pi`). Indiquer, à partir de `pi`, comment désigner simplement les éléments suivants du tableau :*

- *l'élément à droite de celui désigné par `pi`,*
- *l'élément à gauche de celui désigné par `pi`,*
- *l'élément au dessus de celui désigné par `pi`,*
- *l'élément, en diagonale, en dessous et à droite de celui désigné par `pi`*

Aide : On considère qu'on se trouve bien au milieu du tableau, et pas sur un bord ! La taille du tableau est une donnée utile.

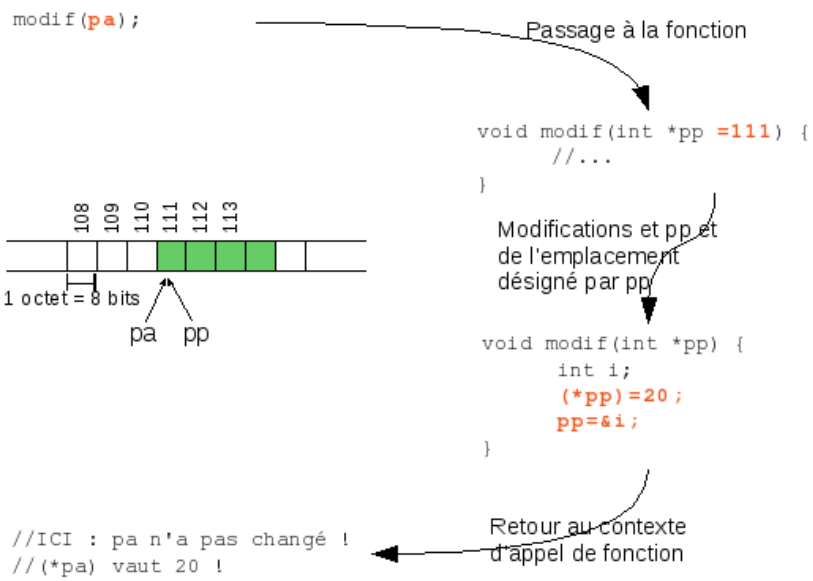
2.1.4 Utiliser les pointeurs dans les paramètres de fonction

Il faut se rappeler que les fonctions utilisent un mécanisme de passage de paramètre par valeurs : c'est à dire que les valeurs mis en arguments d'une fonction (lors de sont appel) sont "recopiés localement" pour l'exécution d'une fonction. Si, dans la fonction, on modifie la valeur de la variable paramètre, cela n'aura aucun effet sur la variable-argument qui a été utilisée lors de l'appel de la fonction.
Lorsqu'on passe un pointeur en paramètre d'une fonction, c'est la même chose : la valeur du pointeur, donc son adresse, est recopiée. En revanche, l'emplacement mémoire qui était désigné par l'argument-pointeur est le même que celui qui est désigné par le paramètre-pointeur et donc **si vous modifiez le contenu de l'emplacement désigné par le pointeur à l'intérieur d'une fonction, cette modification aura les mêmes conséquences à l'extérieur de la fonction appelée.**
Mieux vaut un petit schéma qu'un long discours, voici une illustration pour le programme ci-dessous :

```c
void modif(int *pp)
```

```
{
  int i=0;
  *pp=20;
  pp=&i;
}

int main()
{
  int val=10;
  int *pa=&val;
  modif(pa);
  printf("%d\n",val); //Affiche 20
}
```



Voici deux programmes qui semblent à première vue très similaire, mais le résultat va être très différent. Dans le premier, les paramètres de la fonction **echanger** sont passés **par valeurs**, alors que dans le second cas on passe les paramètres **par pointeurs**.

Exemple 19

Premier cas : affiche 12, 16

```
//Fonction avec passage des parametres par valeurs
void echanger(int a, int b) {
  int temp;
  temp = x;
  x = y;
  y = x;
}

int main(void) {
  int a =12, b=16;
  echanger(a, b);
  printf("%d, %d\n", a,b);
}
```

void int int
int

int void
int

Quel est l'intérêt du passage de pointeurs comme paramètres de fonctions :

1. on peut éviter de recopier les données lors d'un appel de fonction. Lorsque les arguments sont des types simples, la recopie n'est pas longue, mais si vous aviez une structure de données lourde, ça peut être inefficace de la recopier systématiquement.
2. on peut travailler par effet de bords, c'est-à-dire produire des effets dans une fonction qui ont des impactes à l'extérieur de la fonction (pas vraiment conseillé, mais parfois pratique !)

2.1.5 Exercices

Exercice 13 (Manipulation des pointeurs)

Question a) Opérations élémentaires sur les pointeurs

```
1  void main()
2  {
3          int A = 1;
4          int B = 2;
5          int C = 3;
6          int *P1, *P2;
7          P1=&A;
8          P2=&C;
9          *P1=(*P2)++;
10         P1=P2;
11         P2=&B;
12         *P1-=*P2;
13         ++*P2;
14         *P1*=*P2;
15         A=++*P2**P1;
16         P1=&A;
17         *P2=*P1/=*P2;
18         return 0;
19 }
```

Copiez le tableau suivant et complétez-le pour chaque instruction du programme ci-dessus.

	A	*B*	*C*	*P1*	*P2*
Init	*1*	*2*	*3*	/	/
P1=&A	*1*	*2*	*3*	*&A*	
P2=&C					
**P1=(*P2)++*					
P1=P2					
P2=&B					
**P1-=*P2*					
*++*P2*					
**P1*=*P2*					
*A=++*P2**P1*					
P1=&A					
**P2=*P1/=*P2*					

Question b) arithmétique des pointeurs *On déclare les variables suivantes :*

```
int A[] = {12, 23, 34, 45, 56, 67, 78, 89, 90};
int *P;
P = A;
```

Quelles valeurs ou adresses fournissent ces expressions (ou les erreurs) :

1. **P+2*
2. **(P+2)*
3. *&P+1*
4. *&A[4]-3*
5. *A+3*
6. *&A[7]-P*
7. *P+(*P-10)*
8. **(P+*(P+8)-A[7])*

Exercice 14 (incrémentation ou décrementation ??) *Voici une fonction très simple.*

```
void incrdecr(long *T, int n)
{
  int i=0;
  long *p = T;
  for (; i<n; n++) {
    (*p)--;
    p++;
  }
}
```

Question a) Que fait cette fonction ?

Exercice 15 (Tableaux de pointeurs)

Question a) Représentation des objets en présence *Dans la fonction* **main**(), *on déclare et initialise les variables suivantes :*

```
1  char string0[]="Chaine A";
2  char string1[]="Chaine B";
3  char string2[]="Chaine C";
4
5  char *temp=0; // pointeur sur une chaine de caracteres
6
7  char *ps[3]; // Tableau de taille 3 contenant des pointeurs sur ces chaines de caracteres
8  ps[0]=string0;
9  ps[1]=string1;
10 ps[2]=string2;
```

Représenter par une boîte chacune des chaines de caractères et par une seconde boîte le tableau **ps** *et faites des flèches pour indiquer ce sur quoi pointe chaque pointeur.*

Question b) Fonction d'affichage *Écrire une fonction d'affichage avec le profil suivant :* **void affichage**(**char** ***tab**[3]) *qui pourra prendre en paramètre* **ps** *et qui affichera les 3 chaînes de caractères pointées par* **tab** *dans l'ordre du tableau.*

Question c) Évolution *Pour chacune des instructions suivantes, refaire le schéma précédent en modifiant les flèches ! Indiquer le résultat des affichages.*
NB : On pourra également représenter également le pointeur **temp**.

```
1  temp=ps[1];
2  ps[1]=ps[2];
3  ps[2]=temp;
4  affichage(ps);
5  (*temp)[7]='D';
6  ps[0]=temp;
7  affichage(ps);
```

2.2 Gestion dynamique de la mémoire

Jusque là, les pointeurs permettaient de désigner des emplacements mémoires qui correspondent à des variables déclarées dans votre programme. La gestion dynamique de la mémoire va vous permettre de déclarer des emplacements mémoires purement désignés par des pointeurs (sans avoir de référence symbolique dans le programme). Notez que cela modifie la définition initiale d'un pointeur (Définition 2.1.1.) qui ne doit plus être vu que comme une adresse sur un emplacement mémoire.
L'intérêt va encore se montrer important pour les tableaux. En particulier, cela va lever l'impossibilité que nous avions jusque là de déclarer des tableaux dont la taille n'est pas connue par le programmeur mais pourrait l'être lors de l'exécution.
Nous allons présenter trois fonctions de la librairie **stdlib.h** :

– Fonction **malloc** : fonction d'allocation de mémoire

```
void *malloc( size_t s)
```

– Fonction **free** : fonction de libération de mémoire

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

– Fonction sizeof : fonction qui calcule la place mémoire nécessaire d'un type de données

```
size_t sizeof(type_t t)
```

Exemple 20 - Allocation de mémoire

Exemple d'utilisation typique de ces fonctions :

```
int *p=0;
p = (int *)malloc( sizeof(int) );
*p=3400; // initialisation de l'emplacement memoire
... // utilisation de l'emplacement memoire
free(p); //libere l'emplacement mmoire
```

2.2.1 Allocation dynamique de la mémoire

Definition 10 - Allocation dynamique

L'allocation de la mémoire consiste à réserver, auprès du système d'exploitation, un emplacement de la mémoire pour le besoin de l'exécution d'un programme. L'allocation est dite dynamique lorsqu'elle se fait lors de l'exécution du programme et qu'elle n'est que partiellement prévisible lors de la programmation.

La fonction **malloc** alloue de l'espace mémoire de la taille demandée et retourne un *pointeur pur* (type void *) sur le début de l'espace mémoire alloué.
La taille demandée est fournie en paramètre de la fonction **malloc**. Cette taille ne peut être inventée par vos soins ! Vous disposer de la fonction sizeof (...) pour répondre correctement à la question de la taille mémoire.
La fonction sizeof donne la taille mémoire d'une variable à partir de son type. Par exemple :
– sizeof(double) donne la taille mémoire d'un double,
– sizeof(uint) donne la taille mémoire d'un unsigned integer.
– sizeof(montype) donne la taille mémoire d'une structure de données personnelle (nommée **montype**)

Allocation d'une variable

Utilisation dans le cas de l'allocation d'un emplacement pour une valeur de type **montype** (peut être un type de base ou une structure de données) :

```
montype *p = (montype *)malloc( sizeof(montype) );
```

Comme la fonction **malloc** retourne un pointeur pur, il est nécessaire de faire la conversion explicite vers le type de pointeur désiré pour pouvoir affecter la variable **p**.

Allocation d'un tableau

Si on veut allouer un tableau de données, il suffit d'utiliser le nombre d'éléments que doit comporter le tableau pour en déduire la taille de l'emplacement mémoire (par une simple multiplication).
Pour allouer dynamiquement un tableau d'entiers de taille 10, on aura :

```
int *tabint = (int *)malloc( sizeof(int) * 10 );
```

et pour tableau de taille `x`, il est toujours possible d'écrire :

```
int *tabint = (int *)malloc( sizeof(int) * x );
```

L'intérêt des pointeurs est ici mis en évidence dans le second exemple, car s'il est possible d'écrire `int tabint[10]` pour déclarer un tableau de 10 entier comme dans le premier exemple, il est impossible d'écrire en C : `int tabint[x]` !!
Si vous souhaitez déclarer dynamiquement un tableau à dimension multiple, il faut soit allouer le tout en 1 fois (solution efficace en calcul) et "voir" ce tableau linéaire comme un tableau à plusieurs dimension en jouant avec les indices, soit faire un tableau de ligne manuellement.
Dans la mesure où le pointeur désigne un emplacement mémoire qui ne réfère à une variable du programme, alors l'initialisation (et plus généralement l'affectation) du contenu de l'emplacement mémoire doit se faire au travers du pointeur. Par exemple :

```
int *pi=(int *)malloc( sizeof(int) );
*pi=0;
```

Limitation dans l'allocation de la mémoire

C'est le système d'exploitation (le MMU du système d'exploitation) qui est en charge d'allouer la mémoire aux programmes qui s'exécutent sur un ordinateur. Mais la ressource n'est pas infinie ! Alors il est possible que si vous êtes trop gourmand, alors la fonction malloc retourne la valeur NULL auquel cas, le système ne vous aura pas trouver la mémoire nécessaire pour faire fonctionner votre programme.
Pour information, le maximum de mémoire utilisable par un processus est un paramètre du noyau Linux (qui peut être changé à la compilation du noyau). De mémoire, un noyau standard propose 4Go de mémoire avant d'exploser ! Y'a de la marge ... mais pas tant que ça ! Pour des applications scientifiques lourdes, ça peut rapidement se montrer insuffisant (sans compter les courantes fuites de mémoire ... qui seront présentées dans la section suivante).

Definition 11 - Calcul en place

On parle de calcul en place (ou un algorithme en place) lorsque l'utilisation de la mémoire est bornée par une valeur prédéfinie pour toutes les exécutions du programme (et en particulier pour toutes les données d'entrée). C'est une bonne propriété d'un algorithme qu'on obtient généralement au dépend de son efficacité.

2.2.2 Exemple complet

Exemple d'application de l'allocation dynamique de mémoire : récupérer des saisies d'un utilisateur en nombre non-prédéfinie. On veut, par exemple, calculer la moyenne des nombres fournis par un

utilisateur, sans lui imposer le nombre de nombres à entrer. Le programme va d'abord demander combien de nombres l'utilisateur veut entrer puis en faire la saisie dans un tableau.

Ce problème ne peut pas se résoudre sans allocation dynamique, car on n'impose aucune limite de nombre à l'utilisateur. Il faut donc attribuer une taille au tableau de valeurs lors de la saisie de la première information.

Hop... on y va :

```c
#include <stdlib.h>
#include <stdio.h>
int main(void)
{
  int tailleTableau=0;
  int nbreSaisis=0;
  double *floatTable=NULL;
  //Recuperation de la taille du tableau
   printf("Donner le nombre de nombres a rentrer\n");
  scanf("%d", &tailleTableau);
   if(tailleTableau <=0) \{
     printf("\t => saisie incorrecte\n");
    return 1;
  }
  //ICI tailleTableau >0
  //Allocation de la memoire pour le tableau de taille tailleTableau
  floatTable = (double *)malloc( sizeof(double)*tailleTableau );
   if( floatTable==NULL ) {
     printf("\t => Une erreur est survenue lors de l'allocation de la memoire\n");
    return 1;
  }
  //ICI: floatTable pointe sur un emplacement memoire de tailleTableau doubles !
  double *p=floatTable; // p designe un pointeur sur l'emplacement remplir
  while( nbreSaisis < tailleTableau) {
     printf("Entrer votre nombre (%d/%d)\n",nbreSaisis, tailleTableau);
     if( scanf("%lf", p)~!= 1 )
       //Recuperation de la valeur (que l'on place directement dans l'emplacement pointe par p, c'est a dire l
           'emplacement courant a remplir !}
       //ICI la saisie n'a pas permis de reconnaitre un double
        printf("\terreur de saisie\n");
       return 1;
     }
    p++; //On passe a l'emplacement suivant
    nbreSaisis++; //Incrementation
  }
  //ICI toutes les donnes ont ete saisies
   printf("merci\n");

  //TODO HERE : traiter les donnees du tableau !

  free(floatTable); // On libere la memoire
  floatTable=NULL; // On place floatTable NULL pour indiquer que le pointeur ne pointe plus sur rien !
  return 0;
}
```

Avec cet exemple, vous êtes en mesure d'allouer un tableau de taille variable. Mais vous devez tout de même savoir quel sera la taille du tableau avant de le remplir ... Les structures comme les listes chaînées permettent de s'abstraire de cette contrainte.

2.2.3 Libération de la mémoire

Une fois la mémoire utilisée, il est nécessaire de la libérer proprement ! C'est un engagement implicite d'un programme par rapport au système d'exploitation : dès que la mémoire allouée dynamiquement n'est plus utile, il est poli de la rendre à la communauté de processus pour qu'ils en bénéficient.

Definition 12 - Fuite de mémoire

Lorsqu'un programme a une fuite de mémoire, c'est qu'il alloue de plus en plus de mémoire sans la libérer.

Un tout petit oubli de libération de mémoire par le programmeur peut avoir de grandes conséquences si le programme passe des milliers de fois sur cet oubli. Ceci peut aller jusqu'à la défaillance de tout le système d'exploitation.
Pour libérer de la mémoire, rien de plus simple, il suffit d'appeler la fonction **free** en passant en paramètre un pointeur qui désigne l'emplacement mémoire à libérer et voilà (*cf.* exemple précédent, ligne 47) !
Une fois la mémoire libérée, l'emplacement n'appartient plus au programme, il est alors prudent de passer la valeur du pointeur à **NULL** de sorte à ne plus garder trace d'un lien avec cet emplacement mémoire (*cf.* exemple précédent, ligne 41).

! Attention !

Il est possible de perdre la trace d'un emplacement mémoire alloué, auquel cas vous serez dans l'impossibilité de le libérer plus tard.

```
1 int i=0;
2 int *pi=(int *)malloc( sizeof(int) );
3 *pi=20;
4 pi= &i; // cette instruction fait perdre l'adresse de l'emplacement mmoire allou
      dynamiquement!
```

! Attention !

Les emplacements mémoires libérés sont perdus **définitivement** ! Si vous essayez de lire dans un emplacement mémoire libéré, c'est comme lire dans un emplacement non-alloué : vous ne savez pas ce qu'il contient, et quelles seront les conséquences de sa modification !

Remarque 13 - État en mémoire des emplacements libérés

La libération mémoire laisse la mémoire dans l'état où elle est. Il est de la responsabilité du programmeur d'*effacer* les informations pour qu'elles ne soient plus lisibles.
En programmation défensive[3], on replace toute la mémoire à **NULL** avant de la libérer.

3. La programmation défensive est un mode de programmation où on vérifie plutôt deux fois qu'une que tout ce passe normalement au dépend de l'efficacité calculatoire.

Tout l'intérêt et le problème de l'allocation et la libération dynamique de mémoire, c'est que c'est un travail dont le programmeur à l'entière responsabilité.

Exercice 16 (Allocation et libération mémoire) *On donne le programme ci-dessous :*

```
1  void main () {
2      int *ptr;
3      int a, b, c;
4      int tab[3 ];
5      int i;
6
7      a = 1 ;
8      b = 2 ;
9      c = 3 ;
10     tab[0 ] = 5 ;
11     tab[1 ] = 8 ;
12     tab[2 ] = 12 ;
13
14     /* Faire en sorte que ptr reference une variable existante */
15     ptr = &a;
16     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
17
18     ptr = &b;
19     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
20
21     ptr = &tab[1 ];
22     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
23
24     /* Modifier la valeur de ce qui est reference par ptr */
25     ptr = &c;
26     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
27
28     c = 4 ;
29     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
30
31     /* Reserver une zone memoire referencee par ptr */
32     ptr = (int*) malloc(sizeof(int ));
33     *ptr = 10 ;
34     printf ("Valeur actuelle de la donnee referencee par ptr = %d\n",*ptr);
35     free(ptr);
36
37     /* Reserver une zone memoire reference par ptr II : tableau dynamique */
38     ptr = (int*) malloc(5 *sizeof(int ));
39     *ptr = 1 ;
40     *(ptr+1 ) = 5 ;
41     *(ptr+2 ) = 10 ;
42     *(ptr+3 ) = 15 ;
43     *(ptr+4 ) = 20 ;
44     for (i=0 ; i < 5 ; i++) {
45         printf ("Valeur actuelle de l'element %d du tableau dynamique dfini par ptr = %d\n",i,
               *(ptr+i));
46     }
47     free(ptr);
48 }
```

Question a) Commentez ce programme, et essayez de prévoir le comportement de la mémoire (faire des dessins).

Question b) Copiez, compilez et faites tourner ce programme.

Question c) Réécrivez ce programme en affichant également les adresses des cases mémoires affichées. Qu'en déduisez vous ?
Aide : pour faire afficher l'adresse mémoire d'un pointeur **p** *en code hexadécimal, utiliser l'instruction* **printf** *("%x\n",* **p***).*

2.3 Correction des exercices

Solution à l'exercice 11 **a** est un tableau d'entiers mis en variable globale, **i** est un entier et **p** un pointeur sur un entier.
Pour chaque boucle, les affichages se font sur une ligne et entre les boucles il y a des sauts de lignes.
La première boucle est un simple parcours du tableau, rien de bien méchant.
Dans la seconde boucle, l'itération s'effectue sur le pointeur **p** de la position &**a**[0], c'est à dire la première case du tableau (équivalent à **a**) jusqu'à la dernière case. On utilise l'opérateur **p**++ pour passer d'une case à l'autre du tableau. Comme on affiche ∗**p**, on affiche à chaque fois ce sur quoi pointe **p**, soit les case successive du tableau **a**.
Dans la troisième boucle, **p** est initialisé en pointant sur la première case du tableau et on fait évoluer **i** de 1 à 5. Comme on affiche **p**[**i**], on affiche la valeur pointé par **p** déclalé de **i** cases. On a donc un décalage dans le parcours du tableau et le dernier affichage sera très aléatoire ... c'est un dépassement de tableau !
Dans la quatrième boucle, on utilise toujours un décalage, mais écrit différemment ... cette fois-ci tout fonctionne (attention, bien s'assurer de comprendre tous les détails, il y a plein d'astuces dans ce code !)
Les affichages seront les suivants :

```
a[0] = 0, a[1] = 1, a[2] = 2, a[3] = 3, a[4] = 4,
*p = 0, *p = 1, *p = 2, *p = 3, *p = 4,
p[1] = 1, p[2] = 2, p[3] = 3, p[4] = 4, p[5] = 12542,
*(p+0) = 0, *(p+1) = 1, *(p+2) = 2, *(p+3) = 3, *(p+4) = 4,
```

Solution à l'exercice 12 Avant de répondre à cette question, il faut convenir de ce qui est ligne et de ce qui est colonne. Ici, on suppose que **tab** comporte n lignes et m colonnes. **tab**[**k**] désigne donc la k-ieme ligne (un tableau de m valeurs).
– l'élément à droite de celui désigné par **pi** : **pi**+1,
– l'élément à gauche de celui désigné par **pi** : **pi**−1,
– l'élément au dessus de celui désigné par **pi** : **pi**−**m**,
– l'élément, en diagonale, en dessous et à droite de celui désigné par **pi** : **pi**+**m**+1

Solution à l'exercice 13

Question a)

Question b)
*P désigne le premier élément de A, donc

1. *P+2 = 14
2. *(P+2) = 34
3. &P+1 est l'adresse à côté celle de la variable P (valeur ???)
4. &A[4]-3 est l'adresse du second élément du tableau A
5. A+3 (ceci doit provoquer une erreur et au mieux désigne l'adresse du quatrièmement élément de A : 1er élément décalé de 3)
6. &A[7]-P : ici c'est la différence d'adresse entre le 8ème élément de A et le premier élément de A (P), donc 7 (NB : la différence d'adresse n'a généralement pas de sens)
7. P+(*P-10) : de nouveau, c'est absurde, on additionne des adresses avec des valeurs ... mais admettons *P-10 donne 2, donc P+2 est l'adresse du troisième élément de A !
8. *(P+*(P+8)-A[7]) : *(P+8)-A[7] donne 89-78 = 11, on cherche donc *(P+11) : on tombe en dehors du tableau ...

C'est vraiment absurde comme exercice ... je vous l'accorde ! Mais celui qui a tout bon à tout compris aux pointeurs !!

Solution à l'exercice 14 Ce programme décremente les `n` premières valeurs d'un tableau commençant à l'emplacement mémoire `T`, en supposant que `T` contienne au moins `n` valeurs.

Solution à l'exercice 16 L'affichage du programme est le suivant :

```
Valeur actuelle de la donnee referencee par ptr = 1
Valeur actuelle de la donnee referencee par ptr = 2
Valeur actuelle de la donnee referencee par ptr = 8
Valeur actuelle de la donnee referencee par ptr = 3
Valeur actuelle de la donnee referencee par ptr = 4
Valeur actuelle de la donnee referencee par ptr = 10
Valeur actuelle de l'element 0 du tableau dynamique défini par ptr = 1
Valeur actuelle de l'element 1 du tableau dynamique défini par ptr = 5
Valeur actuelle de l'element 2 du tableau dynamique défini par ptr = 10
Valeur actuelle de l'element 3 du tableau dynamique défini par ptr = 15
Valeur actuelle de l'element 4 du tableau dynamique défini par ptr = 20
```