I.U.T. d'Amiens
département Informatique
Année universitaire 2022-2023

Module R3.05
Programmation système en C sous Unix

amel.sghaier@u-picardie.fr
arnaud.clerentin@u-picardie.fr

Chap. 1 Rappels de langage C

A. Les pointeurs

1. *Introduction*

Ce qui fait aujourd'hui de C le langage préféré de la performance (jeux vidéos, applications embarquées multimédia, systèmes d'exploitation, calcul scientifique, etc.), c'est qu'il permet à l'utilisateur de gérer lui-même la mémoire. On peut en effet accéder directement à des zones de la mémoire par l'intermédiaire de leurs **adresses**.

2. *L'opérateur adresse « & »*

Pour connaître l'adresse mémoire d'une variable, on utilise l'opérateur adresse « & ».

Par exemple, si *num* est une variable de type entier, alors &*num* est l'adresse mémoire à laquelle se trouve cette variable.

3. *Variable de type pointeur*

On appelle **pointeur** une variable qui peut contenir une adresse mémoire.

On déclare un pointeur en ajoutant le caractère étoile « * » avant le nom de la variable.

Par exemple, l'instruction suivante :

```
int *ptr;
```

déclare une variable nommée *ptr* destinée à contenir l'adresse de début d'une zone de mémoire contenant un entier. On dit que *ptr* est **un pointeur sur entier**.

Ainsi, l'instruction :

```
ptr = &num;
```

est correcte : *ptr* contient l'adresse mémoire de la variable *num*.

4. *L'opérateur d'indirection « * »*

L'opérateur « * » permet d'accéder à la valeur située à l'adresse contenue dans un pointeur.

Par exemple, **ptr* désigne l'entier qui se trouve à cette adresse, c'est à dire *num*.

5. *Le cas des tableaux*

Les tableaux sont une suite de variables **de même type** situées dans un espace contigu en mémoire.

La syntaxe de déclaration est la suivante :

```
type nom_tableau [nombre_cases];
```

Par exemple, le code suivant déclare un tableau d'entiers nommé **T** et comportant quatre cases :

```
int T[4];     // Attention : les cases sont numérotées à partir de 0
```

En réalité, **T** est un **pointeur** qui contient l'adresse du début de la zone mémoire allouée au tableau (autrement dit, **T** pointe sur le premier élément du tableau **T**[0]).

B. Fonctions

1. *Introduction*

Définir une fonction consiste à associer à un identificateur un groupe d'instructions, qui plus tard sera activé par un simple appel à l'identificateur.

Cette notion de fonction est fondamentale car elle est à la base de **l'analyse descendante**.

L'analyse descendante

Analyser et programmer la résolution d'un problème en « descendant » consiste à aller du général au particulier. Tout d'abord on détermine les grandes actions que doit faire le programme, en précisant l'état des objets reçus et rendus par chacun de ces traitements particuliers. Puis, de la même façon, chaque traitement particulier est étudié et fait l'objet d'une procédure, qui elle même sera divisée en sous-procédures, etc.

La programmation descendante est donc l'art de programmer en appliquant l'adage « diviser pour résoudre ».

Cette approche de la programmation induit des programmes très structurés, dont les principales qualités sont :

- la **lisibilité** : le programme est composé de modules ne dépassant pas les vingt à trente lignes, l'algorithme se détachant alors clairement des tâches de moindres importances
- **l'optimisation** : les groupes d'instructions utilisés de nombreuses fois sur des objets différents constitueront des modules propres
- la **fiabilité** : les liaisons entre modules sont étroites et limitées, et l'information n'est pas disséminée dans tout le programme
- la **portabilité** : un programme est composé de briques logicielles réutilisables pour d'autres programmes.

2. *Définition d'une fonction*

La syntaxe de la définition d'une fonction est la suivante :

```
type_retour nom_fonction(type arg1, ..., type argn)
{
    déclaration des variables locales à la fonction
    instructions
}
```

où :

- *type_retour* est le type de la donnée renvoyée par la fonction
- *nom_fonction* est le nom de la fonction
- *argi* est la déclaration d'un argument formel suivant la syntaxe de déclaration d'une variable

Ces trois éléments constitue **l'en-tête** (ou prototype ou signature) de la fonction.

Le **corps** de la fonction est délimité par des accolades et comporte des éventuelles déclarations de variables et une suite d'instructions.

3. *Appel d'une fonction*

Les appels de fonction en C tiennent compte du nombre et du type des arguments formels.

Ave les définitions précédentes, l'appel d'une fonction respecte la syntaxe :

```
nom_fonction(arg1, ..., argn)
```

Ainsi, lors de l'appel d'une fonction, à chaque argument formel de la fonction doit correspondre un argument réel, la mise en correspondance s'établissant d'après l'ordre d'apparition de chaque paramètre.

4. *Passage par adresse et par valeur*

En C, absolument tous les paramètres formels d'une fonction sont des **copies** des paramètres réels passés lors de l'appel à cette fonction. Cela signifie que l'on ne travaille jamais directement sur les éléments passés lors de l'appel, mais avec d'autres variables qui contiennent les mêmes valeurs que les originales. Ainsi, les modification apportées dans la fonction aux paramètres formels ne se répercutent pas sur les paramètres réels. Ce mode de passage de paramètre s'appelle le passage **par valeur**.

Si on souhaite que les modifications apportées sur les paramètres formels s'appliquent sur les paramètres réels, on doit réaliser un passage **par adresse**. En C, cette notion de *passage par adresse* porte bien son nom en C puisqu'on passe en argument **l'adresse mémoire** du paramètre réel que l'on récupère dans un **pointeur** au niveau du paramètre formel correspondant.

5. *Le cas particulier des tableaux*

Exemple. Créer un sous-programme *foo* qui retourne la longueur de la chaîne de caractères passée en paramètre.

Rappel. Le type string n'existant pas en C, les chaînes de caractères sont représentées par un **tableau de caractères**. Une chaîne de caractères se termine par le caractère « fin de chaîne » \0.

```
…      foo(…                )
{




}
```

A noter. La notation suivante est équivalente (et sans doute plus lisible) :

```
…      foo(…                )
```

Réaliser un appel de cette fonction à partir du programme principal ci-dessous :

```
#include…

int main()
{
     char str[30];
     int longueur;

     scanf("%s", str);       // Noter l'absence de &…

     …   = foo(…     );
     …
}
```