Chapitre 7 : Conception et implémentation des algorithmes. Les sous-programmes

Comme nous l'avons vu au chapitre 1 :

- Les instructions proprement dites, celles qui sont traduites par le compilateur et qui seront exécutées par le processeur lors du lancement du programme exécutable, s'écrivent dans des **sous-programmes (procédures ou fonctions)**.
- Il y a toujours au moins un sous-programme : la fonction principale *main()*. C'est le point d'entrée du programme.
- L'exécution d'un programme commence toujours par la première instruction du *main()*.

Pour obtenir un code bien construit, les différents traitements et algorithmes identifiés lors de l'analyse du cahier des charges et des étapes de conception doivent être codés dans des sous-programmes distincts.

I. Qu'est-ce qu'un sous-programme ?

- Un sous-programme est un bloc d'instructions :
  - **codé séparément, indépendamment des autres**. A part la fonction principale *main()*, un sous-programme seul ne peut pas être exécuté. Pour qu'un sous-programme soit exécuté, il faut qu'il soit **appelé** à partir d'un *main()* ou par un autre sous-programme lui-même appelé à partir d'un *main()*.
  - auquel on donne **un nom**. Pour exécuter la suite d'instructions qu'il contient, il suffira de **l'appeler par son nom**.
  - qu'on peut **paramétrer** : on peut lui envoyer des **données (entrées)** sur lesquels il effectuera ses traitements.
  - qui peut fournir des **résultats**. On parle alors de **fonctions**.

En C une fonction ne peut retourner qu'un seul résultat.

Un sous-programme qui ne retourne pas de résultat est appelé une **procédure**.

- Un sous-programme possède
  - ✓ Un en-tête comportant :
    - Le type du résultat qu'il retourne ou ***void*** s'il ne retourne rien.
    - Son nom.
    - Entre parenthèses : la liste de ses éventuels paramètres avec leurs types.
  - ✓ Son bloc d'instructions (entre accolades).

- L'instruction ***return*** permet de mettre fin à l'exécution du sous-programme et dans le cas d'une fonction de retourner un résultat à l'appelant.

- Vous avez déjà utilisé des sous-programmes, par exemple les fonctions d'affichage *printf* et de saisie *scanf* qui sont codées dans la librairie standard *stdio*.

II. Importance des sous-programmes. Pourquoi ne pas tout coder dans la fonction principale ?

- **Pour rendre le code plus clair, faciliter la relecture et le débogage.**

- **Pour éviter les redondances de code.** Si une même suite d'instructions doit être exécutées plusieurs fois à des endroits différents du programme, il vaut mieux la coder une seule fois dans un sous-programme et l'appeler quand on en a besoin.

- **Pour faciliter la programmation en équipe.** Chaque programmeur doit pouvoir développer et tester ses parties de code le plus indépendamment possible des autres.

- **Pour garantir la réutilisabilité des algorithmes,** il faut les coder dans des sous-programmes bien paramétrés et séparer au maximum les aspects IHM (saisies, affichages) de la logique de traitement. Un même sous-programme peut être utilisé dans plusieurs programmes différents, grâce au concept de bibliothèque.

III. Exemple – Activité

On souhaite écrire un petit programme permettant à des écoliers de s'entrainer aux multiplications. Ce programme doit **afficher un menu** proposant de :

1. **Afficher la table de multiplication**
2. **Afficher les n premiers multiples d'un entier x**
3. **S'entrainer aux multiplications**
4. **Quitter**

Si l'utilisateur choisit la deuxième option, le programme lui demande de saisir les valeurs de n et x.
Si l'utilisateur choisit la troisième option, le programme lui propose une série de 10 multiplications avec des chiffres choisis « au hasard ». Le programme demande à l'utilisateur de saisir le résultat et le vérifie. Il compte le nombre d'erreurs effectuées. A la fin du test, il affiche le nombre d'erreurs et s'il est supérieur à 5, il propose à l'utilisateur de réviser sa table en lui affichant.

Téléchargez le fichier « activite9_multiplications.c » et étudiez le code proposé.

1. Conception : Décomposer le problème – Identifier les algorithmes/sous-programmes à coder.

Le **découpage en sous-programmes** s'est d'abord basé sur les fonctionnalités proposées :

Fonctionnalité	**Données en entrée**	**Données en sortie**	**Traitements à effectuer**
Afficher le menu et récupérer le choix de l'utilisateur		le choix (entier)	Affichages Saisie
Afficher la table de multiplication			Des multiplications et des affichages
Afficher les n premiers multiples d'un entier x	n et x (entiers)		Des multiplications et des affichages
S'entrainer		Nombre d'erreurs (entier)	Recommencer 10 fois : – Proposer une multiplication et vérifier la saisie. – Augmenter le nombre d'erreurs de 1 si la saisie est incorrecte

La fonctionnalité « S'entrainer » fait elle-même appel à un traitement non élémentaire « Proposer et vérifier une multiplication ». Ce traitement a été codé dans un sous-programme à part.

On obtient les sous-programmes suivants :

Nom du sous-programme	**rôle**	**Données en entrée**	**Données en sortie**	**Traitements à effectuer**
menu	Afficher le menu et récupérer le choix de l'utilisateur		le choix (entier)	Affichages Saisie
afficherTableMulti	Afficher la table de multiplication			Des multiplications et des affichages
afficherMultiples	Afficher les n premiers multiples d'un entier x	n et x (entiers)		Des multiplications et des affichages
entrainerMulti	S'entrainer		Nombre d'erreurs (entier)	Recommencer 10 fois : Proposer et vérifier une multiplication
testerMulti	Proposer et vérifier une multiplication		0 (saisie incorrecte) ou 1 (saisie correcte)	– Tirer deux chiffres « au hasard » – Demander de saisir leur produit – Si la saisie est correcte retourner 1, sinon retourner 0

2. Implémentation des sous-programmes

Activité 9. Etudiez-le code « activite9 multiplications.c » en lisant bien les commentaires, excécutez-le, documentez-vous et répondez aux questions suivantes sur boostcamp :

- Au lancement du programme, quelle est la première instruction exécutée ? Indiquez le numéro de ligne.
- Quel est le bloc d'instructions répété tant que l'utilisateur ne choisit pas l'option « Quitter » ? Indiquez les numéros de lignes des accolades ouvrante et fermante du bloc sous la forme numéro_ouvrante-numéro_fermante.
- Pour faire son choix dans le menu, l'utilisateur doit saisir un petit entier. Quel est le type de la variable utilisée pour stocker le choix de l'utilisateur ?
- Si l'utilisateur choisit l'option 1, le *main()* appelle le sous-programme *afficherTableMulti()* (ligne 103). Le programme effectue ce qu'on appelle un branchement :
  - Il mémorise là où il en est dans l'exécution de l'appelant.
  - Il « saute » aux instructions du sous-programme appelé et les exécute.
  - Une fois l'exécution du sous-programme appelé terminée, il revient là où il en était dans l'appelant et passe à l'instruction suivante.

Exemple :

Le programme commence par la première instruction du *main()* (ligne 6).

```
2   void toto(){
3       printf("\ntoto s'exécute");
4   }
5   int main(){
6       int x;
7       printf("branchement : appel de toto");
8       toto();
9       printf("\ntoto est terminé. Le main() reprend où il en était.");
10      x=5;
11      return 0;
12  }
```

Ligne 8 : appel du sous-programme *toto()* : Le programme « se branche » sur les instructions de *toto()* (ligne 3). Une fois les instructions de *toto()* terminées, le programme revient où il en était dans l'appelant (ligne 8) et passe à l'instruction suivante (ligne 9).

Si l'utilisateur choisit l'option 1, quelle est l'instruction exécutée après le test de la ligne 103 ? Indiquez le numéro de la ligne d'instruction.

- Si l'utilisateur choisit l'option 2, quelle est l'instruction exécutée après le test de la ligne 105 ? Indiquez le numéro de la ligne d'instruction.

- Selon vous, si l'utilisateur a choisi l'option 1, que se passe-t-il après l'affichage de la table de multiplication (une fois l'appel du sous-programme *afficherTableMulti()* terminé) ?

  rappel : *if ... else* signifie *si ... sinon*

  - ☐ Le programme effectue le test ligne 105.
  - ☐ Le programme n'effectue pas le test ligne 105 car le test ligne 103 était vrai. Il ne passe donc pas dans le bloc *else*. Il va directement ligne 124 vérifier si le choix était différent de 1. Comme c'est le cas, il réaffiche le menu.

- Après avoir étudié les en-têtes des sous-programmes (lignes 5, 30, 41, 59 et 73), indiquez les en-têtes corrects pour :
  - un sous-programme *toto1* qui ne reçoit aucun paramètre et qui ne retourne aucun résultat
  - un sous-programme *toto2* qui reçoit en paramètres un entier et un caractère et qui ne retourne aucun résultat.
  - un sous-programme *moyenne* qui reçoit en paramètres deux entiers et qui retourne leur moyenne.
  - un sous-programme *toto4* sans paramètre qui retourne deux réels.

- Un **graphe d'appels** montre l'enchainement des appels de sous-programmes. On montre avec des flèches pleines quel sous-programme appelle quels autres sous-programmes. On peut également indiquer avec des flèches creuses les retours de fonction.
  Un graphe d'appels sert à la **compréhension du programme.** On peut ajouter plus d'informations si nécessaire (types des paramètres, condition d'appels …)
  Voici un graphe d'appels détaillé du programme « activite9_multiplications.c » :

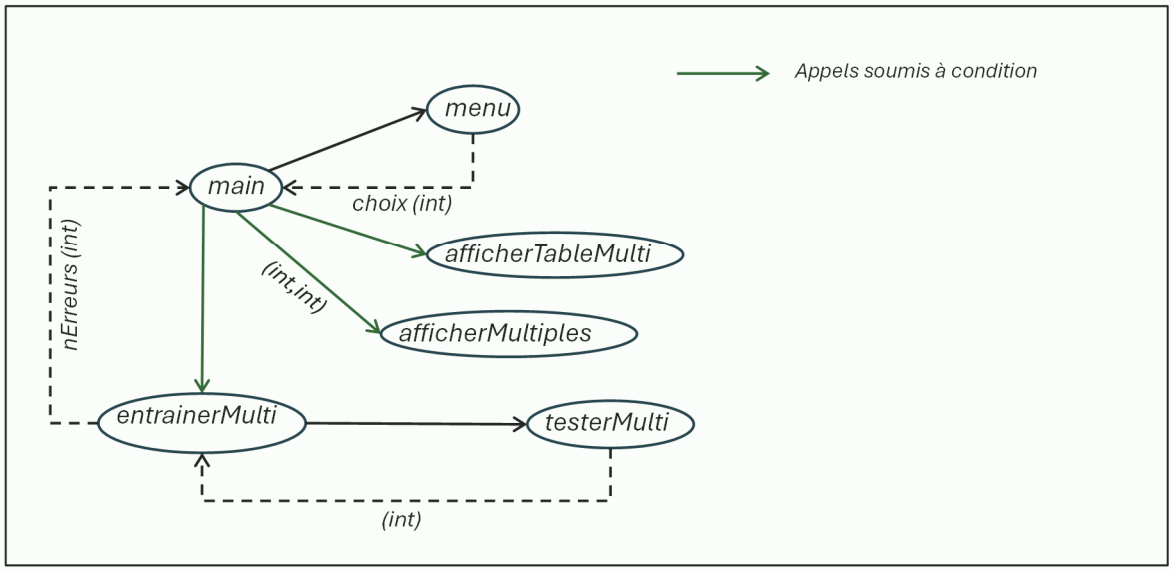

Figure 6 : graphe d'appels

En étudiant le code, répondez aux questions suivantes :

- Lors de l'exécution du programme, combien de fois au minimum le sous-programme *menu* est-il appelé ?
- Lors de l'exécution du programme, combien de fois sera appelé le sous-programme *afficherTableMulti* si l'utilisateur choisit l'option 3, fait 4 erreurs de calcul, puis choisit de quitter le programme ?

- Il n'y a pas de hasard en programmation classique car on utilise des algorithmes déterministes donc prédictibles. Il n'y a donc pas de nombres aléatoires. Mais on peut « simuler » des nombres qui vont sembler aléatoires. On parle de nombre pseudo-aléatoires qui sont générés par une formule de suite récurrente. Le premier terme de la suite est appelé la « graine » du générateur.

  Documentez-vous sur les **nombres pseudo-aléatoires** et sur les sous-programmes :
  - ✓ ***time*** (de la bibliothèque *time*)
  - ✓ ***srand*** et ***rand*** (de la bibliothèque *stdlib*)

  Identifiez à quoi sert chaque sous-programme, leurs paramètres, leurs valeurs de retour.
  Notez un exemple d'utilisation pour générer un nombre pseudo-aléatoire entier compris entre -10 et 10.
  Puis répondez aux questions suivantes :

- A quoi sert le sous-programme *srand* de la bibliothèque *stdlib* ?
  - ☐ A initialiser la graine du générateur (le premier terme de la suite)
  - ☐ A générer un entier pseudo-aléatoire
  - ☐ A obtenir l'heure courante
  - ☐ A générer un réel pseudo-aléatoire

- Que retourne la fonction *time* de la librairie *time* ?
  - ☐ Un entier pseudo-aléatoire
  - ☐ Un entier correspondant au nombre de secondes écoulées depuis le 1er janvier 1970
  - ☐ L'heure actuelle sous forme de texte

- Comment fonctionne la ligne de code 77 ?

- A quoi correspond la constante RAND_MAX définie dans la bibliothèque *stdlib* ?
  - ☐ A la plus grande valeur entière que la fonction *rand* peut retourner.
  - ☐ Au nombre maximal de valeurs que peut retourner la fonction *rand*
  - ☐ Au nombre maximum de fois où la fonction rand peut être appelée

- La fonction *rand* n'a pas de paramètre
  - ☐ Vrai
  - ☐ Faux

- Que retourne la fonction *rand* ?
  - ☐ Un entier pseudo-aléatoire compris entre 0 et RAND_MAX inclus
  - ☐ Un entier pseudo-aléatoire compris entre 1 et RAND_MAX exclu
  - ☐ Un réel pseudo-aléatoire compris entre 0 et 1
  - ☐ Un réel pseudo-aléatoire compris entre 0 et RAND_MAX inclus

- Ecrire une fonction appelée *tirerEntier* qui :
  - ✓ reçoit en paramètres deux entiers a et b tels que a<b
  - ✓ tire puis retourne un entier pseudo-aléatoire compris entre a et b inclus

- Ecrire une fonction appelée *tirerReel* qui génère et retourne un réel compris dans l'intervalle [0,1[(1 exclu) ayant deux chiffres significatifs après la virgule.
  Indice : Utiliser la division ...

IV. Appels de sous-programmes

- A part la fonction principale *main()*qui est appelée automatiquement au début de l'exécution du programme, un sous-programme seul ne peut pas être exécuté. Pour qu'un sous-programme soit exécuté, il faut qu'il soit **appelé** à partir d'un *main()* ou par un autre sous-programme lui-même appelé à partir d'un *main()*.

- Quand on appelle un sous-programme paramétré, il faut lui donner en paramètres des valeurs qui correspondent aux types des paramètres déclarés dans son en-tête.

- L'appel d'un sous-programme est une instruction. Une instruction appartient forcément à un bloc. Un bloc appartient forcément à un sous-programme. Donc, quand on réalise un appel de sous-programme, c'est forcément à partir d'un autre sous-programme. Cela nous amène à distinguer :
  - ✓ Le sous-programme **appelé** : celui dont le nom apparaît dans l'appel, et que l'on veut exécuter.
  - ✓ Le sous-programme **appelant** : celui qui effectue l'appel (qui contient l'instruction d'appel).

  Exemples : Dans le code de l'activité 9 :

n°ligne	instruction d'appel	appelant	appelé
66	test=testerMulti();	entrainerMulti	testerMulti
110	afficherMultiples(n,x);	main	afficherMultiples

V. Les paramètres d'un sous-programme.

Il faut distinguer :

1. **Les paramètres formels** qui sont déclarés dans l'en-tête du sous-programme.
2. **Les paramètres effectifs** qui sont les valeurs envoyées au sous-programme dans l'instruction d'appel.

Les paramètres formels sont des variables locales au sous-programme qui sont initialisées par les paramètres effectifs transmis lors d'un appel :

- ✓ ils ne sont accessibles que dans le bloc du sous-programme, et pas à l'extérieur.
- ✓ leurs espaces mémoires sont réservés au début de l'exécution du sous-programme, et libérés à la fin de son exécution.

- Lors d'un appel, on peut indiquer des expressions dans les parenthèses. Dans ce cas, les expressions sont d'abord évaluées, puis leurs résultats sont envoyés au sous-programme.
- Lors d'un appel, les contraintes suivantes doivent être absolument respectées :
  - ✓ Le nombre de valeurs ou d'expressions doit correspondre au nombre de paramètres formels.
  - ✓ Les types de ces valeurs ou expressions doivent correspondre à ceux des paramètres formels. La 1ère expression passée doit donc avoir le type du 1er paramètre formel déclaré, la 2ème doit avoir le type du 2ème paramètre formel, etc …

VI. Retour de fonction

Si un sous-programme appelle une fonction et veut utiliser le résultat qu'elle retourne, elle doit le stocker dans une variable (par affectation). Sinon le résultat est perdu.

Exemples : Dans le *main()* de l'activité 9 :

```
99          //appel de la fonction menu et récupération de la valeur retournée
100         //qui est stockée dans la variable choix
101         choix=menu();
102         //tests : selon le choix, le programme exécutera des instructions différentes
103         if(choix==1) afficherTableMulti();
```

VII. Organisation du code

Une fonction doit être connue (mais pas forcément complètement définie) du compilateur avant sa première utilisation dans le programme.

1. Cas d'un seul fichier source (petit programme)

Pour ne pas avoir à se préoccuper de l'ordre dans lequel sont codés (définis) les sous-programmes dans un fichier source, on peut les déclarer au début du fichier (au niveau des directives de pré-compilation) en indiquant leurs **prototypes.**
Le prototype d'un sous-programme est son en-tête suivi d'un point-virgule.

2. Programmation multi-fichiers (programme volumineux, programmation en équipe).

Nous verrons plus loin comment répartir les sous-programmes dans plusieurs fichiers.

VIII. Passage par valeurs / Passage par adresses

Activité 10. Comprendre le passage par valeur

Téléchargez le code source « passage_valeurs.c ». Editez le programme et exécutez-le. Que constatez-vous ? Comment expliquez-vous le résultat obtenu ?

- En C, le passage de paramètres se fait toujours **par valeur.**
- On ne transmet pas des variables, mais seulement leurs valeurs.
- Les sous-programmes font **des copies** des valeurs qu'on leur transmet et travaillent sur ces copies.
- Les paramètres formels sont des variables locales initialisés avec les valeurs transmises au sous-programme lors de son appel.

- Rappel (chapitre 2): Les variables locales déclarées dans un sous-programme :
  - N'existent que pendant la durée d'exécution de ce sous-programme
  - Ne sont pas accessibles depuis les autres sous-programmes

Un sous-programme ne peut pas modifier directement les données stockées dans les variables d'un autre sous-programme.

Il peut le faire de manière indirecte si on lui transmet en paramètre l'adresse des données à modifier.
Nous verrons plus loin comment mettre en place le passage par adresse. Il nous permettra également de récupérer plusieurs résultats calculés par un sous-programme.