École de technologie supérieure

THÈME 2
Quelle est votre fonction?

INF125 Introduction à la programmation
Sylvie Ratté et Hugues Saulnier

1. Mes fonctions à moi: comment écrire ses propres fonctions

COMPÉTENCE VISÉE:
Écrire vos propres fonctions, savoir les appeler, être capable de déclarer les prototypes correctement, faire la différence entre une déclaration de fonction (son prototype) et sa définition, manipuler correctement les valeurs de retour.

A. Créer des fonctions et les appeler

Nous le savons maintenant. Un programme C est une collection de fonctions (ce qui inclut la fonction "main"). Certaines fonctions (ex. scanf, printf) sont prédéfinies grâce aux librairies standards du C. On connaît déjà les librairies "stdio" et "stdlib". La plupart des fonctions sont cependant définies par le programmeur c'est-à-dire vous. Voici une nouvelle version de notre programme qui calcule des intérêts composés. Cette fois, nous cumulons les intérêts mensuellement au lieu de les cumuler annuellement.

Création des fonctions

INTRATE7.C

```
int main()
{
```

DÉCLARATION du prototype de la fonction...CONTRAIREMENT à ce qu'illustrent les exemples du livre, nous vous encourageons fortement à placer vos prototypes à l'EXTÉRIEUR des blocs. C'est la façon la plus courante de procéder... pour être compris et se comprendre.

```
  double yearEndBalance(double monthly_bal,
                        double interest_rate);
```

Déclaration des variables

```
  int     period;
  int     year;
  double  balance;
  double  intrate;
```

Traitements préliminaires

```
  printf("Enter interest rate, principal, and period: ");
  scanf("%lf %lf %i", &intrate, &balance, &period);

  printf("Interest Rate:      %7.2f%%\n", intrate * 100);
  printf("Starting Balance: $ %7.2f\n", balance);
  printf("Period:             %7i years\n\n", period);
  printf("Year     Balance\n");
```

```
  for (year = 1; year <= period; year = year + 1)
```

APPEL de la fonction...

```
  {
    balance = yearEndBalance(balance, intrate);
```

et impression.

```
    printf("%4i    $ %7.2f\n", year, balance);
  }
```

```
  return EXIT_SUCCESS;
}
```

DÉFINITION de la fonction

```
/* Compute a year's ending balance, compounding interest monthly */

double yearEndBalance(double monthly_bal,
                      double interest_rate)
{
  double  monthly_intrate;              /* % interest per month */
  int     month;                        /* current month */

  monthly_intrate = interest_rate / 12;
  for (month = 1; month <= 12; month = month + 1)
    monthly_bal = monthly_bal * monthly_intrate + monthly_bal;

  return monthly_bal;
}
```

Pour définir une nouvelle fonction, on fournit:

1. l'en-tête qui spécifie chacun des paramètres (avec leur type associé) et le type de valeur retournée par la fonction;
2. le corps qui contient les déclarations des variables locales nécessaire et le code qui sera exécuté lorsque la fonction sera appelée.

```
type-de-retour NOM-FONCTION(paramètre-1, paramètre-2, ..., paramètre-N)
{
  déclarations locales
  énoncés
}
```

Appel des fonctions

Pour appeler une fonction, il suffit de spécifier son nom et de fournir une liste de valeurs (les paramètres actuels ou arguments). Lors de chaque appel, C assigne la valeur de chaque argument à chaque paramètre formel. Il exécute ensuite le corps de la fonction jusqu'à ce qu'il rencontre l'énoncé "return". Le mécanisme qui effectue le lien entre la liste d'arguments et la liste de paramètres formels est appelé un "passage par valeur" car chaque argument est copié dans chaque paramètre formel. Chaque paramètre formel peut être pensé comme une variable locale agréaglement initialisée automatiquement lorsque la fonction est appelée. Les arguments eux-mêmes ne sont pas modifiés par la fonction. Ce mode de passage est différent de celui utilisé par la fonction "scanf" par exemple.

Lorsque C exécute une fonction, la valeur de retour de cette fonction remplace l'énoncé d'appel.

Fournir un prototype pour chaque fonction

Au tout début du "main", on trouve une déclaration qui s'apparente à une déclaration de variable:

```
double yearEndBalance(double monthly_val, double interest_rate);
```

Il s'agit de la déclaration du PROTOTYPE de la fonction. Un prototype indique clairement les informations dont le compilateur a besoin pour compiler votre code adéquatement: le type de la valeur de retour, le nom de la fonction et le type de chacun des paramètres formels. Il n'est pas strictement obligatoire de mentionner également les noms de ces paramètres mais la présence de ceux-ci aide à la compréhension générale de vos programmes. Ainsi,

```
double yearEndBalance(double, double);
```

est un prototype valide.

DANS CE COURS, NOUS VOUS DEMANDONS DE DÉCLARER VOS PROTOTYPES AVANT LE "MAIN" ET NON À L'INTÉRIEUR DE CE DERNIER.

B. Avec ou sans valeur de retour, elles reviennent toujours

Fonctions sans valeur de retour

Parfois, lors de l'écriture d'une tâche spécifique, on ne parvient pas à trouver une valeur de retour utile c'est le cas, par exemple, des fonctions qui ne font que de l'affichage. Dans ce cas, la valeur de retour sera "void". La fonction "printHeadings" en constitue un bel exemple.

Si une telle fonction doit sortir avant la fin normale, on utilisera l'énoncé "return" comme suit:

```
return;
```

<table>
<tr><td>INTRATE8.C</td><td></td></tr>
<tr><td><pre>void printHeadings(double intrate, double balance, int period);</pre></td><td>Le PROTOTYPE d'une fonction qui ne retourne pas de valeur.</td></tr>
<tr><td><pre>void printHeadings(double intrate, double balance, int period)
{
  printf("Interest Rate: %7.2f%%\n", intrate * 100);
  printf("Starting Balance: $ %7.2f\n", balance);
  printf("Period: %7i years\n\n", period);
  printf("Year   Balance\n");
}</pre></td><td>sa DÉFINITION</td></tr>
<tr><td><pre> printHeadings(intrate, balance, period);</pre></td><td>son UTILISATION (APPEL) avant l'entrée dans la boucle FOR.</td></tr>
</table>

Fonctions sans paramètre

Lorsque votre fonction n'a besoin d'aucun donnée d'entrée pour réaliser sa tâche correctement, vous vous trouvez devant une fonction SANS paramètre formel. Dans ce cas, on place simplement le mot "void" entre parenthèses pour indiquer cette particularité. On peut également placer simplement des parenthèses vides. La fonction "writePrompt" en constitue un bel exemple.

2. On se sépare pour mieux se rassembler: la compilation séparée

COMPÉTENCE VISÉE:

Être capable d'écrire une petite librarie et son fichier en-tête associée, de la compiler et de l'utiliser dans plusieurs programmes.

Le langage C nous permet de redistribuer le code source de nos programmes sur plusieurs fichiers. On peut ainsi compiler séparément chaque fichier pour ensuite laisser l'éditeur de liens effectuer le travail de reconstruction. Voici un bel exemple:

IR_MAIN.C est le fichier principal, celui qui contient notre "main". Ce fichier utilise la librairie IR_LIB.C. Tous les prototypes des fonctions accessibles par tout programmeur-utilisateur de la librairie, sont placés dans le fichier en-tête IR_LIB.H.

IR_MAIN.C

```
#include "ir_lib.h"
```

Inclusion de notre librarie personnelle. Remarquez ici que nous utilisons les guillemets plutôt que les crochets angulaires.

```
int main()
{
  int     period;
  double  balance;
  double  intrate;
  int     n;
```

Nos variables...

```
  writePrompt();
  while ((n = scanf("%lf %lf %i", &intrate, &balance, &period)) == 3)
  {
    printHeadings(intrate, balance, period);
    printBalances(intrate, balance, period);
    writePrompt();
  }
  return notifyIfError(n);
}
```

UTILISATION DES FONCTIONS (APPELS)

IR_LIB.H

```
#include <stdio.h>
#include <stdlib.h>
void writePrompt(void);
void printHeadings(double intrate, double balance, int period);
void printBalances(double intrate, double balance, int period);
int notifyIfError(int values_read);
```

Les PROTOTYPES des fonctions qui seront définies dans la librarie et que tout programme pourra utiliser.

IR_LIB.C

```
static double yearEndBalance(double monthly_bal,
                             double interest_rate);
```

Une fonction préfixée du mot "static" n'est connue qu'à l'intérieur de la librairie. Voici le PROTOTYPE...

```
void printBalances(double intrate, double balance, int period)
{

  int year;

  for (year = 1; year <= period; year = year + 1)
  {
    balance = yearEndBalance(balance, intrate);
    printf("%4i   $ %7.2f\n", year, balance);
  }
}
```

Le prototype de cette fonction apparaît dans le .H... Il s'agit donc d'une fonction que l'on pourra utiliser à L'EXTÉRIEUR de la librairie...

```
static double yearEndBalance(double monthly_bal,
                             double interest_rate)
{
  double  monthly_intrate;             /* % interest per month */
  int     month;                       /* current month */

  monthly_intrate = interest_rate / 12;
  for (month = 1; month <= 12; month = month + 1)
    monthly_bal = monthly_bal * monthly_intrate + monthly_bal;
  return monthly_bal;
}
```

Voici la DÉFINITION de la fonction utilitaire "yearEndBalance" accessible uniquement par la librairie.

```
void writePrompt(void)
  { printf("Enter interest rate, principal, and period: "); }
```

Le prototype de cette fonction apparaît dans le .H... Il s'agit donc d'une fonction que l'on pourra utiliser à L'EXTÉRIEUR de la librairie...

```
void printHeadings(double intrate, double balance, int period)
{
  printf("Interest Rate:      %7.2f%%\n", intrate * 100);
  printf("Starting Balance: $ %7.2f\n", balance);
  printf("Period:             %7i years\n\n", period);
  printf("Year     Balance\n");
}
```

Le prototype de cette fonction apparaît dans le .H... Il s'agit donc d'une fonction que l'on pourra utiliser à L'EXTÉRIEUR de la librairie...

```
int notifyIfError(int values_read)
{
  if (values_read != EOF)
  {
    printf("Warning: Encountered error in reading input.\n");
```

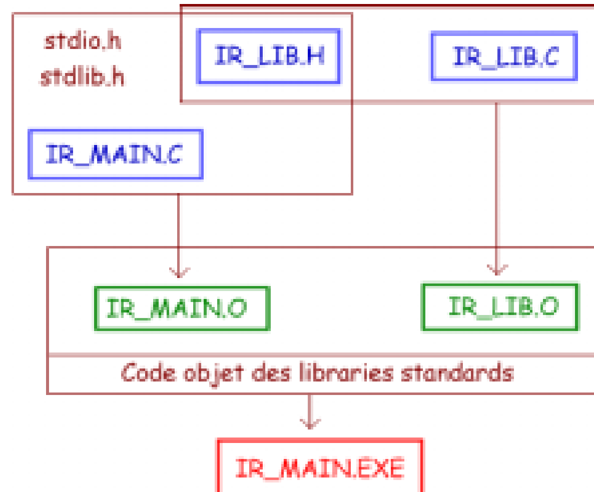
Le prototype de cette fonction apparaît dans le .H... Il s'agit donc d'une fonction que l'on pourra utiliser à L'EXTÉRIEUR de la

`    return EXIT_FAILURE;`
`  }`
`  return EXIT_SUCCESS;`
`}`
	d'une fonction que l'on pourra utiliser à L'EXTÉRIEUR de la librairie...

Réutilisation

Cette méthode s'avère extrêmement utile pour développer de larges programmes ou encore pour créer des fichiers indépendants que vous pouvez réutiliser dans d'autres projets.

La compilation se déroule comme suit:



On notera que le fichier "ir_lib.o" peut dès lors être réutilisé par un autre programme sans avoir à être recompilé.

En GCC, on fera les trois commandes suivantes:

```
gcc -c ir_lib.c
gcc -c ir_main.c
gcc -o ir_main.exe ir_lib.o ir_main.o
```

ou encore, on pourra procéder en effectuant une seule commande:

```
gcc -o ir_main.exe ir_lib.c ir_main.c
```

En Turbo C, on doit d'abord créer un projet (.prj) contenant le fichier ir_main.c et ir_lib.c puis, on lance la compilation. Le processus de compilation produira ir_main.obj, ir_main.exe et ir_lib.obj. (l'extension "obj" signifiant la même chose que l'extension "o" en gcc).

ÉTUDES DE CAS

A. Les nombres premiers

Comment déterminer si un nombre est premier? Facile. Il suffit de vérifier si un nombre entier le divise entièrement. Autrement dit, si le reste de la division par un autre nombre égale zéro, alors notre nombre n'est pas un nombre premier.

En C, il est facile d'obtenir le reste de la division entière entre deux nombres grâce à l'opérateur modulo noté %. Ainsi,

5 % 2 = 1 puisque le reste de la division entière par 2 donne 1.

19 % 5 = 4 puisque le reste de la division entière par 5 donne 4.

Pour vérifier qu'un nombre est premier, on devra donc tester plusieurs diviseurs potentiels. Une première approche serait de coder l'algorithme suivant:

ALGORITHME No 1

Soit N le nombre à tester,

Pour tous les diviseurs allant de 2 à N-1, tester si N % diviseur égale zéro, si c'est le cas, le nombre n'est pas premier et l'on doit alors cesser la recherche. Si, par contre, on ne réussit pas à trouver un tel cas, alors le nombre est premier.

Au lieu de coder cet algorithme directement dans la fonction "main", il serait beaucoup plus pratique de coder une fonction, disons "estPremier". Cette fonction nous permettrait de tester un nombre que l'on fournirait en argument. En guise de valeur de retour, elle pourrait nous retourner 1, si le nombre fourni est premier, et 0 autrement. Il est très courant (et vous comprendrez très vite pourquoi) de

choisir ces valeurs lorsque l'on désire que la fonction retourne un résultat qui, de fait, peut s'interpréter comme OUI ou NON, ou encore, comme VRAI ou FAUX. Le prototype de notre fonction est le suivant:

```
int estPremier(int N);
```

La définition de la fonction serait:

<table>
<tr>
<td>

```
int estPremier(int N) {
   int diviseur;
   for (diviseur = 2; diviseur < N; diviseur = diviseur + 1){
      if (N % diviseur == 0) return 0;
   }
   return 1;
}
```

</td>
<td>L'énoncé "return" a ceci de particulier: dès qu'il est exécuté, on sort immédiatement de la fonction avec le résultat. Le reste de l'itération est oublié, de même pour les énoncés qui pourraient suivre. C'est en quelque sorte une sortie d'urgence.</td>
</tr>
</table>

Est-ce que cela fonctionne? Et bien OUI mais lentement...

ALGORITHME No 2

On notera qu'il est ridicule de tester tous les nombres pairs puisque si 2 ne divise pas le nombre en question, il est certain que 4, 6, 8 etc. ne le diviseront pas plus! Une première amélioration consisterait donc à tester le 2 à part puis, à tester tous les nombres impairs à partir de 3. On obtient:

```
int estPremier(int N) {
   int diviseur;
   if (N % 2 == 0) return 0;
   for (diviseur = 3; diviseur < N; diviseur = diviseur + 2){
      if (N % diviseur == 0) return 0;
   }
   return 1;
}
```

ALGORITHME No 3

on notera que le problème posé par le 2, se pose à nouveau pour le 3, le 5, etc. En effet, si le nombre ne se divise pas par 3 ou par 5, inutile de tester 9, 15, 21, 25, etc. En fait, pour obtenir un algorithme un peu plus efficace, on devrait tester uniquement des diviseurs qui sont eux-mêmes des nombres premiers: 2, 3, 5, 7, 11, 13, etc.

Cette façon de procéder porte un nom: le crible d'Érathostène (du nom du philosophe grec qui a découvert cet algorithme). Malheureusement, la programmation de ce crible en C exige des représentations que nous n'avons pas encore vues. cependant, crible ou pas, il est tout de même possible d'accélérer l'algorithme No 2.

En effet, considérez le nombre 100. OK nous savons qu'il n'est pas premier mais examinons tout de même les diviseurs potentiels qui seraient détectés par l'algorithme No 2 si on lui permettait de poursuivre sa quête de diviseur:

On trouve 2, puisque 2 X 50 = 100
On trouve 4, puisque 4 X 25 = 100
On trouve 5, puisque 5 X 20 = 100
On trouve 10, puisque 10 X 10 = 100
On trouve 20, puisque 20 X 5 = 100
On trouve 25, puisque 25 X 4 = 100
On trouve 50, puisque 50 X 2 = 100

Comme vous pouvez le constater, il ne sert à rien de rechercher des diviseurs lorsqu'on dépasse RACiNE(100), soit 10.

Notre fonction devient:

```
int estPremier(int N) {
   int diviseur;
   if (N % 2 == 0) return 0;
   for (diviseur = 3; diviseur < sqrt(N)+1; diviseur = diviseur + 2){
      if (N % diviseur == 0) return 0;
   }
   return 1;
}
```

Est-ce que cela fonctionne? Et bien NON! Car si je fournis 2, la fonction retourne 0 (= n'est pas premier) puisque 2 % 2 donne un reste de zéro. Le nombre 2 est donc un cas particulier.

```
int estPremier(int N) {
   int diviseur;
   if (N < 3) return 1;
   if (N % 2 == 0) return 0;
   for (diviseur = 3; diviseur < sqrt(N)+1; diviseur = diviseur + 2){
      if (N % diviseur == 0) return 0;
   }
   return 1;
}
```

Évidemment, avec zéro, la fonction retourne 1... à vous de traiter aussi ce cas particulier si vous en avez envie. Est-ce que la fonction donne le bon résultat avec 3? À vous de répondre.

Une version plus robuste consisterait à ne pas tenter d'effectuer des opérations % sur des nombres entiers négatifs. En effet, les résultats dépendent des plate-formes. Il est donc d'usage de toujours traiter les nombres positifs. Pour vérifier si un nombre négatif est premier, on peut d'abord le transformer en positif. On obtient donc:

```
int estPremier(int N) {
   int diviseur;
   if (N < 0) N = -1 * N;
   if (N < 3) return 1;
   if (N % 2 == 0) return 0;
   for (diviseur = 3; diviseur < sqrt(N)+1; diviseur = diviseur + 2){
      if (N % diviseur == 0) return 0;
   }
   return 1;
}
```

B. Lire des entiers pour tous les goûts

Voici trois petites fonctions qui pourraient être bien utiles.

`int lireValEntre_0_et_100(void);`	Cette fonction effectue des lectures répétées tant et aussi longtemps que l'utilisateur n'a pas fourni un nombre entre 0 et 100.
`int lireValEntre_0_et(int borne);`	Cette fonction effectue des lectures répétées tant et aussi longtemps que l'utilisateur n'a pas fourni un nombre entre 0 et le nombre fourni en argument.
`int lireValEntre(int borneInf, int borneSup);`	Cette fonction effectue des lectures répétées tant et aussi longtemps que l'utilisateur n'a pas fourni un nombre dans l'intervalle défini par les deux arguments fournis lors de l'appel.

Les définitions de ces fonctions sont très simples et font appels à des concepts que nous connaissons bien désormais. Notons que ces trois fonctions utiliserons deux constantes, VRAI et FAUX, afin de prendre en note la validité de la valeur lue. De plus, chacune utilise la fonction "fflush" permettant de vider le contenu du tampon d'entrée si jamais ce tampon contenait une valeur non admissible (une séquence de caractères par exemple). Le "fflush", placé ainsi dans la boucle, force également l'utilisateur à ne saisir qu'une seule valeur à la fois. Les valeurs supplémentaires seront "flushées" du tampon d'entrée.

<table>
<tr><td>

```
int lireValEntre_0_et_100(void){
  int valeur;
  int fini;
  fini = FAUX;
  while (fini == FAUX) {
    printf("\nDonnez-moi un entier entre 0 et 100 : ");
    if (scanf("%i", &valeur) == 1) {
      if ((valeur >= 0) && (valeur <= 100))
        fini = VRAI;
    }
    fflush(stdin);
  }
  return valeur;
}
```

</td><td>La variable "fini" sert à indiquer si oui ou non la valeur est acceptable. Sauriez-vous expliquer pourquoi nous n'avons pas utilisé le résultat du "scanf" comme test de sortie de boucle comme nous l'avons fait dans les programmes que nous avons vus jusqu'à présent?

L'opérateur logique "&&" permet d'effectuer un ET logique entre deux conditions. L'expression entière est vraie uniquement lorsque les deux conditions sont vraies.</td></tr>
<tr><td>CODONS MAINTENANT LA FONCTION LA PLUS GÉNÉRALE:</td><td></td></tr>
<tr><td>

```
int lireValEntre(int borneInf, int borneSup){
  int valeur;
  int fini;
  int temporaire;
  fini = FAUX;
  if (borneInf > borneSup) {
    temporaire = borneInf;
    borneInf = borneSup;
    borneSup = temporaire;
  }
  while (fini == FAUX) {
    printf("\nDonnez-moi un entier entre %i et %i : ", borneInf, borneSup);
    if (scanf("%i", &valeur) == 1) {
      if ((valeur >= borneInf) && (valeur <= borneSup))
        fini = VRAI;
    }
    fflush(stdin);
  }
  return valeur;
}
```

</td><td>On notera que cette fonction est très courtoise puisque si jamais l'utilisateur-programmeur fourni une borne inférieure plus grande que la borne supérieure, elle inverse d'abord les bornes avant de procéder. Ce qui rend le code plus simple.</td></tr>
<tr><td>

```
int lireValEntre_0_et(int borneSup){
    if (borneSup > 0)
       return lireValEntre(0, borneSup);
    else
       return lireValEntre(borneSup, 0);
}
```

</td><td>Nous aurions pu coder cette fonction comme les autres mais comme nous sommes un tantinet paresseux, pourquoi ne pas utiliser la fonction générale. C'est le résultat qui compte.

SUGGESTION: Recodez la première fonction afin qu'elle utilise aussi la fonction générale.

CONCLUSION: Une fonction bien pensée peut être utilisée à toutes les sauces.</td></tr>
</table>

Pour tester le bon fonctionnement de ces fonctions, insérons-les dans un petit programme.

<table>
<tr><td>

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

</td><td>Notez que le programme a besoin de ces deux librairies standards. Notez également que les fonctions elles-mêmes ont besoin de "stdio".</td></tr>
<tr><td>

```
#define FAUX 0
#define VRAI 1

int lireValEntre_0_et_100(void);
int lireValEntre_0_et(int borneSup);
int lireValEntre(int borneInf, int borneSup);
```

</td><td>DÉFINITION de constante et PROTOTYPES des fonctions. Cette section deviendra le .H de la future librairie.</td></tr>
<tr><td>

```
int main() {
  printf("\n VOUS AVEZ TAPÉ %i", lireValEntre_0_et_100());
  printf("\n VOUS AVEZ TAPÉ %i", lireValEntre_0_et(2000));
  printf("\n VOUS AVEZ TAPÉ %i", lireValEntre_0_et(-3000));
  printf("\n VOUS AVEZ TAPÉ %i", lireValEntre(2000, 1000));
  printf("\n VOUS AVEZ TAPÉ %i", lireValEntre(-10, 10));
  return EXIT_SUCCESS;
}
```

</td><td>Le "main"... rien à dire.</td></tr>
<tr><td>DÉFINITIONS DES FONCTIONS</td><td>Les définitions de nos fonctions telles que présentées plus haut. Cette section deviendra le .C de la future librairie.</td></tr>
</table>

Tout fonctionne pour le mieux. Et puisque tout va si bien, ne nous arrêtons pas. Elles sont bien utiles. Offrons-les en librairie à nos amis. Nous appellerons notre librairie "lecture". Voici le contenu des deux fichiers nécessaires pour construire cette librairie.

LECTURE.H, LECTURE.C

Désormais, tout programmeur désirant utiliser notre librairie (exemple: TESTLIR2.C) n'aura qu'à faire le "include" approprié, soit:

```
#include "LECTURE.H"
```

Vous noterez que lorsqu'il s'agit d'une librairie personnelle, on remplace les crochets angulaires par les guillemets. C'est tout... ou presque!

Pour le mode d'emploi de l'utilisation des librairies personnelles en Turbo C et en GCC, consultez ce lien.