Introduction au Langage C

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://claroline connect.univ-lyon1.fr` espace d'activités "M1102 M1103 C - Introduction au langage C"

46H prévues $\approx$ 42H de cours+TPs, 2H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

Plan

Quelques définitions

Une fonction

Un sous-programme *paramétrable* qui effectue un traitement et renvoie une valeur.

Une procédure

Un sous-programme *paramétrable* qui effectue un traitement et ne renvoie pas de valeur.

Une seule notion en C, celle de fonction C

Un sous-programme paramétrable qui effectue un traitement et renvoie une valeur qui peut être vide.

Quelques définitions

Une fonction

Un sous-programme *paramétrable* qui effectue un traitement et renvoie une valeur.

Une procédure

Un sous-programme *paramétrable* qui effectue un traitement et ne renvoie pas de valeur.

Une seule notion en C, celle de fonction C

Un sous-programme paramétrable qui effectue un traitement et renvoie une valeur qui peut être vide.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable.**
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable.**
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable.**
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable**.
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable**.
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable**.
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Intérêts des fonctions C

- Une fonction est **une unité de traitement paramétrable**.
- **Proche de la notion de fonctions mathématiques** si elle fournit un résultat scalaire (un type de base) :
  - Des arguments.
  - Elle peut être utilisée au sein d'une expression.
- **Découper un problème en sous-problèmes** :
  - Lecture du programme plus facile.
  - Décomposition d'1 traitement complexe en plusieurs sous-programmes : cela facilite la compréhension et le *débogage*.
  - Réutilisation du code : éviter des erreurs de non-propagation.
  - Génie logiciel : plusieurs fichiers sources (*modules*) :
    - Des fonctions regroupées par fonctionnalité ou type des données traitées.
    - Compilation séparée des fichiers sources.

Spécificités d'une fonction C

- Sa valeur de retour
  - peut ne pas être utilisée (e.g. printf).
  - peut ne pas être scalaire.
- Elle peut réaliser des actions.
- Elle peut modifier la valeur de certains de ces arguments par un mécanisque particulier (cf. cours sur les pointeurs).
- Chaque fonction peut accéder à toutes les variables globales connues avant sa définition.

Spécificités d'une fonction C

- Sa valeur de retour
  - peut ne pas être utilisée (e.g. printf).
  - peut ne pas être scalaire.
- Elle peut réaliser des actions.
- Elle peut modifier la valeur de certains de ces arguments par un mécanisque particulier (cf. cours sur les pointeurs).
- Chaque fonction peut accéder à toutes les variables globales connues avant sa définition.

Spécificités d'une fonction C

- Sa valeur de retour
  - peut ne pas être utilisée (e.g. printf).
  - peut ne pas être scalaire.
- Elle peut réaliser des actions.
- Elle peut modifier la valeur de certains de ces arguments par un mécanisque particulier (cf. cours sur les pointeurs).
- Chaque fonction peut accéder à toutes les variables globales connues avant sa définition.

Spécificités d'une fonction C

- Sa valeur de retour
  - peut ne pas être utilisée (e.g. printf).
  - peut ne pas être scalaire.
- Elle peut réaliser des actions.
- Elle peut modifier la valeur de certains de ces arguments par un mécanisque particulier (cf. cours sur les pointeurs).
- Chaque fonction peut accéder à toutes les variables globales connues avant sa définition.

Déclaration d'une fonction C (prototype)

- **Syntaxe sans argument norme ANSI** :
  <*type*> <*identificateur*> (**void**) **;**
- **Syntaxe avec type(s) argument(s)** :
  <*type*> <*identificateur*> (<type_argument_1>, ..., <type_argument_n>) **;**
- **Syntaxe avec argument(s)** :
  <*type*> <*identificateur*> (<déclaration_argument_1>, ..., <déclaration_argument_n>) **;**

type est le type de retour de la fonction, il peut être soit un type de base, soit un type utilisateur, soit **void** si rien n'est retourné.
identificateur est l'identificateur de la fonction.

Déclaration d'une fonction C (prototype)

- **Syntaxe sans argument norme ANSI** :
  <*type*> <*identificateur*> (**void**) **;**
- **Syntaxe avec type(s) argument(s)** :
  <*type*> <*identificateur*> (<type_argument_1>, ..., <type_argument_n>) **;**
- **Syntaxe avec argument(s)** :
  <*type*> <*identificateur*> (<déclaration_argument_1>, ..., <déclaration_argument_n>) ;

type est le type de retour de la fonction, il peut être soit un type de base, soit un type utilisateur, soit **void** si rien n'est retourné.
identificateur est l'identificateur de la fonction.

Déclaration d'une fonction C (prototype)

- **Syntaxe sans argument norme ANSI** :
  <*type*> <*identificateur*> (**void**) **;**
- **Syntaxe avec type(s) argument(s)** :
  <*type*> <*identificateur*> (<type_argument_1>, ..., <type_argument_n>) **;**
- **Syntaxe avec argument(s)** :
  <*type*> <*identificateur*> (<déclaration_argument_1>, ..., <déclaration_argument_n>) **;**

type est le type de retour de la fonction, il peut être soit un type de base, soit un type utilisateur, soit **void** si rien n'est retourné. *identificateur* est l'identificateur de la fonction.

Définition d'une fonction C (1/2)

- **Syntaxe sans argument norme ANSI** :
  <*type*> <*identificateur*> (**void**) instruction_bloc
- **Syntaxe avec argument(s)** :
  <*type*> <*identificateur*> (<déclaration_argument_1>, ..., <déclaration_argument_n>) instruction_bloc

type est le type de retour de la fonction, il peut être soit un type de base, soit un type utilisateur, soit **void** si rien n'est retourné.
identificateur est l'identificateur de la fonction.

Définition d'une fonction C (1/2)

- **Syntaxe sans argument norme ANSI** :
  <*type*> <*identificateur*> (**void**) instruction_bloc
- **Syntaxe avec argument(s)** :
  <*type*> <*identificateur*> (<déclaration_argument_1>, ..., <déclaration_argument_n>) instruction_bloc

type est le type de retour de la fonction, il peut être soit un type de base, soit un type utilisateur, soit **void** si rien n'est retourné. *identificateur* est l'identificateur de la fonction.

Définition d'une fonction C (2/2)

Si le type de retour est différent de **void**, dans instruction_bloc il doit y avoir obligatoirement un **return** < *expression* >**;** pour chaque déroulement possible de la fonction.
Sinon il peut y avoir un **return;**

Si *expression* est d'un type différent du type de retour de la fonction, le compilateur mettra en place des instructions de conversion.

L'instruction **return** interrompt l'exécution de la fonction courante en revenant dans la fonction appellante.

Définition d'une fonction C (2/2)

Si le type de retour est différent de **void**, dans instruction_bloc il doit y avoir obligatoirement un **return** < *expression* >**;** pour chaque déroulement possible de la fonction.
Sinon il peut y avoir un **return;**

Si *expression* est d'un type différent du type de retour de la fonction, le compilateur mettra en place des instructions de conversion.

L'instruction **return** interrompt l'exécution de la fonction courante en revenant dans la fonction appellante.

Définition d'une fonction C (2/2)

Si le type de retour est différent de **void**, dans instruction_bloc il doit y avoir obligatoirement un **return** < *expression* >**;** pour chaque déroulement possible de la fonction.
Sinon il peut y avoir un **return;**

Si *expression* est d'un type différent du type de retour de la fonction, le compilateur mettra en place des instructions de conversion.

L'instruction **return** interrompt l'exécution de la fonction courante en revenant dans la fonction appellante.

Comment choisir le type de retour d'une fonction ?

- **void** pour les procédures, les fonctions ne faisant que des traîtements et des affichages.
- **int** pour le cas ci-dessus, pour lequel on souhaite cette fois retourner un entier indiquant le bon déroulement (0) ou un code d'erreur (≠0).
- **int** pour le cas où la fonction doit retourner un booléen (faux=0 et vrai=1).
- **int**, **double** etc. pour une fonction mathématique (e.g. factorielle) ; le type de retour exact dépend du problème.

Remarque : il ne faut pas afficher dans une fonction la valeur qu'elle retourne (redondant).

Comment choisir le type de retour d'une fonction ?

- **void** pour les procédures, les fonctions ne faisant que des traîtements et des affichages.
- **int** pour le cas ci-dessus, pour lequel on souhaite cette fois retourner un entier indiquant le bon déroulement (0) ou un code d'erreur (≠0).
- **int** pour le cas où la fonction doit retourner un booléen (faux=0 et vrai=1).
- **int**, **double** etc. pour une fonction mathématique (e.g. factorielle) ; le type de retour exact dépend du problème.

Remarque : il ne faut pas afficher dans une fonction la valeur qu'elle retourne (redondant).

Comment choisir le type de retour d'une fonction ?

- **void** pour les procédures, les fonctions ne faisant que des traîtements et des affichages.
- **int** pour le cas ci-dessus, pour lequel on souhaite cette fois retourner un entier indiquant le bon déroulement (0) ou un code d'erreur (≠0).
- **int** pour le cas où la fonction doit retourner un booléen (faux=0 et vrai=1).
- **int**, **double** etc. pour une fonction mathématique (e.g. factorielle) ; le type de retour exact dépend du problème.

Remarque : il ne faut pas afficher dans une fonction la valeur qu'elle retourne (redondant).

Comment choisir le type de retour d'une fonction ?

- **void** pour les procédures, les fonctions ne faisant que des traîtements et des affichages.
- **int** pour le cas ci-dessus, pour lequel on souhaite cette fois retourner un entier indiquant le bon déroulement (0) ou un code d'erreur (≠0).
- **int** pour le cas où la fonction doit retourner un booléen (faux=0 et vrai=1).
- **int**, **double** etc. pour une fonction mathématique (e.g. factorielle) ; le type de retour exact dépend du problème.

Remarque : il ne faut pas afficher dans une fonction la valeur qu'elle retourne (redondant).

Comment choisir le type de retour d'une fonction ?

- **void** pour les procédures, les fonctions ne faisant que des traîtements et des affichages.
- **int** pour le cas ci-dessus, pour lequel on souhaite cette fois retourner un entier indiquant le bon déroulement (0) ou un code d'erreur (≠0).
- **int** pour le cas où la fonction doit retourner un booléen (faux=0 et vrai=1).
- **int**, **double** etc. pour une fonction mathématique (e.g. factorielle) ; le type de retour exact dépend du problème.

Remarque : il ne faut pas afficher dans une fonction la valeur qu'elle retourne (redondant).

Appel d'une fonction C

- **Syntaxe sans argument** : <*identificateur*> ( )
- **Syntaxe avec argument(s)** : <*identificateur*> (<argument_1>, ..., <argument_n>)

Les types des arguments effectifs doivent être compatibles avec ceux des paramètres formels. A chaque appel de fonction, le type de chaque argument est converti au type du paramètre formel correspondant si nécessaire.

Un appel de fonction peut avoir lieu au sein d'une autre fonction. En particulier une fonction peut s'appeler elle-même. On parle alors de **fonction récursive**.

Appel d'une fonction C

- **Syntaxe sans argument** : <*identificateur*> ( )
- **Syntaxe avec argument(s)** : <*identificateur*> (<argument_1>, ..., <argument_n>)

Les types des arguments effectifs doivent être compatibles avec ceux des paramètres formels. A chaque appel de fonction, le type de chaque argument est converti au type du paramètre formel correspondant si nécessaire.

Un appel de fonction peut avoir lieu au sein d'une autre fonction. En particulier une fonction peut s'appeler elle-même. On parle alors de **fonction récursive**.

Appel d'une fonction C

- **Syntaxe sans argument** : <*identificateur*> ( )
- **Syntaxe avec argument(s)** : <*identificateur*> (<argument_1>, ..., <argument_n>)

Les types des arguments effectifs doivent être compatibles avec ceux des paramètres formels. A chaque appel de fonction, le type de chaque argument est converti au type du paramètre formel correspondant si nécessaire.

Un appel de fonction peut avoir lieu au sein d'une autre fonction. En particulier une fonction peut s'appeler elle-même. On parle alors de **fonction récursive**.

Exemple de fonctions C (1/2)

fonction.c

```c
/* Déclaration et définition globales de fonction */
#include <stdio.h>
double abso(double x) /* définition sans déclaration de abso: marche mais à éviter */
{
    if(x>=0.)
        return x;
    else
        return -x;
}
double absoSomme(double, double); /* prototype de absSomme : déclaration "globale" */
int main(void) /* programme principal */
{
    double z=-4.2, h=3.;
    printf("La valeur absolue de %f est %f et celle de %f+%f est %f\n", z, abso(z), z, h,
          absoSomme(z,h) /* z et h sont des arguments effectifs */
        );
    return 0;
}
/* x et y sont des arguments muets ou paramètres formels */
double absoSomme(double x, double y) /* implantation de absSomme */
{
    if(x+y>=0.)
        return x+y;
    else
        return -(x+y);
}
```

Exemple de fonctions C (2/2)

fonction2.c

```c
#include <stdio.h>
double tirage(void)
{
   return -1.0;
}
void message2(void)
{
   printf("message2\n");
}
int main(void) /* programme principal */
{
    void message1(void) ; /* prototype de message1 : déclaration "locale" */
    tirage();
    message1();
    message2();
    return 0;
}
void message1(void)
{
   printf("message1\n");
   message2();
   return;
}
```

Intérêts des déclarations de fonctions C ? (1/3)

Avant tout une recommandation pour les développeurs en C.

- **Contrôle** : le compilateur s'assurera que les arguments de la définition sont du même type que les arguments du prototype et que le nombre d'argument(s) est le même.

Attention : si le compilateur ne connaît pas le type des arguments, il y a conversion systématique : **char** et **short** en **int** et **float** en **double**. Si le compilateur ne connaît pas le type de retour, il choisira le type **int**.

Intérêts des déclarations de fonctions C ? (1/3)

Avant tout une recommandation pour les développeurs en C.

- **Contrôle** : le compilateur s'assurera que les arguments de la définition sont du même type que les arguments du prototype et que le nombre d'argument(s) est le même.

Attention : si le compilateur ne connaît pas le type des arguments, il y a conversion systématique : **char** et **short** en **int** et **float** en **double**. Si le compilateur ne connaît pas le type de retour, il choisira le type **int**.

Intérêts des déclarations de fonctions C ? (1/3)

Avant tout une recommandation pour les développeurs en C.

- **Contrôle** : le compilateur s'assurera que les arguments de la définition sont du même type que les arguments du prototype et que le nombre d'argument(s) est le même.

  Attention : si le compilateur ne connaît pas le type des arguments, il y a conversion systématique : **char** et **short** en **int** et **float** en **double**. Si le compilateur ne connaît pas le type de retour, il choisira le type **int**.

Intérêts des déclarations de fonctions C ? (2/3)

```c
#include <stdio.h> /* pour printf et scanf */

double mafonctiontest(); /* omission de la liste des paramètres formels: nombre variable
                            de paramètres */

double mafonctiontest2(void); /* pas d'argument */

int main(void) {
    ...
   return 0;
}
double mafonctiontest(int a, int b) /* utiliser un type non promu (i.e. int ou double) ou ne pas
                                       spécifier le type sinon erreur de compilation */
{
       return a+b;
}
double mafonctiontest2() /* erreur de compilation si un argument */
{
       return 0.0;
}
```

Intérêts des déclarations de fonctions C ? (3/3)

- **Conversions** : lors d'un appel de fonction, le compilateur va mettre en place les éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype.
- **Contrôle et conversion pour les fonctions fournies** : `#include <fichier.h>` incorpore les prototypes déclarés dans fichier.h au fichier source dans lequel la directive apparaît.

```
#include <stdio.h> /* pour printf et scanf */
#include <math.h>  /* pour sqrt, pow */
int main(void) {
    ...
   return 0;
}
```

Intérêts des déclarations de fonctions C ? (3/3)

- **Conversions** : lors d'un appel de fonction, le compilateur va mettre en place les éventuelles conversions des valeurs des arguments effectifs dans le type indiqué dans le prototype.
- **Contrôle et conversion pour les fonctions fournies** : **#include** <fichier.h> incorpore les prototypes déclarés dans fichier.h au fichier source dans lequel la directive apparaît.

```
#include <stdio.h> /* pour printf et scanf */
#include <math.h>  /* pour sqrt, pow */
int main(void) {
    ...
   return 0;
}
```

Si on omet le type de retour...

```
f(int); /* le compilateur attribura le type de retour int */
main(void) /* le compilateur attribura le type de retour int */
{
   f(6) ;
   return 0;
}
f(int a) /* ici soit mettre int comme type de retour soit ne rien mettre */
{
   return 2*a-1;
}
```

Cas d'une définition sans déclaration préalable (A éviter !)...

```c
int main(void) /* le compilateur attribura le type de retour int */
{
   f(6) ;
   return 0;
}
int f(a) /* ici soit mettre int ou double pour tous les types argument soit ne rien mettre */
{        /* pour le type de retour mettre int soit ne rien mettre */
   return 2*a-1;
}
```

Le C n'autorise pas la surcharge

Surcharge

Possibilité offerte par certains langages de programmation (e.g. C++ et le Java) qui permet d'utiliser des fonctions avec le même identificateur, mais pas les mêmes arguments formels (le type de retour n'intervient pas).

Passage des paramètres d'une fonction C (1/3)

```c
/* Comment les paramètres effectifs sont-ils passés à une fonction? */
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
   void echange (int a, int b) ; /* rmq : on peut aussi écrire directement la définition
                                    en "local" */
   int n=10, p=20 ;
   printf("avant appel : %d %d\n", n, p);
   echange(n, p);
   printf("après appel : %d %d\n", n, p);
   system("PAUSE");
   return 0;
}
void echange (int a, int b)
{
  int tmp ;
  printf("début echange : %d %d\n", a, b);
  tmp = a;
  a = b;
  b = tmp ;
  printf("fin echange : %d %d\n", a, b);
}
```

Passage des paramètres d'une fonction C (2/3)

- **Le passage des paramètres se fait par valeur**, on copie la valeur dans une variable locale à l'appel de la fonction.
- On ne peut donc pas modifier les arguments effectifs à l'intérieur du bloc de la fonction, *sauf pour les tableaux*.

Quel est l'avantage principal d'un passage des paramètres par valeur ?

Passage des paramètres d'une fonction C (2/3)

- **Le passage des paramètres se fait par valeur**, on copie la valeur dans une variable locale à l'appel de la fonction.
- On ne peut donc pas modifier les arguments effectifs à l'intérieur du bloc de la fonction, *sauf pour les tableaux*.

Quel est l'avantage principal d'un passage des paramètres par valeur ?

Passage des paramètres d'une fonction C (2/3)

- **Le passage des paramètres se fait par valeur**, on copie la valeur dans une variable locale à l'appel de la fonction.
- On ne peut donc pas modifier les arguments effectifs à l'intérieur du bloc de la fonction, *sauf pour les tableaux*.

Quel est l'avantage principal d'un passage des paramètres par valeur ?
La **possibilité d'utiliser des expressions (complexes) comme argument effectif** à l'appel d'une fonction.

Attention : l'ordre d'évaluation des arguments effectifs n'est pas défini (donc choix du compilo) ! $f(i++, i)$ peut être équivalent soit à $f(i, i+1)$, soit à $f(i, i)$.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Passage des paramètres d'une fonction C (3/3)

Solutions à notre problème d'échange ?

- Utiliser des tableaux d'une seule case. Attendre le cours sur les tableaux.
- Passer l'adresse des variables au moment de l'appel. Attendre le cours sur les pointeurs.
- Déclarer les variables n et p de manière globale (avant la définition de la fonction).
  Attention aux *effets de bords* : le comportement du programme devient plus difficile à comprendre.

Sans les possibilités offertes par le **passage par adresse**, les paramètres d'une fonction sont de type donnée/entrée.

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Communications entre plusieurs fonctions

Comment échanger des informations entre différentes fonctions d'un programme ?

- En utilisant la valeur de retour d'une fonction.
- En utilisant des variables globales (déconseillé).
- En utilisant le passage par adresse.

Sans les notions de *variable globale*, de *pointeur*, de *tableau* et de *structure C*, une fonction ne peut communiquer "directement" à l'extérieur qu'une seule information, sa valeur de retour.

Il est également possible de communiquer "indirectement" à l'aide de fichiers (textes ou binaires).

Portée des variables d'une fonction C

Les variables locales (intermédiaires) à une fonction ainsi que ses paramètres ne sont visibles que dans cette fonction.

Variables globales

- **Durée de vie** : elles existent pendant toute l'exécution du programme dans lequel elles apparaissent.
- **Portée** : elles ne sont "visibles" que dans la partie du programme source suivant leur déclaration.
- **Classe d'allocation** : la classe d'*allocation statique*, car leurs emplacements mémoires sont parfaitement définis lors de l'édition de lien.
  Elles sont initialisées à zéro, sauf si on leur spécifie explicitement une valeur d'initialisation.

Variables locales ou intermédiaires (1/2)

- **Durée de vie** : elles existent pendant la durée de l'exécution de la fonction dans laquelle elles apparaissent.
- **Portée** : elles ne sont "visibles" qu'à l'intérieur du bloc de la fonction et dans la partie suivant leur déclaration.
- **Classe d'allocation** : la classe d'*allocation automatique*, car leurs emplacements mémoires sont réservés à l'appel de la fonction et libérés à sa sortie.
  Elles ne sont pas initialisées à une valeur par défaut, sauf si on leur spécifie explicitement une valeur d'initialisation.

Les valeurs des variables locales ne sont pas conservées d'un appel au suivant. Les paramètres d'une fonction font partie de la classe d'allocation automatique.

Variables locales ou intermédiaires (1/2)

- **Durée de vie** : elles existent pendant la durée de l'exécution de la fonction dans laquelle elles apparaissent.
- **Portée** : elles ne sont "visibles" qu'à l'intérieur du bloc de la fonction et dans la partie suivant leur déclaration.
- **Classe d'allocation** : la classe d'*allocation automatique*, car leurs emplacements mémoires sont réservés à l'appel de la fonction et libérés à sa sortie.
  Elles ne sont pas initialisées à une valeur par défaut, sauf si on leur spécifie explicitement une valeur d'initialisation.

Les valeurs des variables locales ne sont pas conservées d'un appel au suivant. Les paramètres d'une fonction font partie de la classe d'allocation automatique.

Variables locales ou intermédiaires (2/2)

Cas de 2 variables avec le même identificateur :

- *Si dans 2 fonctions différentes, il y a une variable locale avec le même identificateur ?* Aucune conséquence, ces 2 variables n'ont aucun rapport.
- *S'il y a une variable globale qui coexiste avec une variable locale de même nom ?* La variable locale est privilégiée, il n'est alors pas possible d'accéder ou de modifier la variable globale.

Variables locales ou intermédiaires (2/2)

Cas de 2 variables avec le même identificateur :

- *Si dans 2 fonctions différentes, il y a une variable locale avec le même identificateur ?* Aucune conséquence, ces 2 variables n'ont aucun rapport.
- *S'il y a une variable globale qui coexiste avec une variable locale de même nom ?* La variable locale est privilégiée, il n'est alors pas possible d'accéder ou de modifier la variable globale.

Exemple

```c
#include <stdio.h>
int n ; /* variable globale init à 0 */
int main(void)
{
   int p, m ; /* variables locales init à ?? */
   ...
   return 0;
}
int m=2 ; /* variable globale init à 2 */
void fct (void)
{
   int p=1, n=1 ; /* variables locales init à 1 */
   ...
}
```

Q'affiche le programme suivant ?

```c
#include <stdio.h>
int n=5 ; /* variable globale init à 5 */
int main(void)
{
   void fct (int p) ;
   int n=3 ; /* variable locale init à 3 */
   fct(n) ;
   return 0 ;
}
void fct (int p)
{
   printf("%d %d", n, p) ;
}
```

Q'affiche le programme suivant ?

```c
#include <stdio.h>
int n=5 ; /* variable globale init à 5 */
int main(void)
{
   void fct (int p) ;
   int n=3 ; /* variable locale init à 3 */
   fct(n) ;
   return 0 ;
}
void fct (int p)
{
   printf("%d %d", n, p) ;
}
```

5 3

Variables locales statiques (1/2)

Pour déclarer une variable locale "statique", il faut précéder sont type du mot-clef **static**.

- **Durée de vie** : elles existent pendant toute l'exécution du programme dans lequel elles apparaissent.
- **Portée** : elles ne sont "visibles" qu'à l'intérieur de la fonction où elles sont déclarées.
- **Classe d'allocation** : la classe d'*allocation statique*, car leurs emplacements mémoires sont parfaitement définis lors de l'édition de lien. Elles sont initialisées à zéro, sauf si on leur spécifie explicitement une valeur d'initialisation.

Les valeurs des variables locales "statiques" sont conservées d'un appel au suivant.

Variables locales statiques (2/2)

```
#include <stdio.h>
int main(void)
{
   void fct (void) ;
   int n ;
   for ( n=1 ; n<=5 ; n++)
        fct();
   return 0;
}
void fct (void)
{
  static int i ;
  i++ ;
  printf("appel numéro : %d\n", i);
}
```

Variables locales à un bloc

Le C offre la possibilité de déclarer des variables locales à un bloc, en les déclarant au début du bloc.

Règle : Si $m \geq 2$ variables avec le même identificateur, mais avec un niveau de déclaration différent coexistent, alors celle du niveau le plus imbriqué masque les autres.

```
#include <stdio.h>
int main(void)
{
   int n ;
   ...
   {
      int n ;
      double x ;
      long double z ;
   }
   ...
   return 0;
}
```

Variables locales à un bloc

Le C offre la possibilité de déclarer des variables locales à un bloc, en les déclarant au début du bloc.

Règle : Si $m \geq 2$ variables avec le même identificateur, mais avec un niveau de déclaration différent coexistent, alors celle du niveau le plus imbriqué masque les autres.

```
#include <stdio.h>
int main(void)
{
   int n ;
   ...
   {
      int n ;
      double x ;
      long double z ;
   }
   ...
   return 0;
}
```

Plan

Aspects plutôt externes

Un programme de qualité aura une durée de vie plus longue.

1. **(Capacité) fonctionnelle** : conformité des résultats aux exigences utilisateurs (certaines normes et la sécurité), précision des résultats.
2. **Fiabilité** (robustesse) : résultats corrects quelles que soient les conditions d'exploitation (plus intensive...).
3. **Performance** mémoire et temps de calculs.
4. **Facilité d'utilisation** : effort pour apprendre à utiliser correctement le (sous-)programme.

Aspects plutôt externes

Un programme de qualité aura une durée de vie plus longue.

1. **(Capacité) fonctionnelle** : conformité des résultats aux exigences utilisateurs (certaines normes et la sécurité), précision des résultats.
2. **Fiabilité** (robustesse) : résultats corrects quelles que soient les conditions d'exploitation (plus intensive...).
3. **Performance** mémoire et temps de calculs.
4. **Facilité d'utilisation** : effort pour apprendre à utiliser correctement le (sous-)programme.

Aspects plutôt externes

Un programme de qualité aura une durée de vie plus longue.

1. **(Capacité) fonctionnelle** : conformité des résultats aux exigences utilisateurs (certaines normes et la sécurité), précision des résultats.
2. **Fiabilité** (robustesse) : résultats corrects quelles que soient les conditions d'exploitation (plus intensive...).
3. **Performance** mémoire et temps de calculs.
4. **Facilité d'utilisation** : effort pour apprendre à utiliser correctement le (sous-)programme.

Aspects plutôt externes

Un programme de qualité aura une durée de vie plus longue.

1. **(Capacité) fonctionnelle** : conformité des résultats aux exigences utilisateurs (certaines normes et la sécurité), précision des résultats.
2. **Fiabilité** (robustesse) : résultats corrects quelles que soient les conditions d'exploitation (plus intensive...).
3. **Performance** mémoire et temps de calculs.
4. **Facilité d'utilisation** : effort pour apprendre à utiliser correctement le (sous-)programme.

Aspects plutôt internes

Un programme de qualité aura une durée de vie plus longue.

1. **Maintenabilité** : effort nécessaire en vue de corriger ou de transformer le programme.
2. **Portabilité du code** : fonctionner dans des environnements matériels ou logiciels différents.

Aspects plutôt internes

Un programme de qualité aura une durée de vie plus longue.

1. **Maintenabilité** : effort nécessaire en vue de corriger ou de transformer le programme.
2. **Portabilité du code** : fonctionner dans des environnements matériels ou logiciels différents.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (1/2)

- Programmer avec les normes les plus restrictives (C ANSI).
- Choisir les types en fonction de la précision attendue.
- Respecter un *nommage des variables* et s'y tenir.
- Respecter un *nommage des fonctions* et s'y tenir.
- Minimiser le nombre de variables, de paramètres de fonctions et le nombre de calculs.
- Eviter les fuites mémoires : faille de sécurité + perte de performance.

Règles de programmation (2/2)

- Fournir une documentation minimale :
  - Commenter certaines lignes de son code.
  - Description brève de ce que fait une fonction, de ses paramètres et de sa valeur de retour.
  - Spécifier les hypothèses d'utilisation d'une fonction (**pré et post conditions**).
- Vérifier ces pré et post conditions en *mode de déboguage* : **#include** <assert.h> et fonction *assert*.
- Vérifier que le programme fonctionne dans toutes les conditions d'utilisation possibles : **Tests unitaires** en boite noire et **débogage**.

Règles de programmation (2/2)

- Fournir une documentation minimale :
  - Commenter certaines lignes de son code.
  - Description brève de ce que fait une fonction, de ses paramètres et de sa valeur de retour.
  - Spécifier les hypothèses d'utilisation d'une fonction (**pré et post conditions**).
- Vérifier ces pré et post conditions en *mode de déboguage* : **#include** <assert.h> et fonction *assert*.
- Vérifier que le programme fonctionne dans toutes les conditions d'utilisation possibles : **Tests unitaires** en boite noire et **débogage**.

Règles de programmation (2/2)

- Fournir une documentation minimale :
  - Commenter certaines lignes de son code.
  - Description brève de ce que fait une fonction, de ses paramètres et de sa valeur de retour.
  - Spécifier les hypothèses d'utilisation d'une fonction (**pré et post conditions**).
- Vérifier ces pré et post conditions en *mode de déboguage* : **#include** <assert.h> et fonction *assert*.
- Vérifier que le programme fonctionne dans toutes les conditions d'utilisation possibles : **Tests unitaires** en boite noire et **débogage**.

Règles de programmation (2/2)

- Fournir une documentation minimale :
  - Commenter certaines lignes de son code.
  - Description brève de ce que fait une fonction, de ses paramètres et de sa valeur de retour.
  - Spécifier les hypothèses d'utilisation d'une fonction (**pré et post conditions**).
- Vérifier ces pré et post conditions en *mode de déboguage* : **#include** <assert.h> et fonction *assert*.
- Vérifier que le programme fonctionne dans toutes les conditions d'utilisation possibles : **Tests unitaires** en boite noire et **débogage**.

Règles de programmation (2/2)

- Fournir une documentation minimale :
  - Commenter certaines lignes de son code.
  - Description brève de ce que fait une fonction, de ses paramètres et de sa valeur de retour.
  - Spécifier les hypothèses d'utilisation d'une fonction (**pré et post conditions**).
- Vérifier ces pré et post conditions en *mode de déboguage* : **#include** <assert.h> et fonction *assert*.
- Vérifier que le programme fonctionne dans toutes les conditions d'utilisation possibles : **Tests unitaires** en boite noire et **débogage**.

Nommage en général

- **1 seule langue** (anglais ou français).
- Des **noms simples et représentatifs** sans ambiguïté.
- Ne pas utiliser les caractères : -, les symboles ($, @, ...), les caractères accentués.
- **1 convention fixe pour la casse et la séparation des mots** :
  - Les variables : tout en minuscule et separation des mots par _ **ou** tout en minuscule et séparation des mots en mettant la première lettre du mot suivant en majuscule.
  - Les symboles et constantes : tout en majuscule et séparation des mots par _.

Nommage en général

- **1 seule langue** (anglais ou français).
- Des **noms simples et représentatifs** sans ambiguïté.
- Ne pas utiliser les caractères : -, les symboles ($, @, ...), les caractères accentués.
- **1 convention fixe pour la casse et la séparation des mots** :
  - Les variables : tout en minuscule et separation des mots par _ **ou** tout en minuscule et séparation des mots en mettant la première lettre du mot suivant en majuscule.
  - Les symboles et constantes : tout en majuscule et séparation des mots par _.

Nommage en général

- **1 seule langue** (anglais ou français).
- Des **noms simples et représentatifs** sans ambiguïté.
- Ne pas utiliser les caractères : -, les symboles ($, @, ...), les caractères accentués.
- **1 convention fixe pour la casse et la séparation des mots** :
  - Les variables : tout en minuscule et separation des mots par _ **ou** tout en minuscule et séparation des mots en mettant la première lettre du mot suivant en majuscule.
  - Les symboles et constantes : tout en majuscule et séparation des mots par _.

Nommage en général

- **1 seule langue** (anglais ou français).
- Des **noms simples et représentatifs** sans ambiguïté.
- Ne pas utiliser les caractères : -, les symboles ($, @, ...), les caractères accentués.
- **1 convention fixe pour la casse et la séparation des mots** :
  - Les variables : tout en minuscule et separation des mots par _ **ou** tout en minuscule et séparation des mots en mettant la première lettre du mot suivant en majuscule.
  - Les symboles et constantes : tout en majuscule et séparation des mots par _.

Nommage en général

- **1 seule langue** (anglais ou français).
- Des **noms simples et représentatifs** sans ambiguïté.
- Ne pas utiliser les caractères : -, les symboles ($, @, ...), les caractères accentués.
- **1 convention fixe pour la casse et la séparation des mots** :
  - Les variables : tout en minuscule et separation des mots par _ **ou** tout en minuscule et séparation des mots en mettant la première lettre du mot suivant en majuscule.
  - Les symboles et constantes : tout en majuscule et séparation des mots par _.

Conventions pour le nommage des variables

Conventions :

- i, j, k, n, m, cpt, $cpt1$, $cpt2$... sont utilisées pour des variables entières, souvent comme variables de contrôle de boucles ou comme compteurs.
- 1 seule lettre n'est admise que pour des compteurs de boucle (i, j, k...) ou pour les variables mathématiques (x, y, z...).
- abréviations courantes : *val* pour valeur, *nb* pour nombre, *surf* pour surface, *tmp* pour temporaire, *inter* pour intermédiaire, *x2* pour x au carré...
- Ne pas utiliser des noms succeptibles d'être utilisés pas des fonctions définies dans `math.h` ou plus généralement dans des bibliothèques mathématiques : e.g. préférer *valMin* et *valMax* à *min* et *max*.

Conventions pour le nommage des variables

Conventions :

- i, j, k, n, m, cpt, $cpt1$, $cpt2$... sont utilisées pour des variables entières, souvent comme variables de contrôle de boucles ou comme compteurs.
- 1 seule lettre n'est admise que pour des compteurs de boucle (i, j, k...) ou pour les variables mathématiques (x, y, z...).
- abréviations courantes : *val* pour valeur, *nb* pour nombre, *surf* pour surface, *tmp* pour temporaire, *inter* pour intermédiaire, *x2* pour x au carré...
- Ne pas utiliser des noms succeptibles d'être utilisés pas des fonctions définies dans `math.h` ou plus généralement dans des bibliothèques mathématiques : e.g. préférer *valMin* et *valMax* à *min* et *max*.

Conventions pour le nommage des variables

Conventions :

- i, j, k, n, m, cpt, $cpt1$, $cpt2$... sont utilisées pour des variables entières, souvent comme variables de contrôle de boucles ou comme compteurs.
- 1 seule lettre n'est admise que pour des compteurs de boucle (i, j, k...) ou pour les variables mathématiques (x, y, z...).
- abréviations courantes : *val* pour valeur, *nb* pour nombre, *surf* pour surface, *tmp* pour temporaire, *inter* pour intermédiaire, *x2* pour x au carré...
- Ne pas utiliser des noms succeptibles d'être utilisés pas des fonctions définies dans math.h ou plus généralement dans des bibliothèques mathématiques : e.g. préférer *valMin* et *valMax* à *min* et *max*.

Conventions pour le nommage des variables

Conventions :

- i, j, k, n, m, cpt, $cpt1$, $cpt2$... sont utilisées pour des variables entières, souvent comme variables de contrôle de boucles ou comme compteurs.
- 1 seule lettre n'est admise que pour des compteurs de boucle (i, j, k...) ou pour les variables mathématiques (x, y, z...).
- abréviations courantes : *val* pour valeur, *nb* pour nombre, *surf* pour surface, *tmp* pour temporaire, *inter* pour intermédiaire, *x2* pour x au carré...
- Ne pas utiliser des noms succeptibles d'être utilisés pas des fonctions définies dans `math.h` ou plus généralement dans des bibliothèques mathématiques : e.g. préférer *valMin* et *valMax* à *min* et *max*.

Les commentaires INUTILES

```c
#include <stdlib.h>
int main(void) {
   /* début du programme */
   int i=0; /* i est initialisée à 0 */

   while( i < 15)
      i++; /* i est incrémentée */

   system("PAUSE"); /* on met pause */
   return 0;
}
```

Les commentaires UTILES (1/2)

```c
/*
Nom fichier : XXXXX.c
Auteur(s) : M. .....
Date dernière modification :
Version :
Copyright :
*/
#include <stdlib.h> /* pour system */
int main(void) {
   int i=0; /* i est une variable compteur initialisée à 0 */

   While( i < 15)
      i++; /* i est mise à jour */

   system("PAUSE"); /* Pause de la sortie console pour lire les eventuels résultats */
   return 0;
}
```

Les commentaires UTILES (2/2)

```
/*
Nom fonction : au_carre
Description brève : Calcule et retourne le carré d'une valeur codée sur un double

Paramètre(s) :
   - donnée : x de type double

Valeur de retour de type double

Précondition(s) : aucune
Postcondition(s) : la valeur de retour est égale à la valeur d'entrée au carré
                   la valeur de retour doit être >=0
*/
double au_carre(double x)
{
   return x * x;
}
```

Qualité fonctionnelle

- Vérifier qu'un (sous-)programme réagit de la façon prévu par les spécifications : **valider un niveau de qualité en accord avec les attentes métier (juridique...) et technique**.
- 3 clés d'un test unitaire en boite noire : les *données en entrée*, le *(sous-)programme à tester* et les *résultats attendus*.
- Un (sous-)programme est testable *automatiquement* (≈en boite noire) seulement si on peut récupérer les informations nécessaires pour déterminer son bon déroulement.
- **Un test réussi seulement si les résultats du (sous-) programme sont conformes aux résultats attendus**.
- Un test ne permet pas d'identifier la cause d'une erreur, ni de la réparer : on mesure la qualité à un instant t.

Qualité fonctionnelle

- Vérifier qu'un (sous-)programme réagit de la façon prévu par les spécifications : **valider un niveau de qualité en accord avec les attentes métier (juridique...) et technique**.
- 3 clés d'un test unitaire en boite noire : les *données en entrée*, le *(sous-)programme à tester* et les *résultats attendus*.
- Un (sous-)programme est testable *automatiquement* (≈en boite noire) seulement si on peut récupérer les informations nécessaires pour déterminer son bon déroulement.
- **Un test réussi seulement si les résultats du (sous-) programme sont conformes aux résultats attendus**.
- Un test ne permet pas d'identifier la cause d'une erreur, ni de la réparer : on mesure la qualité à un instant t.

Qualité fonctionnelle

- Vérifier qu'un (sous-)programme réagit de la façon prévu par les spécifications : **valider un niveau de qualité en accord avec les attentes métier (juridique...) et technique**.
- 3 clés d'un test unitaire en boite noire : les *données en entrée*, le *(sous-)programme à tester* et les *résultats attendus*.
- Un (sous-)programme est testable *automatiquement* (≈en boite noire) seulement si on peut récupérer les informations nécessaires pour déterminer son bon déroulement.
- **Un test réussi seulement si les résultats du (sous-) programme sont conformes aux résultats attendus.**
- Un test ne permet pas d'identifier la cause d'une erreur, ni de la réparer : on mesure la qualité à un instant t.

Qualité fonctionnelle

- Vérifier qu'un (sous-)programme réagit de la façon prévu par les spécifications : **valider un niveau de qualité en accord avec les attentes métier (juridique...) et technique**.
- 3 clés d'un test unitaire en boite noire : les *données en entrée*, le *(sous-)programme à tester* et les *résultats attendus*.
- Un (sous-)programme est testable *automatiquement* (≈en boite noire) seulement si on peut récupérer les informations nécessaires pour déterminer son bon déroulement.
- **Un test réussi seulement si les résultats du (sous-) programme sont conformes aux résultats attendus**.
- Un test ne permet pas d'identifier la cause d'une erreur, ni de la réparer : on mesure la qualité à un instant t.

Qualité fonctionnelle

- Vérifier qu'un (sous-)programme réagit de la façon prévu par les spécifications : **valider un niveau de qualité en accord avec les attentes métier (juridique...) et technique**.
- 3 clés d'un test unitaire en boite noire : les *données en entrée*, le *(sous-)programme à tester* et les *résultats attendus*.
- Un (sous-)programme est testable *automatiquement* (≈en boite noire) seulement si on peut récupérer les informations nécessaires pour déterminer son bon déroulement.
- **Un test réussi seulement si les résultats du (sous-) programme sont conformes aux résultats attendus**.
- Un test ne permet pas d'identifier la cause d'une erreur, ni de la réparer : on mesure la qualité à un instant t.

Conception des tests unitaires

Ordonnancement des tests :

- Les cas généraux avant les cas particuliers.
- Les cas les plus prioritaires/critiques avant les autres.
- Si un (sous-)programme A dépend d'un sous-programme B alors il faut tester B avant A.

Conception des tests unitaires

Ordonnancement des tests :

- Les cas généraux avant les cas particuliers.
- Les cas les plus prioritaires/critiques avant les autres.
- Si un (sous-)programme A dépend d'un sous-programme B alors il faut tester B avant A.

Conception des tests unitaires

Ordonnancement des tests :

- Les cas généraux avant les cas particuliers.
- Les cas les plus prioritaires/critiques avant les autres.
- Si un (sous-)programme A dépend d'un sous-programme B alors il faut tester B avant A.

Caractéristiques des tests unitaires

2 objectifs des tests unitaires : **couverture du code** testé et **couverture des données** testées.

- Un test unitaire doit s'exécuter rapidement (< quelques secondes).
- Un test unitaire doit être indépendant des autres tests.
- Un test unitaire doit être répétable.
- Un test unitaire doit se valider/s'invalider lui-même.

Caractéristiques des tests unitaires

2 objectifs des tests unitaires : **couverture du code** testé et **couverture des données** testées.

- Un test unitaire doit s'exécuter rapidement (< quelques secondes).
- Un test unitaire doit être indépendant des autres tests.
- Un test unitaire doit être répétable.
- Un test unitaire doit se valider/s'invalider lui-même.

Caractéristiques des tests unitaires

2 objectifs des tests unitaires : **couverture du code** testé et **couverture des données** testées.

- Un test unitaire doit s'exécuter rapidement (< quelques secondes).
- Un test unitaire doit être indépendant des autres tests.
- Un test unitaire doit être répétable.
- Un test unitaire doit se valider/s'invalider lui-même.

Caractéristiques des tests unitaires

2 objectifs des tests unitaires : **couverture du code** testé et **couverture des données** testées.

- Un test unitaire doit s'exécuter rapidement (< quelques secondes).
- Un test unitaire doit être indépendant des autres tests.
- Un test unitaire doit être répétable.
- Un test unitaire doit se valider/s'invalider lui-même.

Caractéristiques des tests unitaires

2 objectifs des tests unitaires : **couverture du code** testé et **couverture des données** testées.

- Un test unitaire doit s'exécuter rapidement (< quelques secondes).
- Un test unitaire doit être indépendant des autres tests.
- Un test unitaire doit être répétable.
- Un test unitaire doit se valider/s'invalider lui-même.

Conception d'un test unitaire : mauvais exemple

```
/*
Nom du test : test_au_carre
Description brève : vérifie que la fonction au carre fonctionne pour
                    un double <0, nul et >0 ; vérifie que le résultat
                    est valide

Précondition(s) : la fonction au_carre est bien definie
Postcondition(s) : le test se termine uniquement si le test réussi.
*/
void test_au_carre(void){
   assert( au_carre( 0. ) == 0.0 ) ;

   assert( au_carre( 1. ) == 1.0 ) ;
   assert( au_carre( 2. ) == 4.0 ) ;
   assert( au_carre( 3. ) == 9.0 ) ;
   assert( au_carre( 3000000000. ) == 9e+18 ) ;

   assert( au_carre( -1. ) == 1.0 ) ;
   assert( au_carre( -2. ) == 4.0 ) ;
   assert( au_carre( -3. ) == 9.0 ) ;
   assert( au_carre( -3000000000. ) == 9e+18 ) ; }

int main(void){
   test_au_carre() ;
   test_au_cube() ;  /* ici 2 tests sont lancés ensembles... */
   ... }
```

Conception d'un test unitaire en boite noire

Un test doit pouvoir être lancé indépendament des autres tests.

Une solution adaptée consiste à :

- **écrire un fichier source par test**, le nom du fichier source permettant d'identifier le test de façon unique.
- **mettre les intructions du test dans la fonction** `main` **du fichier source.**

Conception d'un test unitaire en boite noire

Un test doit pouvoir être lancé indépendamment des autres tests.

Une solution adaptée consiste à :

- **écrire un fichier source par test**, le nom du fichier source permettant d'identifier le test de façon unique.
- **mettre les intructions du test dans la fonction** `main` **du fichier source.**

Conception d'un test unitaire en boite noire

Un test doit pouvoir être lancé indépendament des autres tests.

Une solution adaptée consiste à :

- **écrire un fichier source par test**, le nom du fichier source permettant d'identifier le test de façon unique.
- **mettre les intructions du test dans la fonction** `main` **du fichier source**.

Conception d'un test unitaire : bon exemple

```c
/*
Nom du fichier : test_au_carre.c
*/
#include <assert.h>
#include "au_carre.h" /* on inclut le module utilisateur contenant la fonction au_carre */

int main(void){
   assert( au_carre( 0. ) == 0.0 ) ;

   assert( au_carre( 1. ) == 1.0 ) ;
   assert( au_carre( 2. ) == 4.0 ) ;
   assert( au_carre( 3. ) == 9.0 ) ;
   assert( au_carre( 3000000000. ) == 9e+18 ) ;

   assert( au_carre( -1. ) == 1.0 ) ;
   assert( au_carre( -2. ) == 4.0 ) ;
   assert( au_carre( -3. ) == 9.0 ) ;
   assert( au_carre( -3000000000. ) == 9e+18 ) ;

   return 0 ;
}
```

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).

Il est possible de déboguer un programme C à l'aide d'un débogueur. Une aide sur le débogage est disponible sur claroline-connect.

Démo :

1. Créer un projet C sous Code : :Blocks.
2. Sélectionner la cible (*Build target*) Debug et vérifier que l'option -g est bien sélectionnée.
3. Optionnel : en profiter pour cocher les options pour vérifier si votre code respecte les normes en vigueur.
4. Copier un programme avec une fonction en plus du main (e.g. celui avec la fonction echange).
5. Y mettre un point d'arrêt (*breakpoint* en anglais).
6. Lancer le débogage (F8).