Initiation à l'algorithmique
Les fonctions

Mohamed MESSABIHI

mohamed.messabihi@gmail.com

Université de Tlemcen
Département d'informatique
1ère année MI

https://sites.google.com/site/informatiquemessabihi/

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commençe par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Pourquoi les fonctions ?

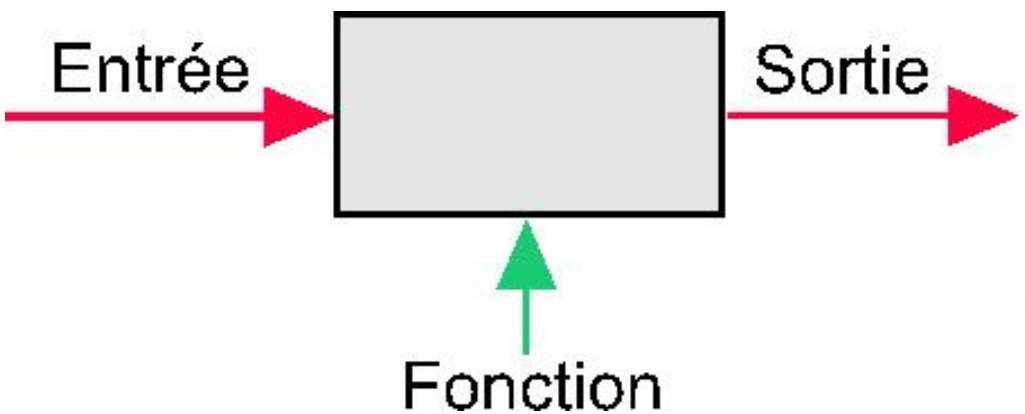
- Un programme en langage C commence par la fonction main.
- Jusqu'ici nous sommes restés à l'intérieur de la fonction main. Nous n'en sommes jamais sortis.
- Ce n'est pas « mal », mais ce n'est pas ce que les programmeurs en C font dans la réalité.
- Quasiment aucun programme n'est écrit uniquement à l'intérieur des accolades de la fonction main.
- Jusqu'ici nos programmes étaient courts, donc ça ne posait pas de gros problèmes
- Mais imaginez des plus gros programmes qui font des milliers de lignes de code

Solution : Notion de fonction

- On doit donc apprendre à nous organiser.
- On doit découper nos programmes en petits bouts.
- Chaque « petit bout de programme » sera ce qu'on appelle une **fonction**.

Fonction

Une fonction exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis.

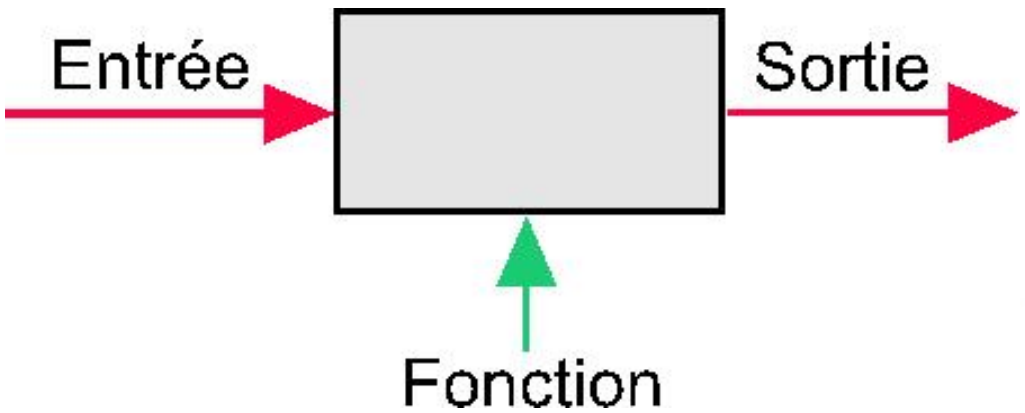


Solution : Notion de fonction

- On doit donc apprendre à nous organiser.
- On doit découper nos programmes en petits bouts.
- Chaque « petit bout de programme » sera ce qu'on appelle une **fonction**.

Fonction

Une fonction exécute des actions et renvoie un résultat. C'est un morceau de code qui sert à faire quelque chose de précis.



Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent
2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)
4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent
2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)
4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Intérêts des fonctions

Les fonctions sont des modules (groupe d'instructions) indépendants désignés par un nom. Elles ont plusieurs intérêts :

1. Elles permettent de **"factoriser"** les programmes, càd de mettre en commun les parties qui se répètent

2. Elles permettent une **structuration** et **une meilleure lisibilité** des programmes

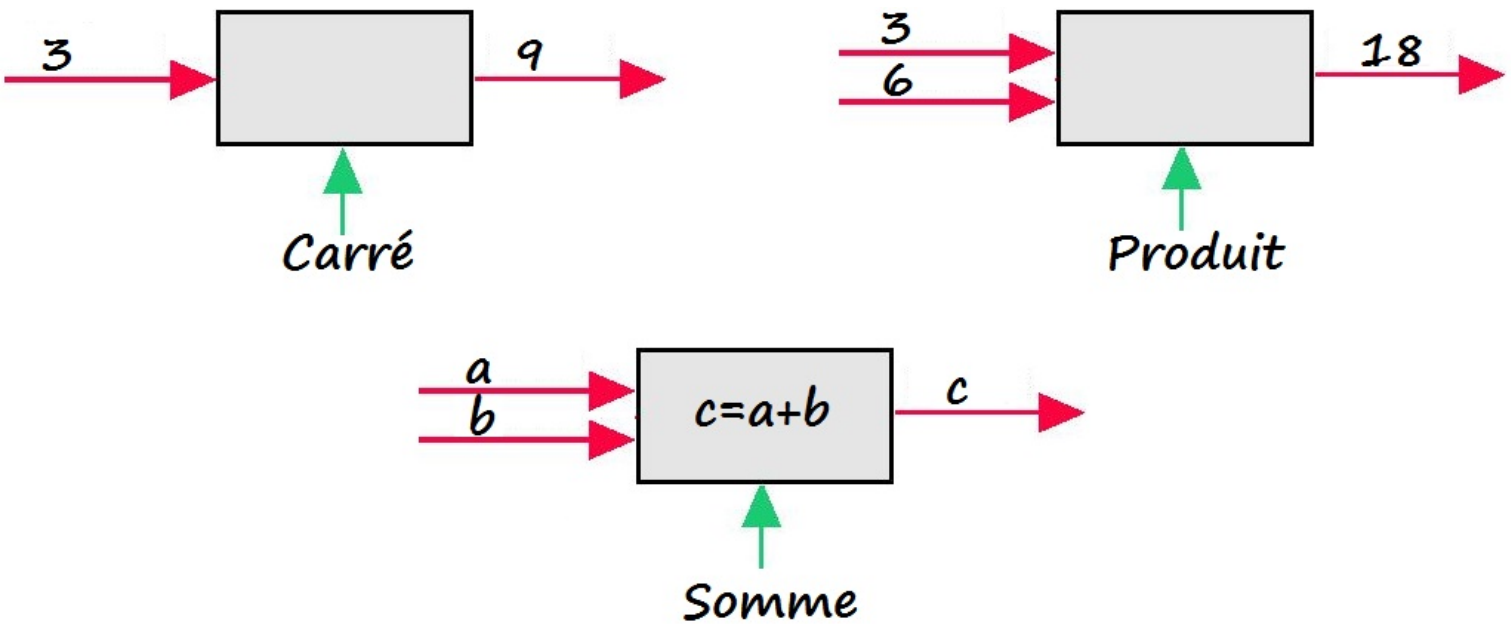
3. Elles **facilitent la maintenance** du code (il suffit de modifier une seule fois)

4. Elles peuvent éventuellement être **réutilisées** dans d'autres programmes

Principe

Une fonction est définie par trois éléments :

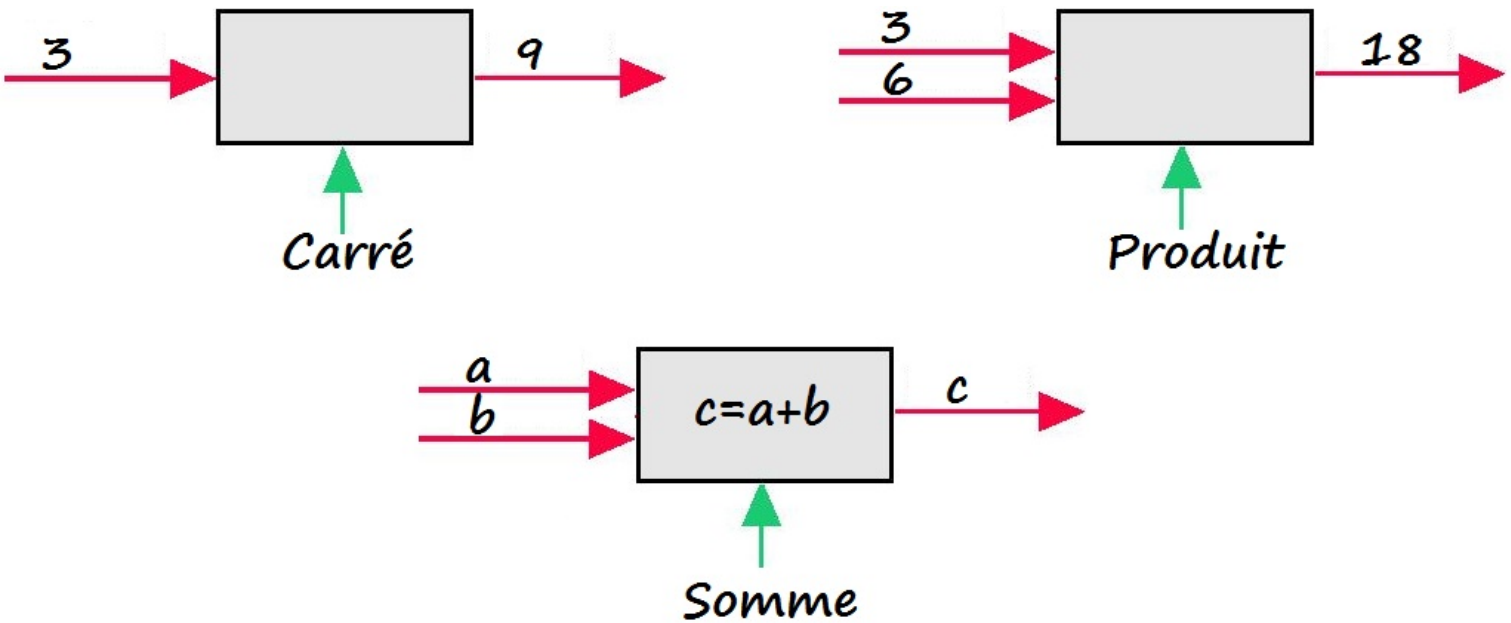
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

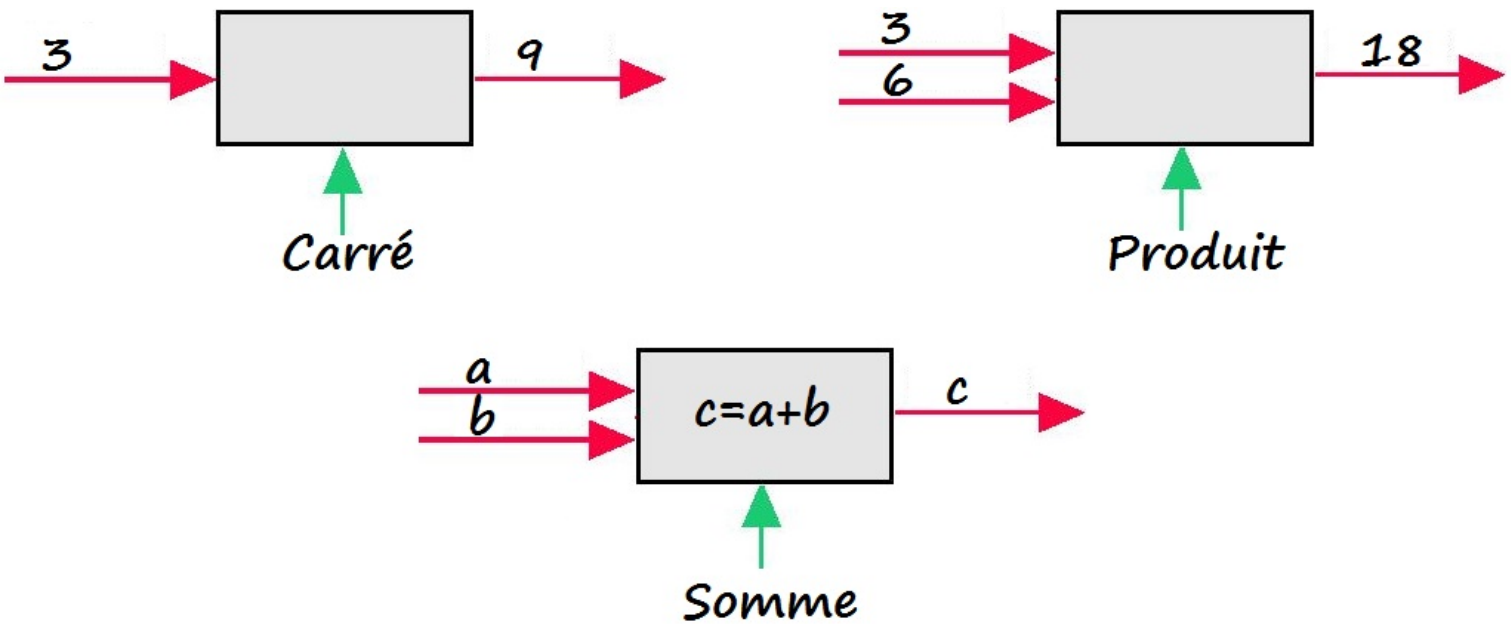
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

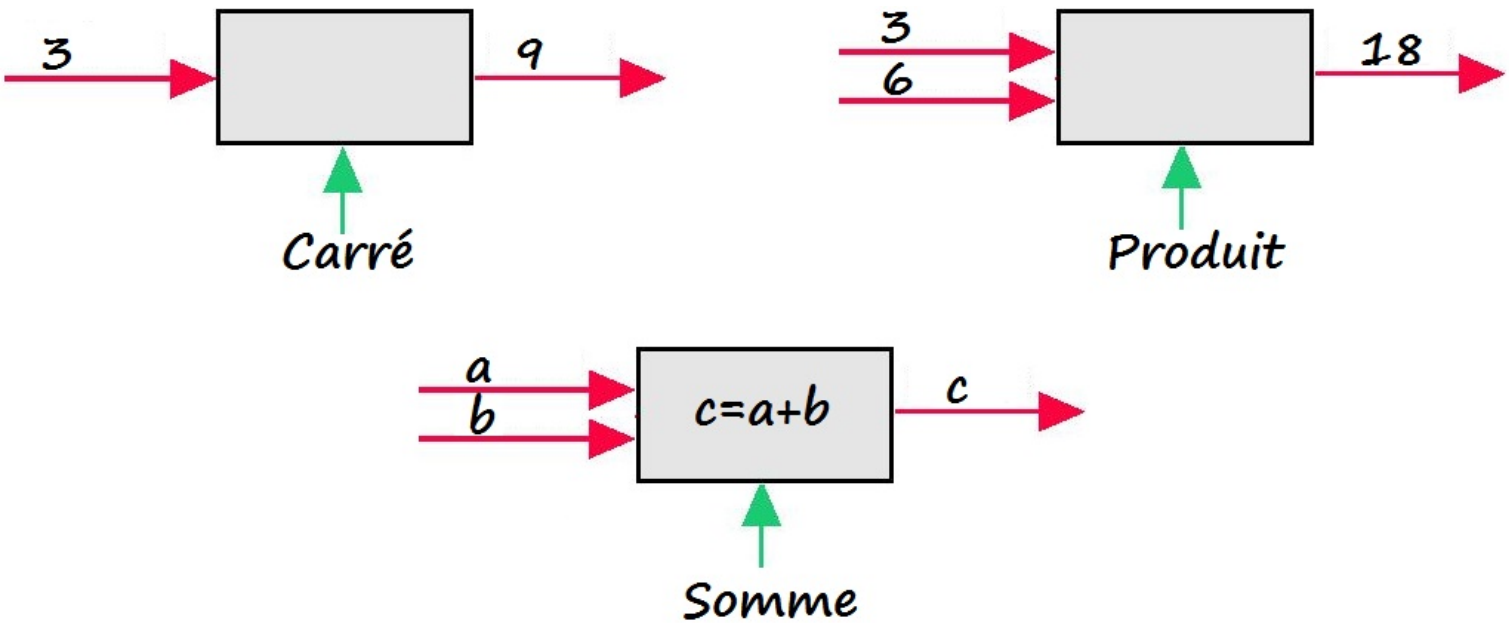
1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Principe

Une fonction est définie par trois éléments :

1. **L'entrée :** on fait « rentrer » des informations dans la fonction (en lui donnant des informations avec lesquelles travailler).
2. **Les calculs :** grâce aux informations qu'elle a reçues en entrée, la fonction travaille.
3. **La sortie :** une fois qu'elle a fini ses calculs, la fonction renvoie un résultat. C'est ce qu'on appelle la sortie, ou encore le retour.



Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Déclarer une fonction

Syntaxe :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <Corps de la fonction>
   }
```

- **type de retour :** (correspond à la sortie) c'est le type de la fonction. Ce type dépend du résultat que la fonction renvoie (**int**, **double**, **void**,...)
- **nomFonction :** c'est le nom de votre fonction. Il doit respecter les mêmes règles que pour les variables.
- **parametres :** (correspond à l'entrée) entre parenthèses, on envoie des paramètres à la fonction.

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Type de retour Void

- Il se peut que l'on est besoin de coder une fonction qui ne retourne aucun résultat.
- C'est un cas courant en C. Ce genre de fonction est appelé procédure.
- Pour écrire une procédure, il faut indiquer à la fonction en question qu'elle ne doit rien retourner.
- Pour ce faire, il existe un "type de retour" spécial : **void**. Ce type signifie "vide", et sert à indiquer que la fonction n'a pas de résultat.

Exemple

```
void afficherMenu()
    {
        printf("===== Menu =====\n\n");
        printf("1. Royal Cheese \n");
        printf("2. Big Burger \n");
        printf("3. Complet Poulet \n");
        printf("4. Panini Thon \n");
    }
```

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Les paramètres des fonctions

- Un paramètre sert à fournir des informations à la fonction lors de son exécution
- Si la fonction nécessite plusieurs paramètres, il suffit de les séparer par une virgule.

Exemple :

```
int Somme(int a, int b)
{
    return a + b;
}
// fonctions sans parametres
void bonjour()
{
    printf("Bonjour");
}
```

- Les paramètres doivent avoir des noms différents
- Il est aussi possible de ne pas mettre d'arguments dans une fonction. Dans ce cas on écrit **()** ou **(void)**.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
  {
   <declarations locales>
   <instructions>
  }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
  {
   <declarations locales>
   <instructions>
  }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

Le corps d'une fonction

- Le corps d'une fonction C est définie à l'aide d'un bloc d'instructions.
- Un bloc d'instructions est encadré d'accolades et composé de deux parties :

```
<Type_Retour> <Nom_Fonct> (<Parametres>)
   {
    <declarations locales>
    <instructions>
   }
```

- Ceci est vrai pour tous les blocs d'instructions (fonction, if, while ou for, etc).
- Les variables déclarées dans une fonction ne sont accessibles que dans cette fonction, et pas de l'extérieur.
- Souvent, ces variables déclarées dans une fonction sont créées quand on commence l'exécution de la fonction, et elles sont supprimées de la mémoire une fois que la fonction renvoie son résultat.

L'instruction return

- L'instruction **return** permet de préciser quel est le résultat que la fonction doit retourner (renvoyer)
- On peut mentionner n'importe quelle expression après un **return**.

Exemple :

```
float polynome (float x, int b, int c)
 {
 float resultat;
 resultat = x * x + b * x + c
 return (resultat) ;

 // est equivalent a

 float polynome (float x, int b, int c)
 {
 return (x * x + b * x + c) ;
 }
```

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

L'instruction return

- L'instruction **return** peut apparaître à plusieurs reprises dans une fonction

Exemple :

```
int produitAbsolu (double u, double v)
 {
        double s ;
        s = u*v ;
        if (s>0) return (s) ;
         else return (-s)
 }
```

- Le type de l'expression dans **return** doit être le même que celui déclaré dans l'en-tête de la fonction. Sinon le compilateur mettra automatiquement en place des instructions de conversion.
- L'instruction **return** définit non seulement la valeur du résultat, mais, en même temps, elle interrompt l'exécution de la fonction en revenant dans la fonction qui l'a appelée.

Utilisation d'une fonction

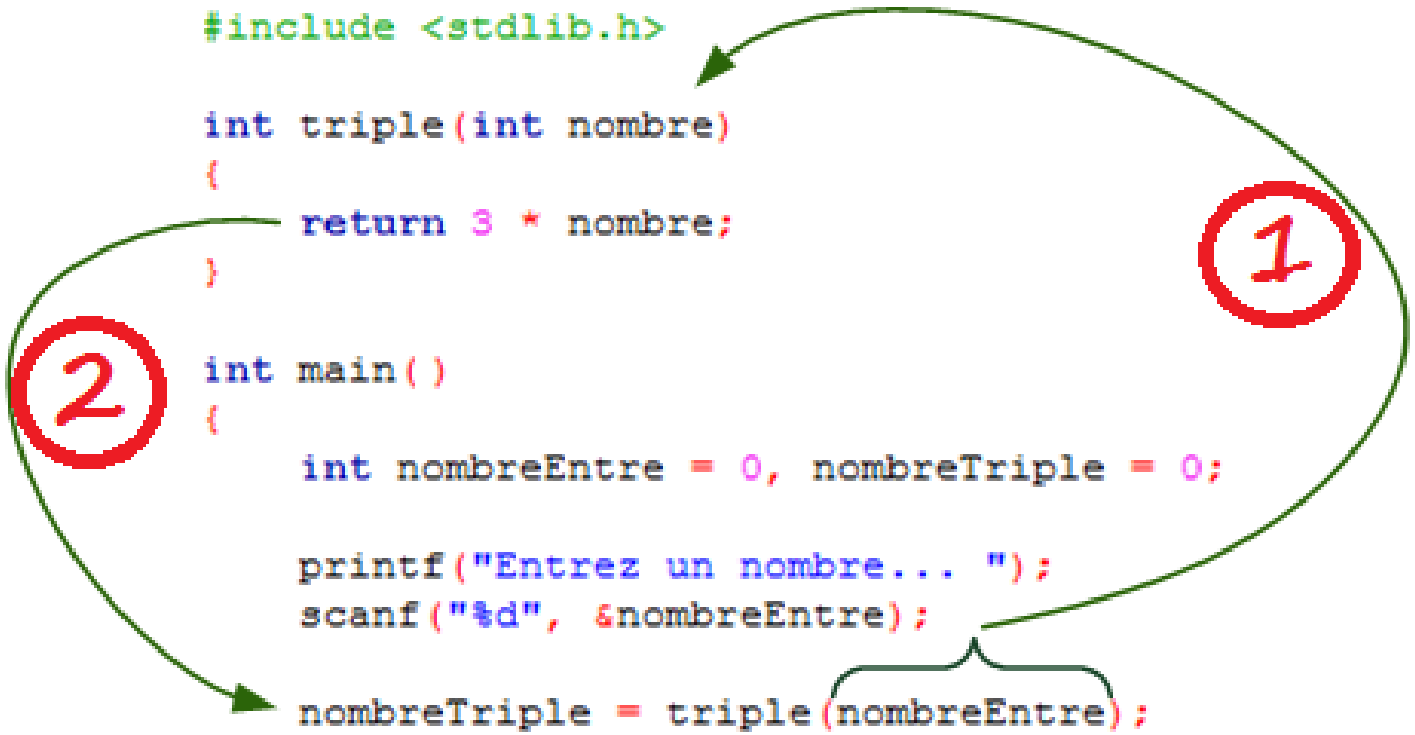
Il suffit de taper le nom de la fonction suivi des paramètres entre parenthèses.

Exemple :

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

int triple(int nombre) // 6
{
    return 3 * nombre; // 7
}
int main() // 1
{
  int nombreEntre = 0, nombreTriple = 0; // 2
  printf("Entrez un nombre... "); // 3
  scanf("%d", &nombreEntre); // 4

  nombreTriple = triple(nombreEntre); // 5

  printf("Le triple de ce nombre est %d\n",nombreTriple);//8
  return 0; // 9
}
```

Appel de fonction

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

int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
    int nombreEntre = 0, nombreTriple = 0;

    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);

    nombreTriple = triple(nombreEntre);
    printf("Le triple de ce nombre est %d\n", nombreTriple);

    return 0;
}
```



On n'est pas obligé de stocker le résultat d'une fonction

Exemple :

```c
int triple(int nombre)
{
    return 3 * nombre;
}
int main()
{
    int nombreEntre = 0;
    printf("Entrez un nombre... ");
    scanf("%d", &nombreEntre);
    // Le resultat de la fonction est directement envoye au
        printf et n'est pas stocke dans une variable
    printf("Le triple de ce nombre est %d\n", triple(
        nombreEntre));
    return 0;
}
```

La fonction **main** appelle la fonction **printf**, qui elle-même appelle la fonction **triple**. C'est une imbrication de fonctions.

Paramètre formels Vs. Paramètre effectifs

```c
int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
        ...
        printf("Le triple est %d\n", triple(nombreEntre));
        ...
}
```

1. Les noms des arguments figurant dans l'en-tête de la fonction se nomment des **« paramètres formels »**. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.
2. Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des **« paramètres effectifs »**. on peut utiliser n'importe quelle expression comme argument effectif.

Paramètre formels Vs. Paramètre effectifs

```c
int triple(int nombre)
{
    return 3 * nombre;
}

int main()
{
        ...
        printf("Le triple est %d\n", triple(nombreEntre));
        ...
}
```

1. Les noms des arguments figurant dans l'en-tête de la fonction se nomment des **« paramètres formels »**. Leur rôle est de permettre, au sein du corps de la fonction, de décrire ce qu'elle doit faire.
2. Les arguments fournis lors de l'utilisation (l'appel) de la fonction se nomment des **« paramètres effectifs »**. on peut utiliser n'importe quelle expression comme argument effectif.

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Variable nombre dans la fonction : 6

Passage de paramètres par valeur

Exemple :

```c
#include <stdio.h>

void fonction(int nombre)
{
    ++nombre;
    printf("Variable nombre dans la fonction : %d\n", nombre
        );
}
int main(void)
{
    int nombre = 5;
    fonction(nombre);
    printf("Variable nombre dans le main : %d\n", nombre);
    return 0;
}
```

Variable nombre dans la fonction : 6
Variable nombre dans le main : 5

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

- En effet, lorsque la fonction est placée avant, le compilateur connait ses paramètres et sa valeur de retour.
- Lors de l'appel de la fonction, le compilateur vérifie que les arguments qu'on lui donne sont bons.
- Si au contraire la fonction est après, le compilateur ne connait pas la fonction.
- Heureusement, il existe une sorte de mode d'emploi qui permet d'indiquer toutes les caractéristiques d'une fonction au compilateur.
- Avec cette indication, on peut placer la fonction où on veut dans le code. Et ce mode d'emploi a un nom : **un prototype**. Un prototype se déclare quasiment comme une fonction :

Exemple :

```
type nom_de_la_fonction(arguments);
```

- Placez le prototype simplement tout en haut de votre fichier et c'est bon ! votre fonction est utilisable partout dans le code

les prototypes

Exemple :

```c
#include <stdio.h>
int carre(int nombre);

int main(void)
{
    int nombre, nombre_au_carre;
    puts("Entrez un nombre :");
    scanf("%d", &nombre);
    nombre_au_carre = carre(nombre);
    printf("Voici le carre de %d : %d\n", nombre,
        nombre_au_carre);
    return 0;
}

int carre(int nombre)
{
    nombre *= nombre;
    return nombre;
}
```

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.
- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.

- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).

- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.

- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.
- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).
- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.
- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.

Remarques

- Le type par défaut est int ; autrement dit : si le type d'une fonction n'est pas déclaré explicitement, elle est automatiquement du type int.

- Il est interdit de définir des fonctions à l'intérieur d'une autre fonction (comme en Pascal).

- En principe, l'ordre des définitions dans le texte du programme ne joue pas de rôle, mais chaque fonction doit être déclarée (prototype) ou définie avant d'être appelée.

- Dans les paramètres du prototype, seuls les types sont vraiment nécessaires, les identificateurs sont facultatifs.