Fonctions

Pierre-Alain FOUQUE
Département d'Informatique
École normale supérieure

Plan

1 – Fonctions
2 – Prototype
3 – Passage d'arguments par valeurs
4 – Retour de résultats

Un programme

Programme (un module) =

- inclusions
  (infos externes)
- types
  (nouveaux types d'objets)
- variables
  (« cases » mémoires à réserver)
- liste de fonctions

Blocs à répétition

Au cours du programme, certaines parties interviennent plusieurs fois (affichage, calcul, etc …)

Autant les écrire une fois pour toutes !

⇒ découper le programme en petits blocs : fonctions

Les fonctions

Fonctions

- en-tête (vue de l'extérieur)
- mode opératoire

  « cases » mémoires à réserver

  liste séquentielle des opérations à effectuer

  ⇒ Instructions

Avantages

- Découper un programme en petits morceaux indépendants

  $\Rightarrow$ aide à écrire un programme correct

  $\Rightarrow$ aide au déboguage
  (fonction par fonction)
- Pouvoir appeler chaque bloc plusieurs fois avec des arguments différents

Prototype de fonction

Le prototype d'une fonction (ou en-tête) précise la « vue de l'extérieur » de la fonction :

- le nom de la fonction
- les types des arguments pris en entrée
- le type de retour

```
int addition(int a, int b)
double puissance(double a, int e)
```

Prototypes (suite)

Exemple de prototype de fonction :

```
double puissance(double a, int e)
```

Prototypes (suite)

Exemple de prototype de fonction :

```
double puissance(double a, int e)
```

Nom de la fonction

Prototypes (suite)

Exemple de prototype de fonction :

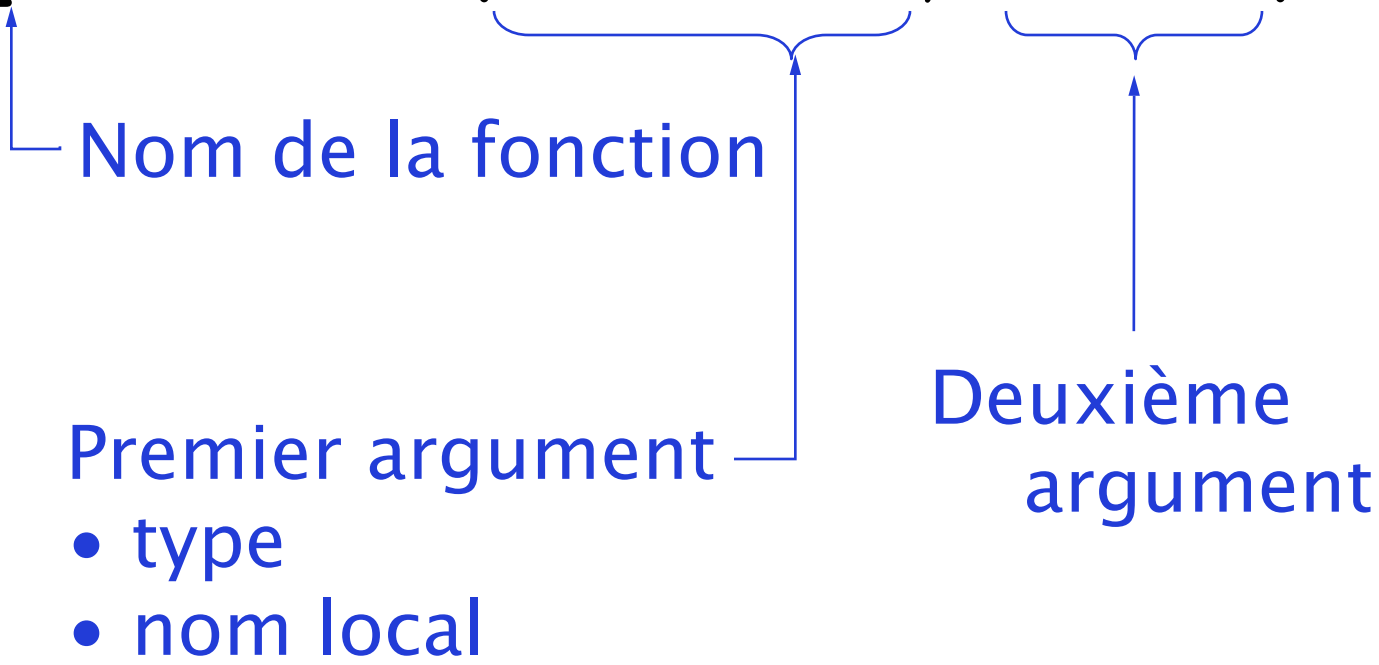
```
double puissance(double a, int e)
```

Premier argument
- type
- nom local

Prototypes (suite)

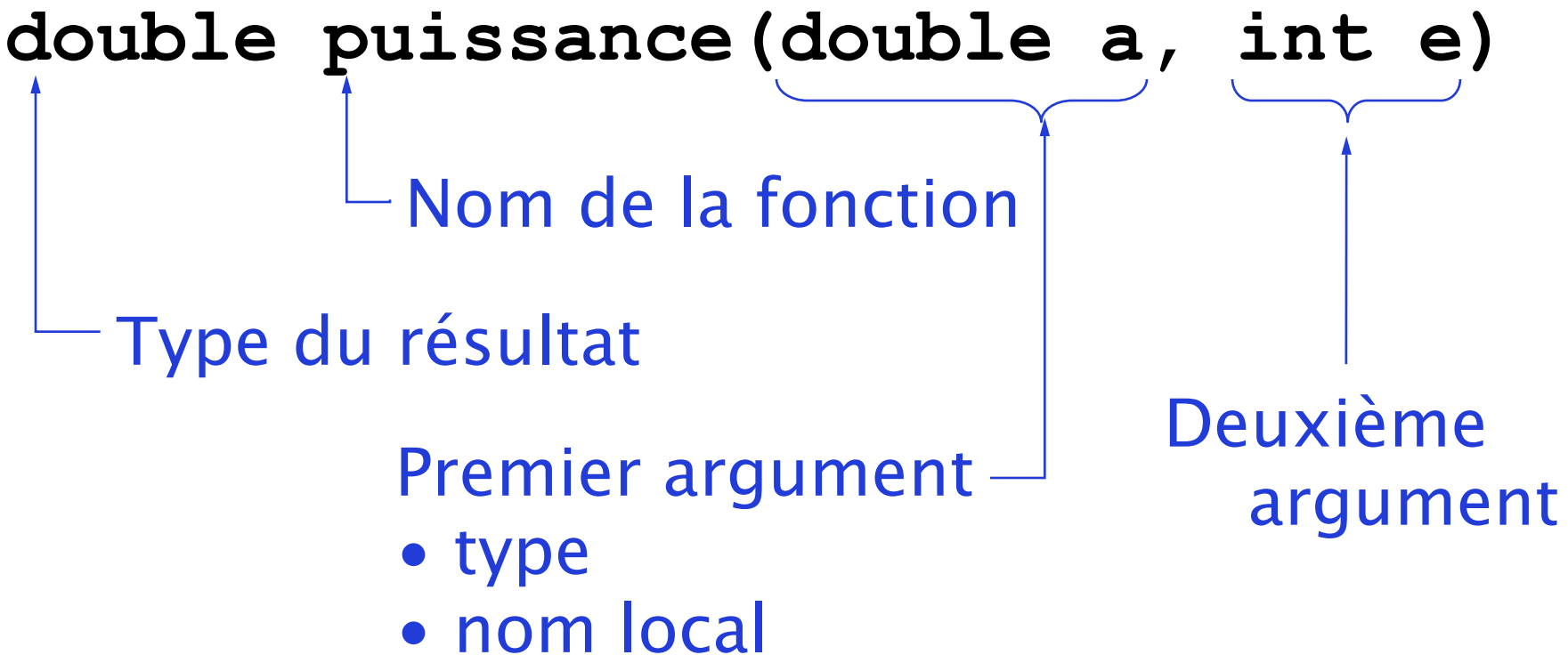
Exemple de prototype de fonction :

```
double puissance(double a, int e)
```



Prototypes (suite)

Exemple de prototype de fonction :



Utilisation

Pour le prototype suivant

```
double puissance(double a, int e)
```

on utilisera la fonction puissance :

```
y = puissance(x,n)
```

où **x** est un **double**, et **n** un **int**

tous deux déclarés et initialisés,

le résultat ira dans la variable **y**,

de type double, déclarée :

⇒ affectation (ou initialisation)

Nom des arguments

Pour le prototype suivant

```
double puissance(double a, int e)
```

⇒ les variables **a** et **e** sont déclarées, et locales à la fonction

lors de l'appel :

```
y = puissance(x,n)
```

- le **contenu** de **x** est copié dans **a**
- le **contenu** de **n** est copié dans **e**

⇒ initialisation des variables **a** et **e**

Retour d'une fonction

```
double puissance(double a, int e)
```

La fonction **puissance** retourne

une valeur de type **double**

⇒ il faut la stocker dans une variable de type **double**

```
double y;
y = puissance(x,n);
```

Exemple I

La fonction **`puissance`** peut être programmée de la façon suivante :

Exemple I

La fonction **puissance** peut être programmée de la façon suivante :

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I

La fonction **puissance** peut être programmée de la façon suivante :

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

La valeur de **b** en fin de fonction est la valeur résultat ⇒ **return b;**

Exemple I – analyse

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I – analyse

```
   double y;

   y = puissance(3.01, 5);
```

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I – analyse

À l'appel de la fonction **`puissance`**

```
   double y;

   y = puissance(3.01, 5);
```

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I – analyse

À l'appel de la fonction **`puissance`**

- le **`double '3.01'`** est stocké dans **`a`**

```
   double y;

   y = puissance(3.01, 5);
```

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I – analyse

À l'appel de la fonction **`puissance`**

- le **`double '3.01'`** est stocké dans **`a`**
- et l'**`int '5'`** est stocké dans **`e`**

```
   double y;

   y = puissance(3.01, 5);
```

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Exemple I – analyse

À l'appel de la fonction **`puissance`**

- le **`double '3.01'`**
  est stocké dans **`a`**
- et l'**`int '5'`**
  est stocké dans **`e`**

Le résultat,
à la fin de la fonction
(la valeur de **`b`**)
est stocké dans **`y`**

```
   double y;

   y = puissance(3.01, 5);
```

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction sans valeur de retour

Une fonction peut ne rien retourner
ex. : une fonction d'affichage

- affiche à l'écran des résultats
- ne retourne rien

⇒ type **`void`**

```
void affiche(int tab[], int n)
```

⇒ pas de return en fin de fonction

Exemple II

L'affichage d'un tableau peut être programmé de la façon suivante :

Exemple II

L'affichage d'un tableau peut être programmé de la façon suivante :

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II

L'affichage d'un tableau peut être programmé de la façon suivante :

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **`void`**

Exemple II

L'affichage d'un tableau peut être programmé de la façon suivante :

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **`void`**

⇒ pas de **`return`**

Exemple II – analyse

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II – analyse

```
   int T[5] = {2, 5, 3, 9, 1};

   affiche(T, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II – analyse

À l'appel de la fonction **affiche**

```
   int T[5] = {2, 5, 3, 9, 1};

   affiche(T, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II – analyse

À l'appel de la fonction **affiche**

- le tableau **T** (pointeur) est stocké dans **tab**

```
int T[5] = {2, 5, 3, 9, 1};

affiche(T, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II – analyse

À l'appel de la fonction **affiche**

- le tableau **T** (pointeur) est stocké dans **tab**
- et l'**int '5'** est stocké dans **n**

```
int T[5] = {2, 5, 3, 9, 1};

affiche(T, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Exemple II – analyse

À l'appel de la fonction **affiche**

- le tableau **T** (pointeur) est stocké dans **tab**
- et l'**int '5'** est stocké dans **n**

À la fin de la fonction (après les affichages), on revient au **main** après l'appel à la fonction, sans rien retourner

```
int T[5] = {2, 5, 3, 9, 1};

affiche(T, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **double**

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**

⇒ **return b**

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**
⇒ **return b**

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**
⇒ pas de **return**

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**

⇒ **return b**

```
   double y;

   y = puissance(3.01, 5);
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**

⇒ pas de **return**

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**

⇒ **return b**

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**

⇒ pas de **return**

```
   double y;

   y = puissance(3.01, 5);
```

⇒ affectation **y =** ...

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

Fonction de type **double**

⇒ **return b**

```
   double y;

   y = puissance(3.01, 5);
```

⇒ affectation **y =** ...

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **void**

⇒ pas de **return**

```
   int T[5] = {2, 5, 3, 9, 1};

   affiche(T, 5);
```

void, pas void

```
double puissance(double a, int e)
{
   int i;
   double b = 1;

   for (i=0; i<e; i++) b = b*a;
   return b;
}
```

```
void affiche(int tab[], int n)
{
   int i;

   for (i=0; i<n; i++)
      printf("%d ",tab[i]);
   printf("\n");
}
```

Fonction de type **double**

⇒ **return b**

Fonction de type **void**

⇒ pas de **return**

```
   double y;

   y = puissance(3.01, 5);
```

```
   int T[5] = {2, 5, 3, 9, 1};

   affiche(T, 5);
```

⇒ affectation **y =** ...

⇒ appel sans affectation

Passage par valeur

La fonction ne manipule que des **copies** des variables passées en arguments

⇒ ne connaît que les valeurs
de ces arguments,
pas leur emplacement mémoire

⇒ ne peut pas modifier le **contenu**
d'une variable passée en argument

seules les copies locales sont modifiées,
mais détruites à la sortie.

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
   -o incrementation
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
  -o incrementation
 >incrementation 10
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
  -o incrementation
 >incrementation 10
```

Résultat ??

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
  -o incrementation
 >incrementation 10
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
   -o incrementation
  >incrementation 10
```

```
b = 11
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III

```
/* incrementation.c */

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

int incrementation(int x)
{
   x = x+1;
   return x;
}
```

```
>gcc incrementation.c
   -o incrementation
  >incrementation 10
```

```
b = 11 , a = 10
```

```
int main(int argc, char *argv[])
{
   int a,b;
   a = atoi(argv[1]);

   b = incrementation(a);
   print("b = %d, a = %d\n",b,a);
   return 0;
}
```

Exemple III – analyse

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← 10

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← 10

Appel de **incrementation(a)**

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

x variable locale à la fonction

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

x variable locale à la fonction

x ← **a** (soit **x** ← **10**)

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

x variable locale à la fonction

x ← **a** (soit **x** ← **10**)

x ← **x+1** (soit **x** ← **11**)

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

- **x** variable locale à la fonction
- **x** ← **a** (soit **x** ← **10**)
- **x** ← **x+1** (soit **x** ← **11**)
- **return x** (soit **11**)

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

- **x** variable locale à la fonction
- **x** ← **a** (soit **x** ← **10**)
- **x** ← **x+1** (soit **x** ← **11**)
- **return x** (soit **11**)

b ← **11**

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Exemple III – analyse

```
a = 10;
b = incrementation(a);
```

a ← **10**

Appel de **incrementation(a)**

x variable locale à la fonction

x ← **a** (soit **x** ← **10**) ⇒ **a** pas modifiée

x ← **x+1** (soit **x** ← **11**)

return x (soit 11)

b ← **11**

```
int incrementation(int x)
{
   x = x+1;
   return x;
}
```

Variables globales et locales

Les variables déclarées en tête d'un module sont visibles dans toutes les fonctions du module

Les variables déclarées dans une fonction ne sont visibles qu'au cours de la fonction (et ont une durée de vie limitée à l'exécution de la fonction)

Rq : Une variable locale « masque » une variable globale du même nom

Intérêt des fonctions

On a déjà vu les avantages des fonctions
utilisations multiples et réutilisation ultérieure d'une partie de code

Mais surtout,

ça aide à écrire un programme qui marche !

- plus une fonction est courte,
  plus elle a de chances d'être correcte

⇒ découper le programme en petites fonctions de moins de 20 lignes (une fenêtre)

Utilisation de fonctions

Fichiers de headers (*.**h**)

```
int addition(int a, int b)
double puissance(double, int)
```

⇒ premier argument, un double
second argument, un entier
retourne le premier puissance le second

L'en-tête suffit à décrire une fonction

cf. **man**

Exemple : `atoi`

```
man atoi
int atoi(const char *str)
```

⇒ prend une chaîne de caractères

(éventuellement une constante,
et non une variable)

et retourne un entier