Langage C

TD 4: Pointeurs et Fonctions

Julien SEINTURIER
Associate Professor

Université de Toulon
julien.seinturier@univ-tln.fr
http://www.seinturier.fr/teaching

1. Fonctions

Exercice 1.1. Expliquer ce qu’est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de **factoriser** un certain nombre de lignes de code en un endroit du programme.

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de **factoriser** un certain nombre de lignes de code en un endroit du programme.

Une fois la fonction **définie**, elle peut être utilisée autant de fois que nécessaire à partir de n'importe quel autre point du programme.

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de **factoriser** un certain nombre de lignes de code en un endroit du programme.

Une fois la fonction **définie**, elle peut être utilisée autant de fois que nécessaire à partir de n'importe quel autre point du programme.

Une fonction peut accepter des **paramètres** et produire une **valeur de retour** que le programme peut récupérer.

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de **factoriser** un certain nombre de lignes de code en un endroit du programme.

Une fois la fonction **définie**, elle peut être utilisée autant de fois que nécessaire à partir de n'importe quel autre point du programme.

Une fonction peut accepter des **paramètres** et produire une **valeur de retour** que le programme peut récupérer.

Syntaxe:
```
type nom(type1 param1, … , typeN paramN){
  [intruction;]*
  [return valeur;]
}
```

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction permet de **factoriser** un certain nombre de lignes de code en un endroit du programme.

Une fois la fonction **définie**, elle peut être utilisée autant de fois que nécessaire à partir de n'importe quel autre point du programme.

Une fonction peut accepter des **paramètres** et produire une **valeur de retour** que le programme peut récupérer.

Syntaxe:
```
type nom(type1 param1, … , typeN paramN){
  [intruction;]*
  [return valeur;]
}
```

`valeur` doit être de type `type`

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction peut **ne pas prendre de paramètres**

Syntaxe:

```
type nom(){
  [intruction;]*
  [return valeur;]
}
```

```
type nom(void){
  [intruction;]*
  [return valeur;]
}
```

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction peut **ne pas prendre de paramètres**

Syntaxe:

```
type nom(){
  [intruction;]*
  [return valeur;]
}
```

```
type nom(void){
  [intruction;]*
  [return valeur;]
}
```

Une fonction peut **ne pas retourner de valeur**

Syntaxe:

```
void nom(type param){
  [intruction;]*
}
```

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Exemples:

```c
int add(int a, int b){
  return a + b;
}
```

Paramètres et retour

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Exemples:

```c
int add(int a, int b){
  return a + b;
}
```

Paramètres et retour

```c
float read(){
  int a;
  scanf(%d, &a);
  return a;
}
```

Pas de paramètre et retour

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Exemples:

```c
int add(int a, int b){
  return a + b;
}
```

Paramètres et retour

```c
float read(){
  int a;
  scanf(%d, &a);
  return a;
}
```

Pas de paramètre et retour

```c
void display(int a, int b){
  printf("%d - %d", a, b);
}
```

Paramètres et pas de retour

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Exemples:

```c
int add(int a, int b){
  return a + b;
}
```

Paramètres et **retour**

```c
float read(){
  int a;
  scanf(%d, &a);
  return a;
}
```

Pas de paramètre **et** **retour**

```c
void display(int a, int b){
  printf("%d - %d", a, b);
}
```

Paramètres et pas de retour

```c
void hello(){
  printf("hello");
}
```

Pas de paramètre **et** pas de retour

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

Une fonction doit être **déclarée avant son utilisation**.

De façon standard, on déclare les fonctions **avant la fonction principale `main`**

```
int add(int a, int b){
  return a + b;
}

int main(){
  return 0;
}
```

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

L'appel d'une la fonction se fait en utilisant **son nom** et en lui **passant les paramètres** nécessaires

```c
int add(int a, int b){
  return a + b;
}

int main(){
  int n1 = add(1, 2);
  return 0;
}
```

Exercice 1.1. Expliquer ce qu'est une fonction en langage C. Comment une fonction est intégrée à un programme ?

L'appel d'une la fonction se fait en utilisant **son nom** et en lui **passant les paramètres** nécessaires

```
int add(int a, int b){
  return a + b;
}

int main(){
  int n1 = add(1, 2);
  return 0;
}
```

Lors de l'exécution, l'appel à la fonction **est remplacé** par **la valeur de retour** de celle-ci

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```
int main(){
  int a,b;
  a=f1(0);
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0);
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Déclaration des variables a et b

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0);
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Appel à `f1` avec la valeur `0` pour le paramètre `i`

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0);
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f1 retourne la valeur 0+1, c'est à dire 1

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f1 retourne la valeur 0+1, c'est à dire 1
a prend la valeur 1

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Appel à f2 avec la valeur 1 pour le paramètre i

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1);
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f2 retourne la valeur 1++, c'est à dire 1

➡ Voir TD 2, ex 1.3.
(`i++` retourne la valeur de `i` avant le ++)

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f2 retourne la valeur 1++, c'est à dire 1
b prend la valeur 1

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Affiche sur la console:

```
1 : a=0, b=1
```

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Appel à f3 avec la valeur de a (1) pour le paramètre i

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Exécution du `printf` de f3 qui affiche:

`f3 : 0`

Car `i = 1` et donc `i == 0` vaut `0`

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a);
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f3 retourne i, c'est-à-dire 1

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f3 retourne i, c'est-à-dire 1
a reçoit la valeur 1

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Appel à f4 avec la valeur de a (1) pour le paramètre i

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Exécution du `printf` de f4 qui affiche:

`f4 : 0`

Car `i = 0` affecte `0` à `i` et le retoure

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a);
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f4 retourne i, c'est-à-dire 0

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a); // 0
  printf("2 : a=%d, b=%d\n",a,b);
}
```

f4 retourne i, c'est-à-dire 0
a reçoit la valeur 0

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```c
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```c
int main(){
  int a,b;
  a=f1(0); // 1
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a); // 0
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Affiche sur la console:

```
2 : a=1, b=1
```

Exercice 1.2. Analyser le programme suivant (donner la séquence des affichages produits) :

```
#include <stdio.h>
int f1(int i){return i+1;}
int f2(int i){return i++;}
int f3(int i) {
  printf("f3 : %d\n",i==0);
  return i;
}
int f4(int i) {
  printf("f4 : %d\n",i=0);
  return i;
}
```

```
int main(){
  int a,b;
  a=f1(0);
  b=f2(1); // 1
  printf("1 : a=%d, b=%d\n",a,b);
  a=f3(a); // 1
  b=f4(a); // 0
  printf("2 : a=%d, b=%d\n",a,b);
}
```

Affichage complet:

```
1 : a=0, b=1
f3 : 0
f4 : 0
2 : a=1, b=1
```

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

En langage C, les paramètres d'une fonction sont passés **par valeur**

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

En langage C, les paramètres d'une fonction sont passés **par valeur**

La fonction reçoit **une copie des valeurs originales** de chaque paramètre.

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

En langage C, les paramètres d'une fonction sont passés **par valeur**

La fonction reçoit **une copie des valeurs originales** de chaque paramètre.

Au niveau de la mémoire, lors de l'appel d'une fonction, une pile indépendante est **créée** puis est **détruite** à la fin de la fonction **sans modifier le programme principal**

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

En langage C, les paramètres d'une fonction sont passés **par valeur**

La fonction reçoit **une copie des valeurs originales** de chaque paramètre.

Au niveau de la mémoire, lors de l'appel d'une fonction, une pile indépendante est **créée** puis est **détruite** à la fin de la fonction **sans modifier le programme principal**

Ce fonctionnement permet de sécuriser l'exécution mais **empêche aussi une fonction de modifier ses paramètres** et de répercuter ces modifications dans le programme principal

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Déclare et initialise a et b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Affiche 1 2

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Appel de `increment` qui reçoit les valeurs copiées de a et b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
   a = a + 1;
   b = b + 1;
}

void main(){
   int a = 1, b = 2;
   printf("%d %d", a, b);
   increment(a, b);
   printf("%d %d", a, b);
}
```

Modification des valeurs copiées,
sans effet dans le programme principal

Appel de `increment` qui reçoit les valeurs copiées de a et b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Affiche 1 2

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

a	b
1	2

Création de deux nouvelles variables pour a et b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

a	b
1	2

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

a	b
1	2

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
   a = a + 1;
   b = b + 1;
}

void main(){
   int a = 1, b = 2;
   printf("%d %d", a, b);
   increment(a, b);
   printf("%d %d", a, b);
}
```

Mémoire

a	b	a	b
1	2	1	2

Création de deux nouvelles variables pour a et b

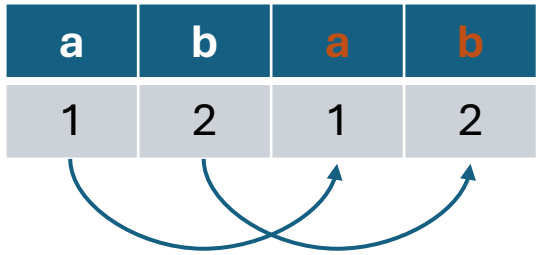
Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
   a = a + 1;
   b = b + 1;
}

void main(){
   int a = 1, b = 2;
   printf("%d %d", a, b);
   increment(a, b);
   printf("%d %d", a, b);
}
```

Mémoire



Création de deux nouvelles variables pour a et b

Copie des valeurs de a vers a et b vers b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
   a = a + 1;
   b = b + 1;
}

void main(){
   int a = 1, b = 2;
   printf("%d %d", a, b);
   increment(a, b);
   printf("%d %d", a, b);
}
```

Mémoire

a	b	a	b
1	2	2	3

Modification des valeurs de a et b

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

a	b	a	b
1	2	2	3

Fin d'appel de la fonction

Destruction de sa mémoire

Exercice 1.3. Comment sont passés les paramètres du programme principal à une fonction ? Quelles limitations cela engendre-t-il ?

Exemple:

```
void increment(int a, int b){
  a = a + 1;
  b = b + 1;
}

void main(){
  int a = 1, b = 2;
  printf("%d %d", a, b);
  increment(a, b);
  printf("%d %d", a, b);
}
```

Mémoire

a	b
1	2

Mémoire libérée

Affichage des valeurs de a et b

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  }
}
```

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  } else if (x == y){
    return 0;
  }
}
```

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  } else if (x == y){
    return 0;
  } else {
    return -1;
  }
}
```

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  } else if (x == y){
    return 0;
  } else {
    return -1;
  }
}
```

Exercice 1.4. Ecrire une fonction `int compare(int x, int y)` qui pour deux nombres x et y passés en paramètre renvoie 1 si x est le plus grand nombre, -1 si y est le plus grand nombre et 0 si les deux nombres sont égaux.

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  } else if (x == y){
    return 0;
  } else {
    return -1;
  }
}
```

```
int compare(int x, int y) {
  if (x > y){
    return 1;
  }

  if (x == y){
    return 0;
  }

  return -1;
}
```

Version allégée

Exercice 1.5. Ecrire une fonction `float valeurAbsolue(float x)` qui renvoie la valeur absolue de x.

Exercice 1.5. Ecrire une fonction `float valeurAbsolue(float x)` qui renvoie la valeur absolue de x.

```
float valeurAbsolue(float x) {
  if (x < 0) {
    return -x;
  }

  return x;
}
```

Exercice 1.5. Ecrire une fonction `float valeurAbsolue(float x)` qui renvoie la valeur absolue de x.

```
float valeurAbsolue(float x) {
  if (x < 0) {
    return -x;
  }

  return x;
}
```

```
float valeurAbsolue(float x) {
  return x<0 ? -x : x;
}
```

Version ?

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de `10`. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de `10`. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  return result;
}
```

Définition de la fonction

Au sens strict de l'algorithmique une fonction contient **un seul return**

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  return result;
}
```

Calcul des distances au carré

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  return result;
}
```

Calcul des distances au carré

Soit d_1 et d_2 deux distances, si $\mathbf{d_1 < d_2}$ alors $\mathbf{d_1^2 < d_2^2}$ car la fonction racine carrée est continue et strictement croissante

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  return result;
}
```

Si la distance pour x est la plus petite, le résultat est x

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  return result;
}
```

Si la distance pour y est la plus petite, le résultat est y

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

Si les distances sont égales, on renvoie le plus grand nombre

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
}
```

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;
}
```

Suppression des multiplications

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;

    if (distx < disty) return x;
}
```

Si x est plus proche, on le retourne

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;

    if (distx < disty) return x;
    if (disty < distx) return y;
}
```

Si y est plus proche, on le retourne

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;

    if (distx < disty) return x;
    if (disty < distx) return y;
    if (x > y) return x;
}
```

Si les deux nombres sont aussi proches de 10 et que x est le plus grand on le retourne

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;

    if (distx < disty) return x;
    if (disty < distx) return y;
    if (x > y) return x;

    return y;
}
```

Aucun return précédent n’a été effectué, y est le plus grand

Version optimisée

Exercice 1.6. Ecrire une fonction `int lePlusProcheDe10(int x, int y)` qui renvoie le nombre x ou y qui est le plus proche de 10. En cas de distance égale, la fonction renvoie le plus grand des deux nombres.

```
int lePlusProcheDe10(int x, int y) {
  int result;
  int distx = (10-x)*(10-x);
  int disty = (10-y)*(10-y);
  if (distx < disty)
    result = x;
  else if (disty < distx)
    result = y;
  else if (x > y)
    result = x;
  else
    result = y;
  return result;
}
```

```
int lePlusProcheDe10(int x, int y) {
    int distx = (x > 10) ? x - 10 : 10-x;
    int disty = (y > 10) ? y – 10 : 10-y;

    if (distx < disty) return x;
    if (disty < distx) return y;
    if (x > y) return x;

    return y;
}
```

Version optimisée

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 0;
  return s;
}
```

Corps de la fonction

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 0;
  for(int i = 1; i <= n ; i++)
    s += i*i;
  return s;
}
```

Calcul de la somme par itération

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 0;
  for(int i = 1; i <= n ; i++)
    s += i*i;
  return s;
}
```

Calcul de la somme par itération

`Pas besoin de bloc (1 instruction)`

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 0;
  for(int i = 1; i <= n ; i++)
    s += i*i;
  return s;
}
```

Version `for`

```
int sommeDesCarres(int n) {
  int s = 0, i = 1;
  while(i <= n){
    s += i*i;
    i = i + 1;
  }
  return s;
}
```

Utilisation de bloc obligatoire (2 instructions)

Version `while`

Exercice 1.7. Ecrire une fonction `int sommeDesCarres(int n)` qui renvoie la somme des n premiers entiers au carré.

```
int sommeDesCarres(int n) {
  int s = 0;
  for(int i = 1; i <= n ; i++)
    s += i*i;
  return s;
}
```

Version for

```
int sommeDesCarres(int n) {
  int s = 0, i = 1;
  while(i <= n){
    s += i*i;
    i = i + 1;
  }
  return s;
}
```

Version while

```
int sommeDesCarres(int n) {
  int s = 0, i = 1;
  do {
    s += i*i;
    i = i + 1;
  } while(i <= n)
  return s;
}
```

Version do ... while

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 < i \leq j \leq n$.

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

Pour n = 4

$$s = \underbrace{1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4}_{\substack{i = 1 \\ 1 \leq j \leq 4}}$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

Pour n = 4

$$s = \underbrace{1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4}_{\substack{i=1 \\ 1 \leq j \leq 4}} + \underbrace{2 \times 2 + 2 \times 3 + 2 \times 4}_{\substack{i=2 \\ 2 \leq j \leq 4}}$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

Pour n = 4

$$s = \underbrace{1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4}_{\substack{i=1 \\ 1 \leq j \leq 4}} + \underbrace{2 \times 2 + 2 \times 3 + 2 \times 4}_{\substack{i=2 \\ 2 \leq j \leq 4}} + \underbrace{3 \times 3 + 3 \times 4}_{\substack{i=3 \\ 3 \leq j \leq 4}}$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

Pour n = 4

$$s = \underbrace{1 \times 1 + 1 \times 2 + 1 \times 3 + 1 \times 4}_{\substack{i=1 \\ 1 \leq j \leq 4}} + \underbrace{2 \times 2 + 2 \times 3 + 2 \times 4}_{\substack{i=2 \\ 2 \leq j \leq 4}} + \underbrace{3 \times 3 + 3 \times 4}_{\substack{i=3 \\ 3 \leq j \leq 4}} + \underbrace{4 \times 4}_{\substack{i=4 \\ 4 \leq j \leq 4}}$$

Le calcul peut s'écrire $s = \sum_{i=1}^{n} \sum_{j=i}^{n} i \times j$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

Pour n = 4

$$s = \underbrace{1\times1 + 1\times2 + 1\times3 + 1\times4}_{i=1,\ 1\leq j\leq 4} + \underbrace{2\times2 + 2\times3 + 2\times4}_{i=2,\ 2\leq j\leq 4} + \underbrace{3\times3 + 3\times4}_{i=3,\ 3\leq j\leq 4} + \underbrace{4\times4}_{i=4,\ 4\leq j\leq 4}$$

Le calcul peut s'écrire $s = \sum_{i=1}^{n}\sum_{j=i}^{n} i \times j$

$\sum_{i=1}^{n}$ → `for` | `while` | `do … while`

Itération

Réflexe du développeur

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){
  int s = 0;
  return s;
}
```

Corps de la fonction

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){
  int s = 0;
  for(int i = 1; i <= n; i++)


  return s;
}
```

$$s = \sum_{i=1}^{n} \sum_{j=i}^{n} i \times j$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){
  int s = 0;
  for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++)


  return s;
}
```

$$s = \sum_{i=1}^{n} \sum_{j=i}^{n} i \times j$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){
  int s = 0;
  for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++)
      s += i*j;
  return s;
}
```

$$s = \sum_{i=1}^{n} \sum_{j=i}^{n} i \times j$$

Exercice 1.8. Ecrire une fonction `int sommeDesProduits(int n)` qui renvoie la somme des produits $i \times j, 1 \leq i \leq j \leq n$.

```
int sommeDesProduits(int n){
  int s = 0;
  for(int i = 1; i <= n; i++)
    for(int j = i; j <= n; j++)
      s += i*j;
  return s;
}
```

```
int sommeDesProduits(int n){
  int s = 0;
  for(int i = 1; i <= n; i++){
    for(int j = i; j <= n; j++){
      s += i*j;
    }
  }
  return s;
}
```

Version avec blocs

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Division euclidienne:

$$n = qd + r$$

où:

- $n \in \mathbb{N}$ est le numérateur
- $d \in \mathbb{N}^*$est le diviseur
- $q \in \mathbb{N}$ est le quotient
- $r \in \mathbb{N}$ est le reste avec $r < d$

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Division euclidienne:

$$n = qd + r = \sum_{i=1}^{q} d + r$$

où:

- $n \in \mathbb{N}$ est le numérateur
- $d \in \mathbb{N}^*$est le diviseur
- $q \in \mathbb{N}$ est le quotient
- $r \in \mathbb{N}$ est le reste avec $r < d$

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Division euclidienne:

$$n = qd + r = \sum_{i=1}^{q} d + r$$

$$n = \sum_{i=1}^{q} d + r$$

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Division euclidienne:

$$n = qd + r = \sum_{i=1}^{q} d + r$$

$$n = \sum_{i=1}^{q} d + r$$

$$n - \sum_{i=1}^{q} d = r$$

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

Division euclidienne:

$$n = qd + r = \sum_{i=1}^{q} d + r$$

$$n = \sum_{i=1}^{q} d + r$$

$$n - \sum_{i=1}^{q} d = r$$

Le quotient q est le **nombre de fois** que l'on peut **retrancher** le diviseur d à n.

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

```
int reste(unsigned int x, unsigned int y){
  unsigned int q = 0;
  unsigned int r = 0;
  return r;
}
```

Corps de la fonction

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

```
int reste(unsigned int x, unsigned int y){
  unsigned int q = 0;
  unsigned int r = 0;
  if (y == 0)
    return 0;
  return r;
}
```

Cas de division par 0

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

```
int reste(unsigned int x, unsigned int y){
  unsigned int q = 0;
  unsigned int r = 0;
  if (y == 0)
    return 0;
  r = x;
  while(r >= y){
    r = r - y;
    q += 1;
  }
  return r;
}
```

Calcul du quotient et du reste

Exercice 1.9. Ecrire une fonction `unsigned int reste(unsigned int x, unsigned int y)` qui renvoie le reste de la division entière de x par y (utiliser une boucle while, l'opérateur modulo % est interdit).

```
int reste(unsigned int x, unsigned int y){
  unsigned int q = 0;
  unsigned int r = 0;
  if (y == 0)
    return 0;
  r = x;
  while(r >= y){
    r = r - y;
    q += 1;
  }
  return r;
}
```

Exercice 1.10. Ecrire une fonction `void echange(int a, int b)` qui échange les valeurs de a et b. Que donne alors le programme suivant :

```
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

Exercice 1.10. Ecrire une fonction `void echange(int a, int b)` qui échange les valeurs de a et b. Que donne alors le programme suivant :

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int a, int b){
    int c = a;
    a = b;
    b = c;
}
```

Exercice 1.10. Ecrire une fonction `void echange(int a, int b)` qui échange les valeurs de a et b. Que donne alors le programme suivant :

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int a, int b){
    int c = a;
    a = b;
    b = c;
}
```

Le programme affiche: 1 2

Exercice 1.10. Ecrire une fonction `void echange(int a, int b)` qui échange les valeurs de a et b. Que donne alors le programme suivant :

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int a, int b){
    int c = a;
    a = b;
    b = c;
}
```

Le programme affiche: 1 2

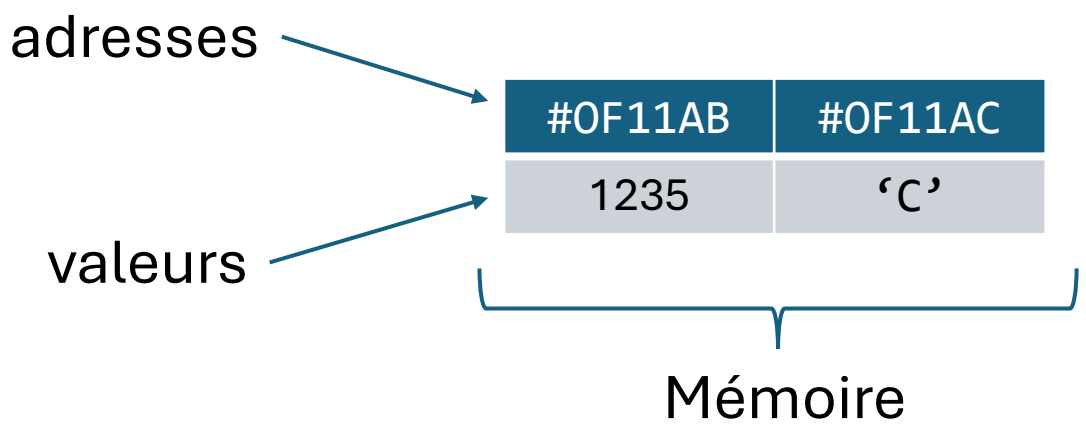
Les variables échangées par la fonction ne sont que des **copies** des variables du programme principal (voir ex. 1.3.)

2. Pointeurs simples

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

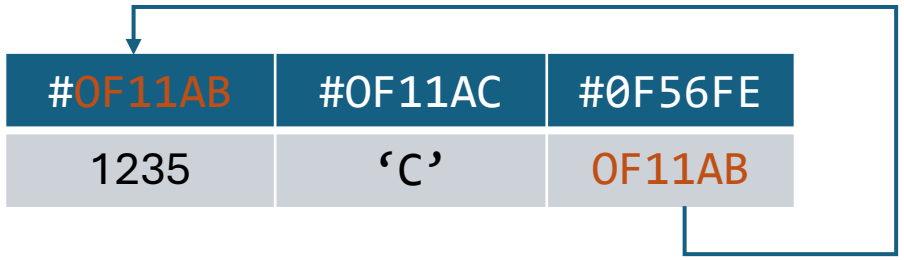
Une **variable** est un espace dans la mémoire identifié par une **adresse** et réservé à une **valeur** d'un certain type.



Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Une **variable** est un espace dans la mémoire identifié par une **adresse** et réservé à une **valeur** d'un certain type.

Un **pointeur** est une **variable** dont la **valeur** est l'**adresse** d'un espace mémoire (par exemple celui une autre variable).



Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour déclarer un **pointeur** vers une **variable** d'un certain type on ajoute * après la déclaration de type.

Type	Type de pointeur
`char`	`char*`
`short`	`short*`
`int`	`int*`
`float`	`float*`
`double`	`double*`

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Une **variable** est un espace dans la mémoire identifié par une **adresse** et réservé à une valeur d'un certain type.

Un **pointeur** est une **variable** dont la **valeur** est l'**adresse** d'un espace mémoire (par exemple celui une autre variable).

Pour déclarer un **pointeur** vers une **variable** d'un certain type on ajoute * après la déclaration de type.

Type	Type de pointeur
`char`	`char*`
`short`	`short*`
`int`	`int*`
`float`	`float*`
`double`	`double*`

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

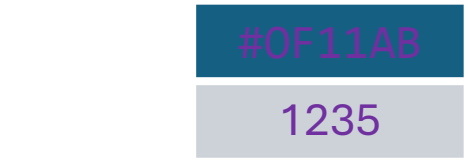
Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
}
```

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
}
```

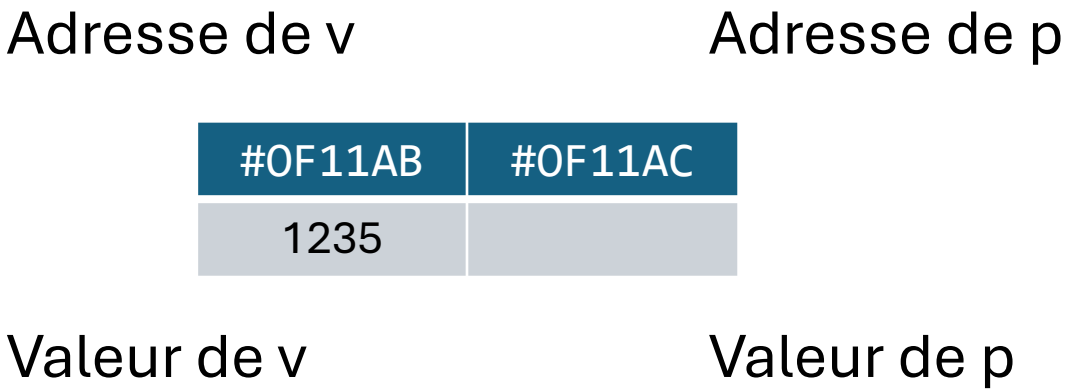


Création d'un espace mémoire pour stocker v

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
  int* p;
}
```

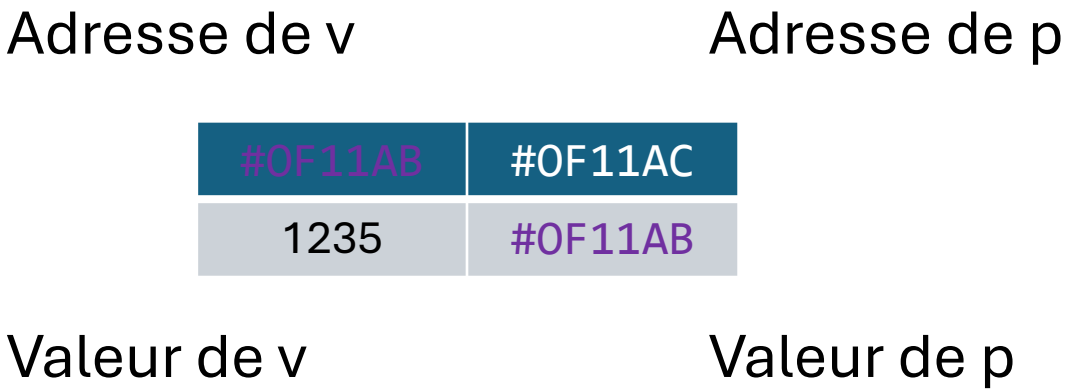


Création d'un espace mémoire pour stocker p

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
  int* p;
  p = &v;
}
```

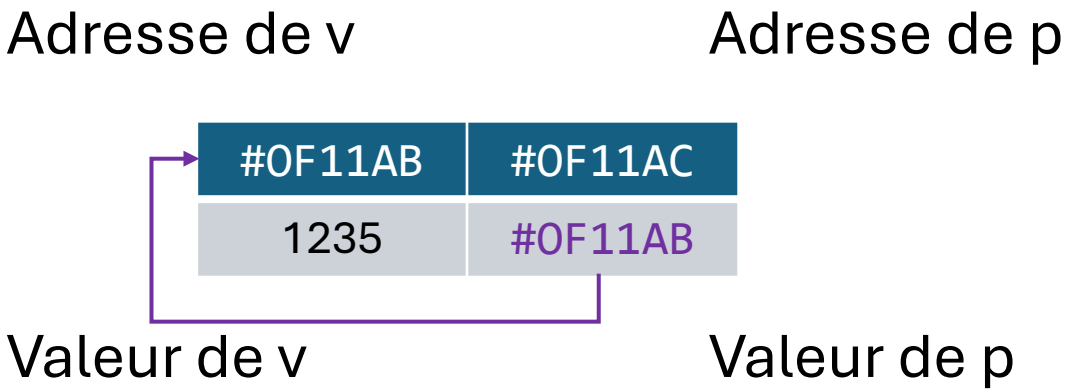


p reçoit comme valeur l'adresse de v

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

```
#include <stdio.h>
int main(){
   int v =1235;
   int* p;
   p = &v;
}
```



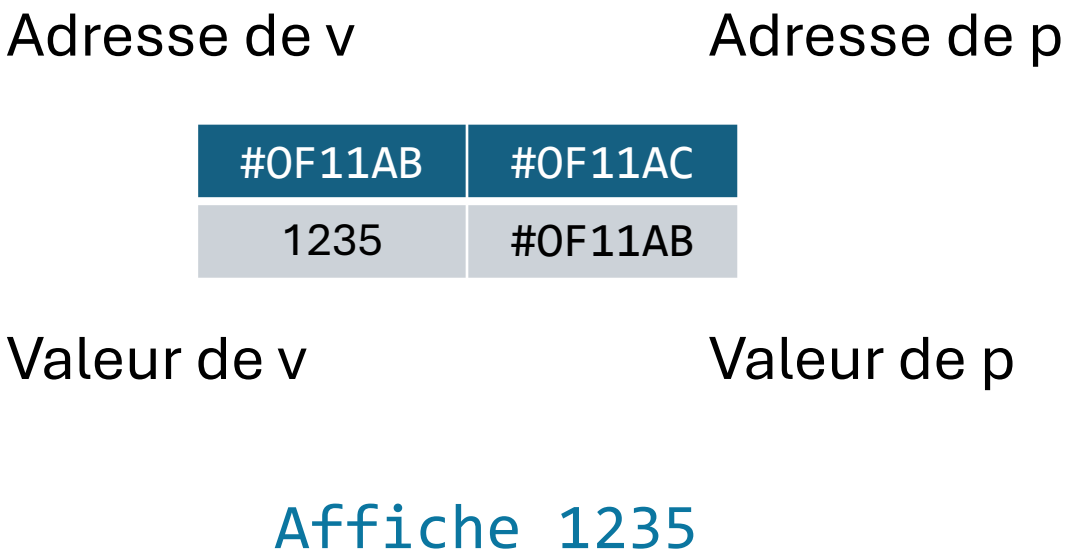
p pointe vers l'adresse de v

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

Pour **obtenir une valeur** depuis un pointeur, on ajoute une étoile * devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
  int* p;
  p = &v;
  printf("%d", *p);
}
```

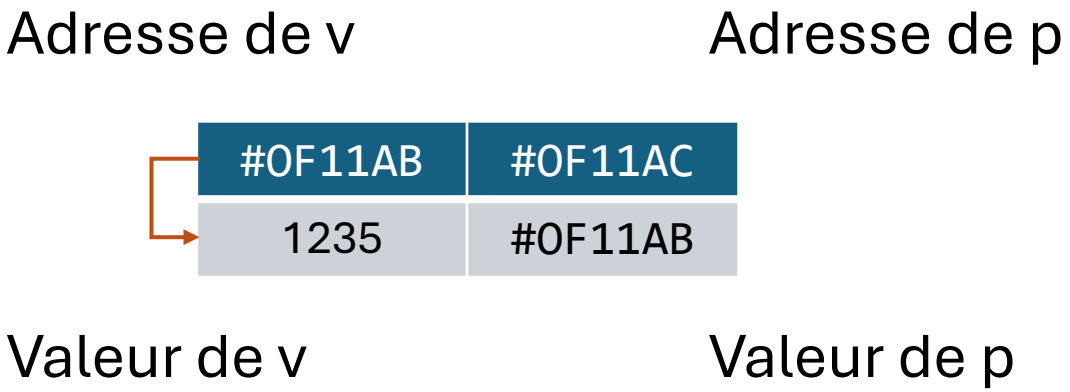


Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

Pour **obtenir une valeur** depuis un pointeur, on ajoute une étoile * devant son nom

```
#include <stdio.h>
int main(){
  int v =1235;
  int* p;
  p = &v;
  printf("%d", *p);
}
```

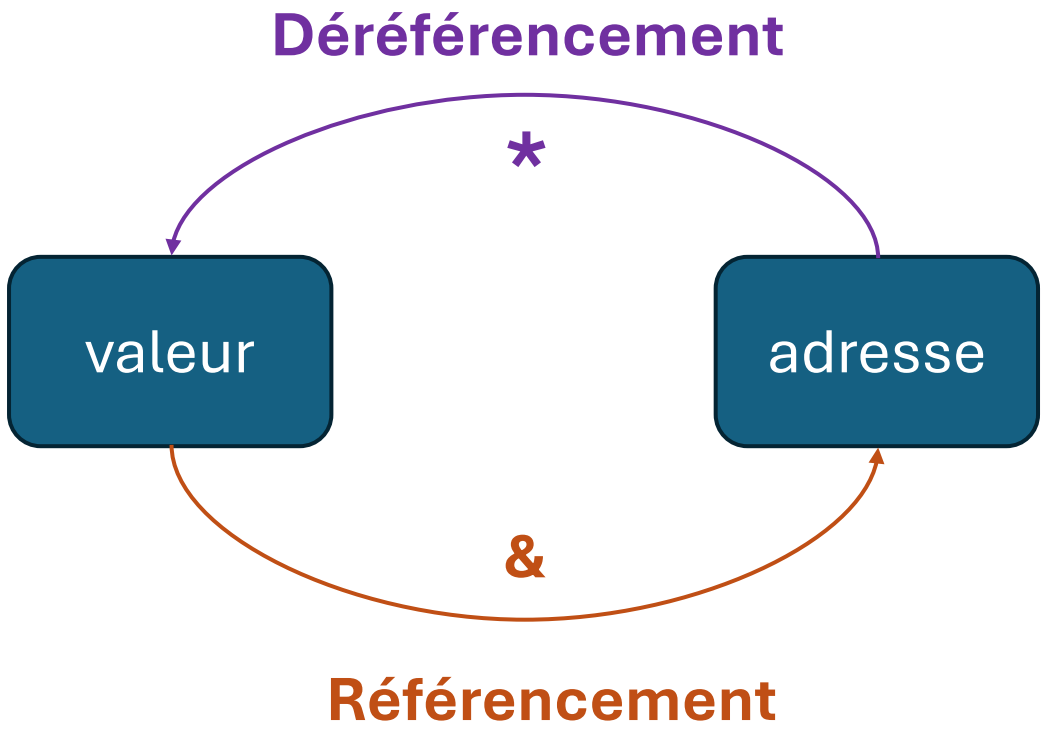


* Permet de récupérer une valeur à partir de son adresse

Exercice 2.1. Décrire ce que sont une variable et un pointeur au sens informatique du terme. Faire un schéma pour expliquer leur fonctionnement. Expliquer ensuite comment passer de l'un à l'autre.

Pour **obtenir l'adresse** d'une variable, on ajoute une esperluette **&** devant son nom

Pour **obtenir une valeur** depuis un pointeur, on ajoute une étoile * devant son nom



Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1] int i = 10;  [2] int* p;  [3] p = &i;  [4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1] int i = 10;  [2] int* p;  [3] p = &i;  [4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

`int i = 10;` Déclare une variable entière `i` et l'initialise avec la valeur `10`

Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1] int i = 10;  [2] int* p;  [3] p = &i;  [4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

`int i = 10;` Déclare une variable entière `i` et l'initialise avec la valeur `10`

`int* p;` Déclare un pointeur `p` vers un entier

Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1] int i = 10;  [2] int* p;  [3] p = &i;  [4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

`int i = 10;` Déclare une variable entière `i` et l'initialise avec la valeur `10`

`int* p;` Déclare un pointeur `p` vers un entier

`p = &i;` Affecte l'adresse de `i` à `p`

Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1]int i = 10;  [2]int* p;  [3]p = &i;  [4] printf("%p, %d \n", &i, i);
[5]printf("%p, %d \n", p, *p);
```

`int i = 10;` Déclare une variable entière `i` et l'initialise avec la valeur `10`

`int* p;` Déclare un pointeur `p` vers un entier

`p = &i;` Affecte l'adresse de `i` à `p`

`printf("%p, %d \n", &i, i);` Affiche l'adresse de `i` et sa valeur

Pour afficher un pointeur (ou une adresse), il faut utiliser `%p`

Exercice 2.2. Décrire ce que font les instructions suivantes :

```
[1] int i = 10;  [2] int* p;  [3] p = &i;  [4] printf("%p, %d \n", &i, i);
[5] printf("%p, %d \n", p, *p);
```

`int i = 10;` Déclare une variable entière `i` et l'initialise avec la valeur `10`

`int* p;` Déclare un pointeur `p` vers un entier

`p = &i;` Affecte l'adresse de `i` à `p`

`printf("%p, %d \n", &i, i);` Affiche l'adresse de `i` et sa valeur

`printf("%p, %d \n", p, *p);` Affiche la valeur de `p` (l'adresse de `i`) et la valeur pointée par `p` (la valeur de `i`)

Pour afficher un pointeur (ou une adresse), il faut utiliser %p

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

```
int i = 0;
```

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

```
int i = 0;
i = 12;
```

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

```
int i = 0;
i = 12;
int* p;
```

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

```
int i = 0;
i = 12;
int* p;
p = &i;
```

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

```
int i = 0;
i = 12;
int* p;
p = &i;
scanf("%d", p);
```

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0	`int i = 0;`
Déclarer un pointeur vers un entier p	`i = 12;`
Affecter à i une valeur arbitraire	`int* p;`
Faire pointer p vers l'adresse de i	`p = &i;`
Saisir une nouvelle valeur pour la variable pointée par p	`scanf("%d", p);`
Afficher la valeur de i	`printf("%d", i);`

Exercice 2.3. Ecrire un programme qui réalise les actions suivantes :

Déclarer un entier i initialisé à 0

Déclarer un pointeur vers un entier p

Affecter à i une valeur arbitraire

Faire pointer p vers l'adresse de i

Saisir une nouvelle valeur pour la variable pointée par p

Afficher la valeur de i

```c
int main(){
   int i = 0;
   i = 12;
   int* p;
   p = &i;
   scanf("%d", p);
   printf("%d", i);
   return 0;
}
```

Exercice 2.4. Trouver l'erreur dans cet extrait de programme :

```
int x=5;
int *p = &x;
p = 9;
```

Exercice 2.4. Trouver l'erreur dans cet extrait de programme :

```
int x=5;
int *p = &x;
p = 9;
```

L'instruction `p = 9` soulève une **erreur de compilation**.

La variable `p` étant un pointeur vers un entier (`int*`), elle ne peut recevoir de valeur entière (`int`).

Les adresses mémoires **ne sont pas des entiers** au sens du langage C.

Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

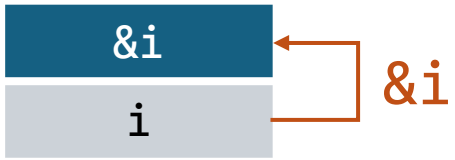
```
p = &*&i;

i = *&*&j;
```

Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;

i = *&*&j;
```



Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

`p = &i`

```
i = *&*&j;
```

Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

`p = &i`

```
i = *&*&j;
```

Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

`p = &i`

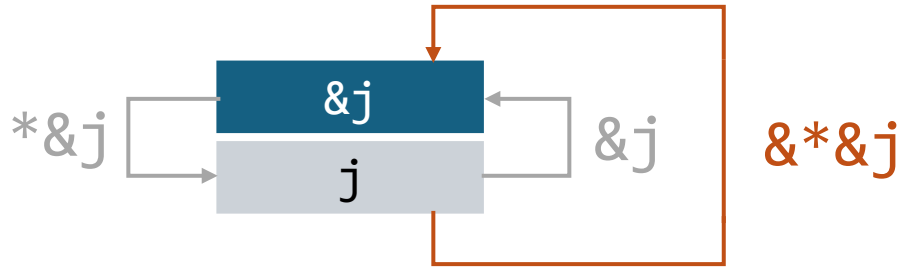
```
i = *&*&j;
```

Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

```
p = &i
```

```
i = *&*&j;
```

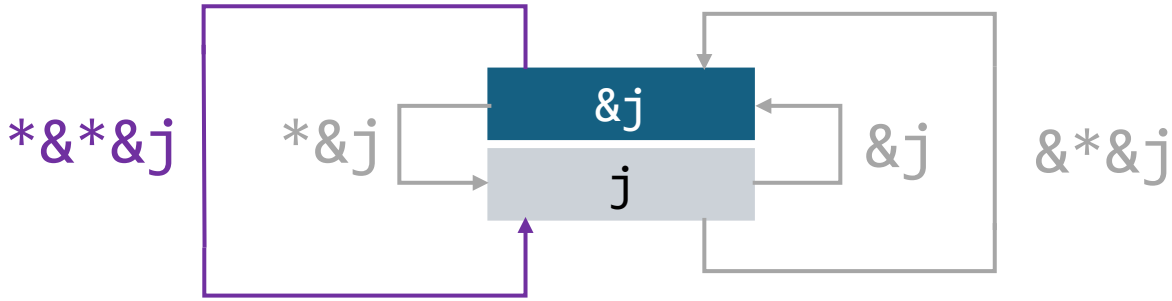


Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

`p = &i`

```
i = *&*&j;
```



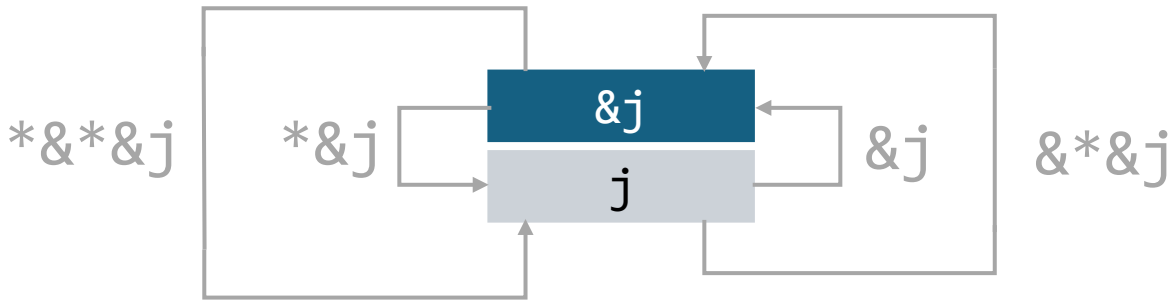
Exercice 2.5. Simplifier les expressions suivantes où `p` est de type `int*` et `i` de type `int` :

```
p = &*&i;
```

```
p = &i
```

```
i = *&*&j;
```

```
i = j;
```



Exercice 2.6. Refaire l'exercice 1.10. en écrivant une fonction d'échange permettant réellement d'échanger les 2 valeurs. Modifier le programme principal pour que l'échange soit effectif.

```
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```
void echange(int a, int b){
    int c = a;
    a = b;
    b = c;
}
```

Exercice 2.6. Refaire l'exercice 1.10. en écrivant une fonction d'échange permettant réellement d'échanger les 2 valeurs. Modifier le programme principal pour que l'échange soit effectif.

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(a, b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int* a, int* b){
    int c = a;
    a = b;
    b = c;
}
```

Echange prends maintenant en paramètre des pointeurs vers les variables à echanger

Exercice 2.6. Refaire l'exercice 1.10. en écrivant une fonction d'échange permettant réellement d'échanger les 2 valeurs. Modifier le programme principal pour que l'échange soit effectif.

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(&a, &b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int* a, int* b){
    int c = a;
    a = b;
    b = c;
}
```

echange prends maintenant en paramètre des **pointeurs** vers les variables à echanger

L'appel à echange se fait alors à partir des **adresses** des variables à echanger

Exercice 2.6. Refaire l'exercice 1.10. en écrivant une fonction d'échange permettant réellement d'échanger les 2 valeurs. Modifier le programme principal pour que l'échange soit effectif.

```c
#include <stdio.h>
int main(){
  int a=1,b=2;
  echange(&a, &b);
  printf("a=%d, b=%d\n",a,b);
}
```

```c
void echange(int* a, int* b){
    int c = *a;
    *a = *b;
    *b = *c;
}
```

Les variables de echange étant des pointeurs, l'accès à leur valeur se fait par **déréférencement**