Initiation au C
cours n°2

Antoine Miné

École normale supérieure

22 février 2007

Plan du cours

- compléments sur les déclarations et expressions,
- les fonctions.

Compléments sur les déclarations et expressions

Déclarations multiples

Déclaration simultanée de plusieurs variables de même type.

Syntaxe

```
type var1=expr1, var2=expr2, ..., varN=exprN;
```

- les variables sont séparées par des virgules ,
- chacune est optionnellement initialisée grâce à =
- l'ordre de création et d'évaluation des initialiseurs est inconnu.

Exemples

```
int x,y=2,z;      /* seul y est initialisé */
int X=3, Y=2*X;   /* invalide */
```

Affectations combinées

Opération et affectation combinées *op*= :

Opérateurs combinées

```
var += exp ;   équivalent à   var = var + (expr) ;
var -= exp ;        "         var = var - (expr) ;
var *= exp ;        "         var = var * (expr) ;
var /= exp ;        "         var = var / (expr) ;
var %= exp ;        "         var = var % (expr) ;
```

Attention :

- pas d'espace entre l'opérateur *op* et le égal =,
- `x *= y+1` est équivalent à `x = x*(y+1)`
  et pas `x = x*y+1`.

Résultat retourné par une affectation

Comme +, -, etc. toute affectation retourne une valeur :

- `var = expr ;` retourne la valeur de `expr`,
- `var` *op*`= expr ;` retourne la nouvelle valeur de `var`,
- l'affectation associe à droite.

Exemples

`x = (y = 12*z+3);`	met la valeur de `12*z+3` dans `x` et `y`
`x = y = 12*z+3;`	" "
`x = (y+=2)+1;`	met `y+2` dans `y` et `y+3` dans `x`

Incrémentation et décrémentation

Opérateurs

- `var++ ;` équivalent à `var = var + 1 ;`
- `var-- ;` équivalent à `var = var - 1 ;`

- pas d'espace entre les deux symboles,
- espace possible entre la variable et l'opérateur.

L'opérateur peut se placer avant ou après la variable :

- même effet sur `var` mais valeur retournée différente,
- `var++` et `var--` renvoient la valeur de `var` **avant** l'affectation,
- `++var` et `--var` renvoient la valeur de `var` **après** l'affectation.

Incrémentation et décrémentation

Utilisation courante

```
int x = 0;
while (x<10) {
    printf("%i\n",x);
    x++;
}
```

Exemples compliqués

```
x = 12;
y = x++;          /* ici, y=12 et x=13 */
x = --y;          /* ici, y=11 et x=11 */
z = x++ + ++y;    /* ici, z=23 */
```

Question : que signifie C++ ?

Conflits d'effets de bord

Une unique instruction (terminée par ;) peut :

- lire plusieurs variables,
- modifier plusieurs variables.

L'ordre de ces opérations est indéfini !

Conséquence

Une instruction ne doit pas :

- modifier deux fois la même variable,
- lire et modifier une même variable.

Conflits d'effets de bord

Exemples incorrects

```
x = (y=2) + (y=3);   /* invalide: y modifié deux fois */
x = (x=2) + 1;       /* invalide: x modifié deux fois */
x = (y=2) + y;       /* invalide: y lu et modifié */
y = x++ + x;         /* invalide: x lu et modifié */
```

exception : dans `var=expr`, `var` peut apparaître dans `expr`

```
x = x+1; /* correct */
```

Conclusion

Se limiter à une affectation par instruction.

Le “type” booléen

booléen = valeur de vérité : vrai ou faux.

Le C n’a pas de type booléen dédié :

- tout entier ou flottant non nul signifie vrai,
- 0 ou 0.0 signifie faux.

Les comparaisons et opérateurs booléens `==` `!=` `>` `>=` `<` `<=` `&&` `||` renvoient toujours un entier :

- 0 pour faux,
- 1 pour vrai.

Exemples

Raccourcis classiques

`if (x) ...` est équivalent à `if (x !=0) ...`

`if ( !x) ...` est équivalent à `if (x==0) ...`

Exemples plus complexes :

- `expr1 && expr2` est équivalent à `(expr1 !=0)*(expr2 !=0)`,
- `expr !=0` est équivalent à `! !expr`.

Opérateurs booléens et court-circuits

`expr1 && expr2` et `expr1 || expr2` :

- évaluent d'abord `expr1`,
- évaluent `expr2` uniquement si nécessaire :
  - pour `&&`, si `expr1` est fausse, `expr2` n'est pas évaluée,
  - pour `||`, si `expr1` est vraie, `expr2` n'est pas évaluée.

Application

```
if ( x!=0 && y/x>100 ) ...
```

ne provoque jamais de division par zéro.

Opérateur ternaire

Opérateur d'alternative ternaire :

Syntaxe

```
expr1 ? expr2 : expr3
```

Effet :

- évalue `expr1`,
- si `expr1` est vrai, évalue et renvoie `expr2`,
- si `expr1` est fausse, évalue et renvoie `expr3`,
- un seul parmi `expr2` et `expr3` est évalué.

Exemple

```
x = y? 10/y : 99999;
```

évite encore une division par zéro.

Intermède

Que fait le programme suivant ?

je_compte_mal.c

```
#include <stdio.h>
int main()
{
  int x = 1;
  while (x<100) {
    if (x=2) printf("je n'aime pas 2\n");
    else     printf("%i\n",x);
    x++;
  }
  return(0);
}
```

Solution

L'erreur

```
while (x<100) {
  if (x=2) printf("je n'aime pas 2\n");
  else     printf("%i\n",x);
```

On a confondu = et == !

Solution

L'erreur

```
while (x<100) {
  if (x=2) printf("je n'aime pas 2\n");
  else     printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans `x`,

Solution

L'erreur

```
while (x<100) {
  if (x=2) printf("je n'aime pas 2\n");
  else     printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans `x`,
- comme 2 est vrai, on affiche toujours `je n'aime pas 2`,
- la branche else n'est jamais prise,

Solution

L'erreur

```
while (x<100) {
  if (x=2) printf("je n'aime pas 2\n");
  else     printf("%i\n",x);
```

On a confondu = et == !

À chaque itération :

- on met 2 dans `x`,
- comme 2 est vrai, on affiche toujours `je n'aime pas 2`,
- la branche else n'est jamais prise,
- en plus, on boucle indéfiniment !

Les fonctions

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Nom de la fonction.
N'importe quel identificateur non déjà utilisé convient.

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Nom des arguments formels de la fonction,
séparés par des virgules ,

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Type de chaque argument formel.

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Type de la valeur renvoyée par la fonction,
ou void si la fonction ne renvoie pas de valeur.

Définition de fonctions

Fonction : bloc d'instruction, nommé, avec arguments et valeur de retour.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN)
{
  instructions
}
```

Instructions à exécuter à chaque appel.
Les accolades `{ }` sont obligatoires.

Exemple de fonction sans valeur de retour

Table de multiplication

```
#include <stdio.h>
void table(int x,int max)
{
  int y=1;
  while (y<max) {
    printf("%i x %i = %i\n",x,y,x*y);
    y++;
  }
}
int main()
{
  table(3,10);
  table(4,10);
  return 0;
}
```

Appel de fonction sans valeur de retour

Appel de fonction : instruction terminée par ;

Syntaxe

```
nom(expr1,expr2,...,exprN) ;
```

Effet :

- évalue les expressions `expr1` à `exprN`,
- crée des variables locales `arg1` à `argN`,
- initialise chaque `argi` à la valeur de `expri` (conversion implicite éventuelle),
- exécute les `instructions` de la fonction,
- détruit les variables locales `arg1` à `argN`.

Position des définitions

Les définitions de fonctions apparaissent dans l'"espace global" avec :

- les directive `#include`,
- les déclarations de variables globales.

Chaque fonction définit un "espace local" contenant :

- des déclarations de variables locales,
- des instructions,
- des sous blocs (espace locaux imbriqués).

Attention

Il n'existe pas de 'fonctions locales' en C

Déclarations en avance

Attention

Toute fonction doit être déclarée avant d'être appelée.

On peut déclarer une fonction sans la définir, par un prototype.

Syntaxe

```
type nom(type1 arg1, type2 arg2, ..., typeN argN);
```

- le 'corps' de la fonction n'est pas fourni et remplacé par ;
- il faut quand même préciser tous les types,
- le prototype se place dans l'"espace global".

Exemple de déclaration en avance

Table de multiplication

```
#include <stdio.h>

void table(int x,int max);

int main()
{
  int i = 2;
  while (i<10) {
    table(i,10);
    i++;
  }
  return 0;
}
```

```
void table(int x,int max)
{
  int y=1;
  while (y<max) {
    printf("%i x %i = %i\n",
           x,y,x*y);
    y++;
  }
}
```

Variables accessibles

Une fonction peut accéder :

- aux variables globales déjà déclarées,
- aux fonctions déjà déclarées,
- à ses variables locales,
- à ses arguments formels.

Attention

Une fonction ne peut pas accéder aux variables locales d'une autre fonction !

Vie et mort des variables locales

Chaque appel de fonction crée son lot de variables locales.

Variables locales de l'**appelé**

- crées lors de l'appel,
- détruites quand la fonction se termine.

Variables locales de l'**appelant**

- masquées durant l'appel,
- mais gardent leur valeur.

$\Rightarrow$ les valeurs sont perdues entre deux appels successifs,
$\Rightarrow$ une fonction qui s'appelle elle-même a plusieurs copies de ses variables.

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

Résultat

```
2 3 4
```

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire

main

l. 13
x = 0

Résultat

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire

main

l. 13
x = 0

compte

l. 4
x = 4
y = 2

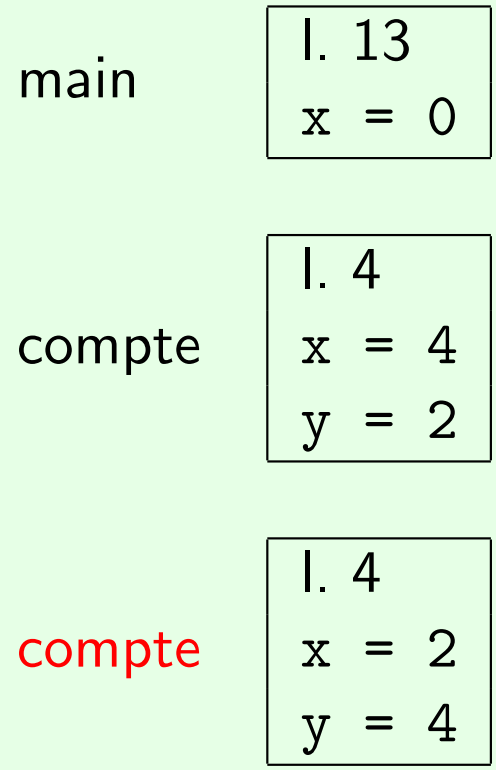
Résultat

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire



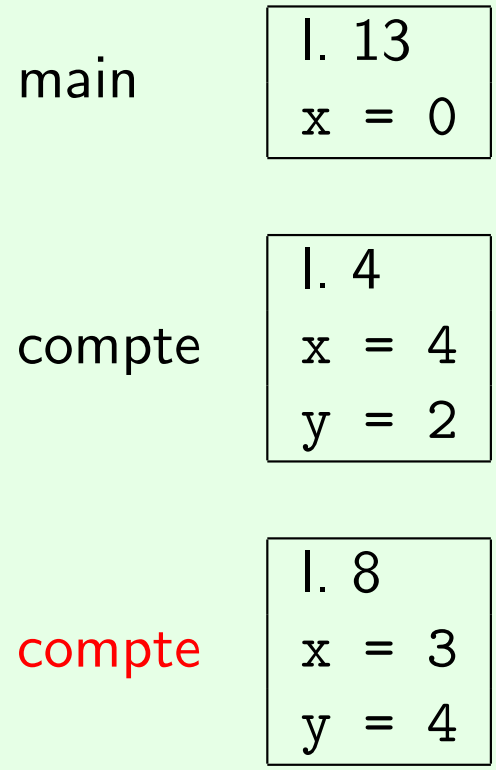
Résultat

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire



Résultat

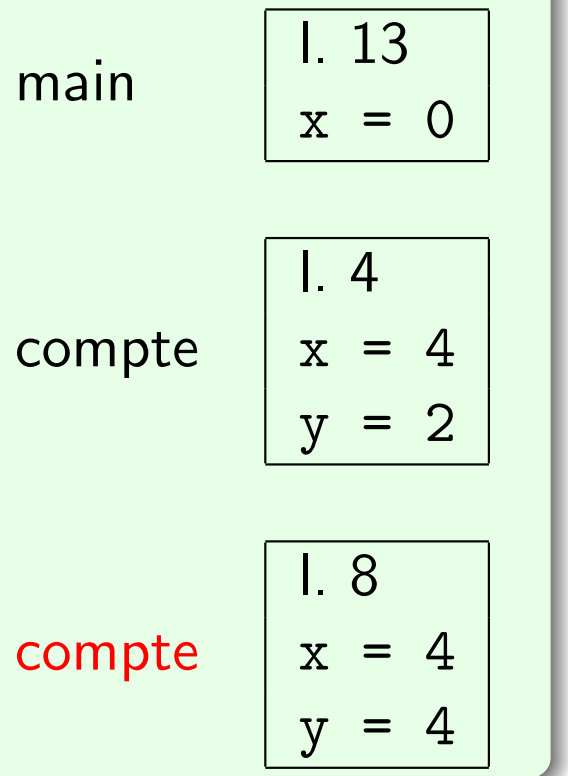
2

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire



Résultat

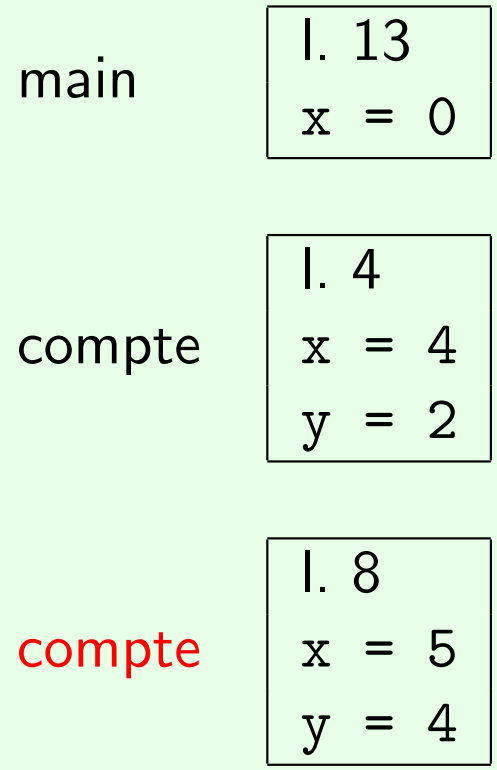
2 3

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire



Résultat

2 3 4

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire

main

l. 13
x = 0

compte

l. 5
x = 4
y = 2

Résultat

2 3 4

Exemple

Programme

```
1  #include <stdio.h>
2  void compte(int x,int y)
3  {
4    if (x>y) compte(y,x);
5    while (x<=y) {
6      printf("%i ",x);
7      x++;
8    }
9  }
10 int main()
11 {
12   int x=0;
13   compte(x+4,x+2);
14 }
```

État de la mémoire

main

l. 14
x = 0

Résultat

2 3 4

Fonctions avec valeur de retour

Si `type` n'est pas `void`, la fonction doit retourner une valeur.

Syntaxe

return expr ;

Effet :

- évalue `expr`,
- quitte immédiatement la fonction,
- renvoie la valeur de `expr` à l'appelant.

Exemple

Nombres premiers

```
int est_premier(int n)
{
  int i = 2;
  while (i<n) {
    if ( !(n%i) ) return 0;
    i++;
  }
  return 1;
}
```

Renvoie 1 si et seulement si `n` est premier.

Utilisation de la valeur de retour

On peut appeler la fonction dans n'importe quelle expression.

Affichage des nombre premiers

```
int i=2;
while (i<1000) {
  if (est_premier(i)) printf("%i\n",i);
  i++;
}
```

Comptage des nombre premiers

```
int i=2, n=0;
while (i<1000)
  n = n + est_premier(i++);
printf("%i\n",n);
```

(On peut aussi ignorer la valeur de retour. . .)

Notes sur `return`

Notes :

- il peut y avoir plusieurs `return`,
- dans une fonction `void`,
  `return ;` quitte immédiatement la fonction,
- dans une fonction non `void`,
  on doit toujours sortir de la fonction par `return expr ;`

Conflits d'effets de bord

Attention

L'ordre d'évaluation dans une expression est indéterminé.

Cela peut être gênant si l'appel de fonction a un effet :

- affichage sur l'écran,
- modification d'une variable globale,
- modification d'un fichier, etc.

Exemple dangereux

```
x = printf("a") + printf("b");
```

Affiche soit `ab` soit `ba`.

Conflits d'effets de bord

Exemple dangereux

```
int a = 0;

int compte()
{ return ++a; }

int main()
{
  printf("%i\n", compte() + a);
  return 0;
}
```

Cet exemple illustre aussi l'intérêt des variables globales pour :

- partager des variables entre fonctions,
- garder une valeur entre deux appels.