I3 – Algorithmique et programmation
Introduction à la programmation en langage C

Cours n°4
Introduction au langage C

Camille Coti
camille.coti@iutv.univ-paris13.fr

IUT de Villetaneuse, département R&T

2011 – 2012

Bloc d'instructions

Les blocs sont délimités par des accolades : { et }

```
1 #include <stdio.h>
2 #include <stdlib.h>
3
4 int main(){
5    printf( "Hello world\n" );
6    return EXIT_SUCCESS;
7 }
```

- Une variable déclarée dans un bloc **a une portée limitée à ce bloc**
- Les fonctions et les structures de contrôle utilisent des **blocs pour structurer le code**
- Ici : le bloc de la fonction `main` commence ligne 4 et termine ligne 7
  - La fonction `main` exécute les instructions contenues dans ce bloc

Affichage formaté à l'écran

Utilisation de la fonction **printf**

- Définie dans `stdio.h` : il faut indiquer `#include <stdio.h>` pour inclure sa définition

Syntaxe :

- On donne le format de la chaîne de caractères à afficher
- Si on affiche des variables : on les met ensuite

Exemple :

```
1    int i;
2    i = 0;
3    printf( "i vaut %d\n", i );
```

On indique le **type** de la variable à mettre dans la chaîne de caractères avec un %

- type int : %d ; long int : %ld
- type caractère : %c
- type flottant : %f (simple précision), %lf (double précision)

Caractères spéciaux : \n = retour chariot, \t : tabulation...

Saisie formatée au clavier

Utilisation de la fonction **scanf**

- Définie dans `stdio.h`

Syntaxe :

- On donne le format des variables lues (identique à celui utilisé par printf)
- On donne l'emplacement de ces variables en mémoire avec leur nom précédé de & : **adresse en mémoire de la variable**

Exemple :

```
1  int i;
2  printf( "Saisir i: \n");
3  scanf( "%d", &i );
```

Expressions booléennes

Comparaison de valeurs :

- Égalité : ==
  - Attention : DEUX signes égal
  - Exemple : a == b ; a == 2 (à éviter) ; 2 == a (recommandé)
- Non-égalité : !=
  - Exemple : a != b
- Ordre : < et >, >= et <=
  - Exemple : a > b ; a >= b

Opérateurs logiques :

- ET logique : &&
  - Exemple : (a != b) && (b == c)
- OU logique : ||
  - Exemple : (a != b) || (b == c)

Tests if

Syntaxe :

- `if` suivi de la condition à évaluer entre parenthèses
- bloc d'instructions à exécuter si la condition est validée

```
1    if( i == 0 ) {
2       printf( "i est nul\n" );
3    }
```

Alternative (facultative) :

- `else`
- bloc d'instructions à exécuter si la condition n'est pas validée

```
1    if( 0 == i ) {
2       printf( "i est nul\n" );
3    } else {
4       printf( "i vaut %d\n", i );
5    }
```

Tests if (suite)

Tests imbriqués : on peut mettre d'autres tests dans un bloc

```
1    if( i > 0 ) {
2       printf( "i est positif\n" );
3    } else {
4       if( 0 == i ) {
5          printf( "i est nul\n" );
6       } else {
7          printf( "i est négatif\n" );
8       }
9    }
```

Attention à l'indentation du code (décalage vers la droite)

- Le contenu d'un bloc est au même niveau d'indentation
- Chaque sous-bloc imbriqué est décalé d'un cran vers la droite
- Pas obligatoire d'un point de vue syntaxique mais aide grandement la lisibilité

Tests if (suite)

Utilisation d'opérateurs booléens pour combiner des expressions booléennes dans la condition :

```
1     if( ( lettre == 'a' )
2         || ( lettre == 'e' )
3         || ( lettre == 'i' )
4         || ( lettre == 'o' )
5         || ( lettre == 'u' )
6         || ( lettre == 'y' ) ) {
7       printf( "%c est une voyelle\n", lettre );
8     } else {
9       printf( "%c est une consonne\n", lettre );
10    }
```

Tests switch

Test sur une variable

- Plusieurs égalités testées successivement (les **case**)
- Définition d'une série d'instructions **pour chaque case**
- Fin de la série d'instructions avec **break**

```
1  int valeur;
2  valeur = 42;
3  switch( valeur ){
4    case 0:
5      printf( "c'est nul\n" );
6      break;
7    case 1:
8      printf( "valeur 1\n" );
9      break;
10   case -1:
11     printf( "valeur -1\n" );
12     break;
13   default:
14     printf( "autre valeur\n" );
15     break;
16 }
```

Tests switch

Test sur une variable

- Plusieurs égalités testées successivement (les **case**)
- Définition d'une série d'instructions **pour chaque case**
- Fin de la série d'instructions avec **break**

```
1  int valeur;
2  valeur = 42;
3  switch( valeur ){
4    case 0:
5      printf( "c'est nul\n" );
6      break;
7    case 1:
8      printf( "valeur 1\n" );
9      break;
10   case -1:
11     printf( "valeur -1\n" );
12     break;
13   default:
14     printf( "autre valeur\n" );
15     break;
16 }
```

Boucle for

Boucle **Pour**. Syntaxe :

```
1  for( initialisation du compteur ;
2       condition d'arrêt ;
3       modification du compteur ) {
4    bloc d'instructions
5  }
```

Le compteur doit être un entier ou un entier non signé.

Exemple :

```
1  int i;
2  for( i = 0 ; i < 3 ; i++ ) {
3    printf( "valeur de i : %d\n", i );
4  }
```

Boucle for (suite)

On peut définir n'importe quelle opération de modification du compteur.
Exemple avec un pas négatif :

```
1   int i;
2   for( i = 10 ; i > 0 ; i = i - 2 ) {
3     printf( "valeur de i : %d\n", i );
4   }
```

Affichage :

```
valeur de i : 10
valeur de i : 8
valeur de i : 6
valeur de i : 4
valeur de i : 2
```

Boucle while

Boucle **Tant que**. Syntaxe :

```
1   while( condition ) {
2     actions;
3   }
```

- Attention à la condition : si elle est toujours vérifiée, boucle infinie...
- Si la condition n'est jamais vérifiée on n'entre jamais dans la boucle : on teste la condition **puis on exécute le bloc si nécessaire**

Exemple :

```
1   float i;
2   i = 0;
3   while( i < 5 ) {
4     printf( "Valeur : %f\n", i );
5     i++;
6   }
```

Boucle do ... while

Boucle **Faire... tant que**. Syntaxe :

```
1  do {
2     actions;
3  } while( condition );
```

- Attention à la condition : si elle est toujours vérifiée, boucle infinie...
- Le bloc est exécuté **au moins une fois** : on teste la condition **après exécution du bloc**

Exemple :

```
1  i = 0;
2  do {
3     printf( "Valeur : %f\n", i );
4     i++;
5  } while( i < 5 );
```

Différence entre while et do ... while

Boucle **while** :

- On évalue la condition **avant** d'exécuter le bloc
- Si la condition est réalisée, on exécute le bloc
- Si la condition n'est jamais réalisée, on n'exécute pas le bloc du tout

Boucle **do ... while** :

- On évalue la condition **après** avoir exécuté le bloc
- Si la condition est réalisée, on ré-exécute le bloc
- Si la condition n'est jamais réalisée, on exécute une fois le bloc et on s'arrête là

Différence entre while et do ... while (suite)

Exemple :

```
1  int i;
2  i = 0;
3  while( i == 1 ) {
4     printf( "boucle while"\n );
5  }
6  do {
7     printf( "boucle do ... while"\n );
8  } while( i == 1 );
```

- La première condition n'est jamais réalisée. Donc on n'exécute pas la ligne 4
- On exécute la ligne 7 puis la condition ligne 8 n'est pas réalisée donc on s'arrête là

Affichage obtenu :

```
boucle do ... while
```

Les pointeurs

Définition

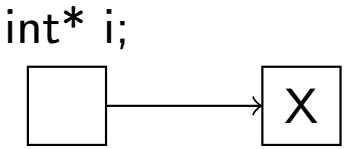
Un pointeur est une **adresse** vers une zone mémoire.

On déclare un pointeur avec le type de la zone pointée et une étoile :

```
type* nom;
```

Exemple :

```
int* i;
```



On désigne :

- Le pointeur (l'adresse) : le nom de la variable
  - Exemple : `i`
- La valeur pointée : * + nom de la variable
  - Exemple : `*i`

Attention : quand on a juste déclaré le pointeur, il ne pointe sur rien.

Adresse d'une variable

On obtient l'adresse d'une variable en utilisant l'opérateur `&`

Exemple :

```
1   int* i;
2   int k;
3   i = &k;
```

Ligne 3 : on affecte l'**adresse** de `k` dans le pointeur `i`.

Cas de `scanf` : on met la valeur lue à l'adresse de la variable

```
1   scanf( "%d", &i );
```

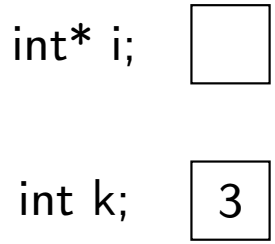
Adresse particulière : `NULL`

- Adresse inutilisable
- Utilisé pour gagner en sécurité (initialisations, effacement de valeurs...)
  - Exemple : `int* i = NULL;`
- Possibilité de tester
  - Exemple : `if( NULL == i )  /* ...  */`

Affectation d'une valeur dans un pointeur

Exemple : déclaration d'un pointeur

```
int* i;
int k = 3;
```



Affectation de l'adresse de `k` dans le pointeur `i`

```
i = &k;
```


Affichage :

```
printf("i = %d\n", *i);
```

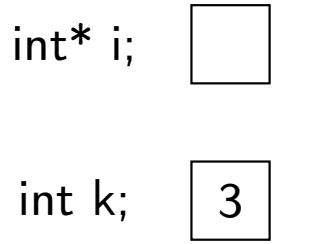
Résultat :

```
i = 3
```

Affectation d'une valeur dans un pointeur

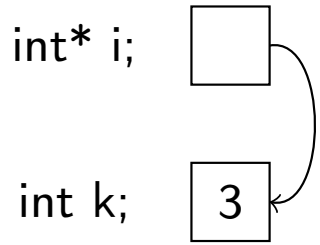
Exemple : déclaration d'un pointeur

```
int* i;
int k = 3;
```



Affectation de l'adresse de `k` dans le pointeur `i`

```
i = &k;
```



Affichage :

```
printf("i = %d\n", *i);
```

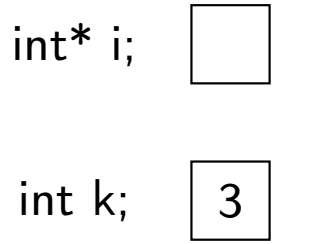
Résultat :

```
i = 3
```

Affectation d'une valeur dans un pointeur

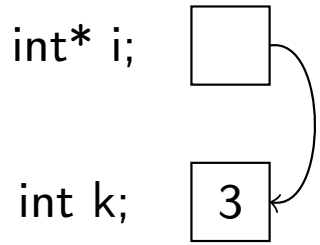
Exemple : déclaration d'un pointeur

```
int* i;
int k = 3;
```



Affectation de l'adresse de `k` dans le pointeur `i`

```
i = &k;
```



Affichage :

```
printf( "i = %d\n", *i );
```

Résultat :

```
i = 3
```

Affectation d'une valeur dans un pointeur

Autre exemple :

```
int* i;
int* j;
int k;

k = 3;
i = &k;
j = i;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 3 - j = 3`

Modification de la valeur de `k` :

```
k = 4;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 4 - j = 4`

Modification de la valeur pointée par `i` :

```
*i = 5;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 5 - j = 5`

Affectation d'une valeur dans un pointeur

Autre exemple :

```
int* i;
int* j;
int k;

k = 3;
i = &k;
j = i;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 3 - j = 3`

Modification de la valeur de `k` :

```
k = 4;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 4 - j = 4`

Modification de la valeur pointée par `i` :

```
*i = 5;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 5 - j = 5`

Affectation d'une valeur dans un pointeur

Autre exemple :

```
int* i;
int* j;
int k;

k = 3;
i = &k;
j = i;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 3 - j = 3`

Modification de la valeur de `k` :

```
k = 4;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 4 - j = 4`

Modification de la valeur pointée par `i` :

```
*i = 5;
```

Affichages :

```
printf("i = %d -- j = %d\n", *i, *j);
```

Donne : `i = 5 - j = 5`

Fonctions

Déclaration d'une fonction :

- Type retourné par la fonction
- Nom de la fonction
- Arguments et leurs types

Valeur retournée renvoyée avec le mot-clé `return` + la valeur retournée

Exemple :

```
1  int calculCarreHypothenuse( int cote1, int cote2 ){
2    int resultat;
3    /* implémentation de la fonction */
4    return resultat;
5  }
```

Procédures

Déclaration d'une procédure :

- Pas de type retourné : `void`
- Nom de la procédure
- Arguments et leurs types

Exemple :

```
1  void afficherResultat( int valeur ){
2    /* implémentation de la procédure */
3  }
```

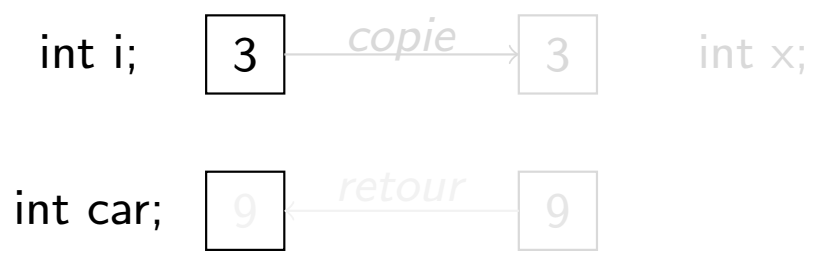
On sort de la procédure à la fin du bloc d'instruction où quand on rencontre le mot-clé `return`

Passage de paramètres

On passe les paramètres des fonctions et procédures en les appelant depuis la fonction principale

```c
int carre( int x ) {
    return (x*x);
}
int main(){
  int i = 3;
  int car;
  car = carre( i );
  printf( "%d\n", car );
  return EXIT_SUCCESS;
}
```

`i` est copié dans l'espace mémoire de la fonction `carre()`



Leur valeur est copiée dans l'espace mémoire de la fonction ou la procédure

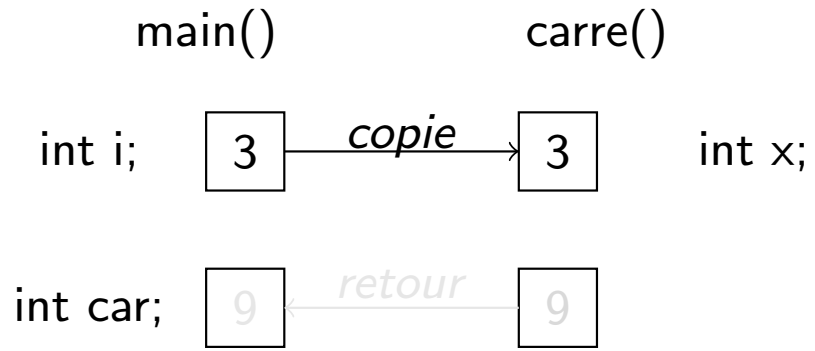
- Utilisation locale dans la fonction ou la procédure

Passage de paramètres

On passe les paramètres des fonctions et procédures en les appelant depuis la fonction principale

```c
int carre( int x ) {
    return (x*x);
}
int main(){
  int i = 3;
  int car;
  car = carre( i );
  printf( "%d\n", car );
  return EXIT_SUCCESS;
}
```

`i` est copié dans l'espace mémoire de la fonction `carre()`



Leur valeur est copiée dans l'espace mémoire de la fonction ou la procédure

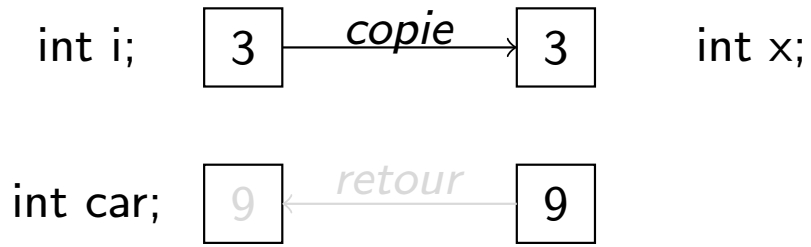
- Utilisation locale dans la fonction ou la procédure

Passage de paramètres

On passe les paramètres des fonctions et procédures en les appelant depuis la fonction principale

```c
int carre( int x ) {
    return (x*x);
}
int main(){
  int i = 3;
  int car;
  car = carre( i );
  printf( "%d\n", car );
  return EXIT_SUCCESS;
}
```

`i` est copié dans l'espace mémoire de la fonction `carre()`



Leur valeur est copiée dans l'espace mémoire de la fonction ou la procédure

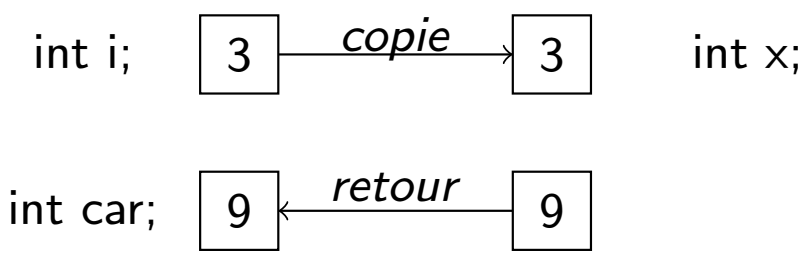
- Utilisation locale dans la fonction ou la procédure

Passage de paramètres

On passe les paramètres des fonctions et procédures en les appelant depuis la fonction principale

```c
int carre( int x ) {
    return (x*x);
}
int main(){
  int i = 3;
  int car;
  car = carre( i );
  printf( "%d\n", car );
  return EXIT_SUCCESS;
}
```

`i` est copié dans l'espace mémoire de la fonction `carre()`



Leur valeur est copiée dans l'espace mémoire de la fonction ou la procédure

- Utilisation locale dans la fonction ou la procédure

Passage de paramètres par valeur vs par pointeur

Attention au passage de paramètres par valeur !

- La valeur est **copiée** dans l'espace mémoire de la fonction
- Modifications locales dans la fonction : **non répercutées** sur l'appelant

Solution : passage de paramètres modifiables **par pointeur**

- Le pointeur n'est pas modifié
- La valeur pointée peut l'être

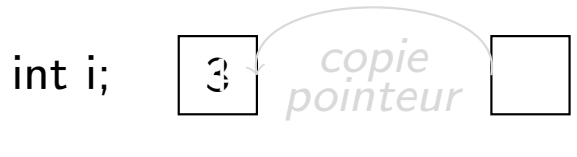
Passage de paramètres par pointeur

```
void incr( int* x ) {
  *x = *x + 1;
}
int main(){
  int i = 3;
  incr( &i );
  printf( "%d\n", i );
  return EXIT_SUCCESS;
}
```

`*i` est copié dans l'espace mémoire de la fonction `incr()`

- Il pointe toujours vers la valeur qui nous intéresse
- On peut modifier la valeur pointée

main() carre()



Utilisation

On doit passer les paramètres d'une fonction ou d'une procédure **par leur adresse** lorsque leur valeur est modifiée dans la fonction ou la procédure.

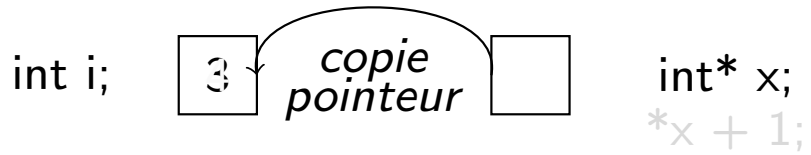
Passage de paramètres par pointeur

```c
void incr( int* x ) {
  *x = *x + 1;
}
int main(){
  int i = 3;
  incr( &i );
  printf( "%d\n", i );
  return EXIT_SUCCESS;
}
```

`*i` est copié dans l'espace mémoire de la fonction `incr()`

- Il pointe toujours vers la valeur qui nous intéresse
- On peut modifier la valeur pointée

main() carre()



Utilisation

On doit passer les paramètres d'une fonction ou d'une procédure **par leur adresse** lorsque leur valeur est modifiée dans la fonction ou la procédure.

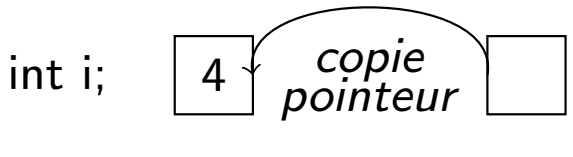
Passage de paramètres par pointeur

```c
void incr( int* x ) {
  *x = *x + 1;
}
int main(){
  int i = 3;
  incr( &i );
  printf( "%d\n", i );
  return EXIT_SUCCESS;
}
```

`*i` est copié dans l'espace mémoire de la fonction `incr()`

- Il pointe toujours vers la valeur qui nous intéresse
- On peut modifier la valeur pointée

main() carre()



Utilisation

On doit passer les paramètres d'une fonction ou d'une procédure **par leur adresse** lorsque leur valeur est modifiée dans la fonction ou la procédure.

Cas de la fonction main()

Paramètres de `main()` = arguments de la ligne de commande du programme

Paramètres récupérés sous forme d'un **tableau de chaînes de caractères**

- Rappel = chaîne de caractères = tableau de `char`

```
1 int main( int argc, char** argv ) {
2   printf("Nb d\'arguments de %s : %d\n", argv[0], argc);
3   return EXIT_SUCCESS;
4 }
```

Remarques :

- `argv[0]` = nom du programme appelant
- Les arguments sont des tableaux de `char` = nécessité de faire des conversions
  - `atoi()`, `atol()`...
- Le dernier élément (`argv[argc - 1]`) vaut `NULL`