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

Cours n°9
Rappels de C

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

IUT de Villetaneuse, département R&T

2011 – 2012

Types de variables définis par le langage C

Types de variables :

- Types entiers (integer) :
  - **int** : 4 octets, valeurs de -2^{31} à $2^{31}-1$
  - **short int** : 2 octets, de -32.768 à 32.767
  - **long int** : comme int
  - **long long int** : 8 octets, valeurs de -2^{63} à $2^{63}-1$
  - version non signées : pas de bit de signe
- Types caractères :
  - **char** : 1 octet, valeur ascii
  - **unsigned char** : caractère non signé
- Types réels :
  - **float** : nombre à virgule flottante simple précision (4 octets)
  - **double** : nombre à virgule flottante double précision (8 octets)
  - **long double** : 10 octets

Déclaration d'une variable :

```
1 int hauteur;
2 float largeur, aire;
3 long double diagonale;
```

Il faut déclarer les variables avant de les utiliser !!!

Les pointeurs

Définition

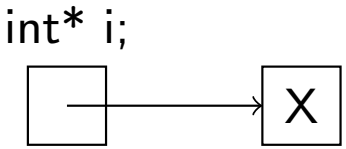
Un pointeur est une **adresse** vers une zone mémoire.

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

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`

Déclaration d'un tableau

En C :

- Déclaration d'un tableau
- Type des données contenues
- Taille fixe
- On ne peut pas dépasser les limites du tableau (SEGFAULT)

On peut parcourir le tableau avec une boucle `for` (par exemple) :

```
1   int i;
2   /* on a un tableau tab de N entiers */
3   for( i = 0 ; i < N ; i++ ) {
4       printf( "tab[%d] : %d\n", i, tab[i] );
5   }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1  int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
3
4  for( i = 0 ; i < TAILLETAB ; i++ ) {
5     tab[i] = 0;
6  }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1  int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
3
4  for( i = 0 ; i < TAILLETAB ; i++ ) {
5     tab[i] = 0;
6  }
```

Allocation statique

La taille du tableau est connue **à la compilation**

- Le compilateur réserve l'espace mémoire nécessaire

Le nombre d'éléments est donné entre crochets :

```
1  int tab[10];
```

Bonne pratique : utilisation d'une **constante de compilation**

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
```

À la compilation, le préprocesseur remplace `TAILLETAB` par sa valeur

- Code source plus lisible
- Plus facile à modifier ultérieurement
- Réutilisable dans le code

```
1  #define TAILLETAB 10
2  int tab[TAILLETAB];
3
4  for( i = 0 ; i < TAILLETAB ; i++ ) {
5    tab[i] = 0;
6  }
```

Déclaration statique et initialisation

Remplissage du tableau :

```
1  int tab[3] = { 0, 1, 2};
```

Le compilateur :

- Réserve l'emplacement en mémoire pour 3 cases
- Les remplit avec les valeurs données entre accolades (séparées par des virgules)

Autre possibilité :

- Déclaration sans la taille
- Initisalisation **tout de suite**
- Le compilateur compte le nombre d'éléments donnés pour l'initialisation et il réserve la taille nécessaire

```
1  int tab[] = { 0, 1, 2};
```

Application aux chaînes de caractères

Chaîne de caractères = tableau de caractères

Exemple : Coucou

C	o	u	c	o	u	\0

Le dernier élément est le caractère spécial \0

- Caractère de fin de chaîne de caractères
- Attention : pour une chaîne de N caractères, il faut donc un tableau de taille N+1

Déclaration d'une chaîne de caractères : `char* chaine;`

Opérations sur des chaines de caractères : la plupart déclarées dans string.h

- `#include <string.h>`

Allocation dynamique

Allocation d'une zone mémoire : fonction `malloc()`

- Prototype : void *malloc(size_t size);
- Définie dans stdlib.h : `#include <stdlib.h>`
- Retourne un pointeur vers l'espace mémoire alloué
- L'espace mémoire alloué est un tampon de mémoire contigüe

Remarques :

- Le type `size_t` est un entier
- `malloc()` retourne un pointeur de type non spécifié (`void*`)
  - Transtypage pour utiliser un pointeur du type désiré

Exemple :

```
1  #define TAILLETAB   10
2  int* i;
3  i = (int*) malloc( TAILLETAB * 4 );
```

Allocation dynamique

Allocation d'une zone mémoire : fonction `malloc()`

- Prototype : void *malloc(size_t size);
- Définie dans stdlib.h : `#include <stdlib.h>`
- Retourne un pointeur vers l'espace mémoire alloué
- L'espace mémoire alloué est un tampon de mémoire contigüe

Remarques :

- Le type `size_t` est un entier
- `malloc()` retourne un pointeur de type non spécifié (`void*`)
  - Transtypage pour utiliser un pointeur du type désiré

Exemple :

```
1  #define TAILLETAB    10
2  int* i;
3  i = (int*) malloc( TAILLETAB * 4 );
```

Utilisation de malloc()

Exemple précédent : non portable !

- Taille transmise en dur `TAILLETAB * 4`
- Et si un entier est codé sur autre chose que 4 octets ?

Fonction qui retourne la taille d'un élément :

- `size_t sizeof( type )`
- Retourne un entier (`int` ou `size_t`)
- Prend le nom du type en argument

```
1 #define TAILLETAB    10
2 int* tab;
3 tab = (int*) malloc( TAILLETAB * sizeof( int ) );
```

Attention : `malloc()` retourne **toujours** un pointeur utilisable (contrairement à ce que dit le man)

- Si le système n'a pas pu allouer la mémoire : `bus error` lors de l'utilisation (plantage sur SIGBUS)

Modification de la taille d'un espace alloué

Utilisation de la fonction **realloc()**

- Prototype : `void* realloc(void *ptr, size_t size);`
- Modification de la taille d'un tampon mémoire
- Concrètement : réallocation d'un nouvel espace mémoire, copie du contenu de l'ancien dans le nouveau (ce qui tient dans le nouveau)

Attention : le pointeur vers le tableau change

- On récupère un nouveau pointeur en retour de la fonction

```
1 #define TAILLETAB    10
2 int* tab;
3 int taille;
4 taille = TAILLETAB;
5 tab = (int*) malloc( taille * sizeof( int ) );
6 taille = taille * 2;
7 tab = (int*) realloc( tab, taille * sizeof( int ) );
```

Libération de l'espace mémoire

La mémoire allouée doit être **libérée** !!

- Utilisation de la procédure **free()**
- Prototype : `void free( void* ptr );`
- Le système libère le tampon mémoire pointé par le pointeur

```
1 #define TAILLETAB    10
2 int* tab;
3 tab = (int*) malloc( TAILLETAB * sizeof( int ) );
4 free( tab );
```

Après avoir libéré une zone mémoire, il est plus sûr de mettre le pointeur qui pointait dessus à la valeur `NULL`

- Éviter de l'utiliser ultérieurement par erreur

```
1 if( NULL != tab ) {
2   free( tab );
3   tab = NULL;
4 }
```

Définition d'une structure

Définition d'une structure de données contenant des éléments de types différents

```
1 struct maStruct {
2   type1 champ1;
3   type2 champ2;
4   ...
5 };
```

Syntaxe

- Utilisation du mot-clé **struct**
- Nom de la structure
- Définition des champs
- Attention au point-virgule final

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
```

Utilisation d'une structure

Déclaration d'un élément du type de la structure :

- `struct` + nom de la structure + nom de la variable

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
6
7 struct etudiant et1;
```

Possibilité d'utiliser des structures dans les champs d'une structure

```
1 struct date {
2   int jour;
3   int mois;
4   int annee;
5 };
```

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   struct date dateDeNaissance;
5 };
```

Utilisation d'une structure

Déclaration d'un élément du type de la structure :

- `struct` + nom de la structure + nom de la variable

Exemple :

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   int dateDeNaissace[3];
5 };
6
7 struct etudiant et1;
```

Possibilité d'utiliser des structures dans les champs d'une structure

```
1 struct date {
2   int jour;
3   int mois;
4   int annee;
5 };
```

```
1 struct etudiant {
2   char nom[256];
3   char prenom[256];
4   struct date dateDeNaissance;
5 };
```

Utilisation d'une structure (suite)

Accès aux champs de la structure :

- nom de la variable + point (.) + nom du champ

Exemple :

```
1 struct etudiant et1;
2 et1.dateDeNaissace = { 0, 0, 0 };
```

Cas d'un pointeur

- Si on déclare un pointeur vers un élément du type de cette structure
- Utilisation d'une flèche (->) pour accéder à ses champs

```
1 struct etudiant* et2;
2 et2 = (struct etudiant*) malloc( sizeof( etudiant ) );
3 et2->dateDeNaissace = { 0, 0, 0 };
```

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
```

Exemple : parcours d'un tableau

Tableau à une dimension :

```
1  #define TAILLE 16
2  int i;
3  float* tab;
4  /* allocation de memoire pour le tableau */
5  tab = (float*) malloc( TAILLE * sizeof( float ) );
6  /* Parcours du tableau */
7  for( i = 0 ; i < TAILLE ; i++ ) {
8      tab[i] = (float)i;
9  }
```

Ici :

- On alloue de la mémoire pour le tableau (ligne 5)
- On parcourt le tableau grâce à une boucle `for` (ligne 7)
- L'indice dans le tableau est la variable `i`
- On met dans le tableau l'indice `i`, converti en `float` (ligne 8)

Exemple : parcours d'un tableau

Tableau à deux dimensions :

```
1  #define HAUTEUR 16
2  #define LARGEUR  8
3  int i, j;
4  float** tab;
5  /* allocation de memoire pour le tableau */
6  tab = (float**) malloc( HAUTEUR * sizeof( float* ) );
7  for( i = 0 ; i < HAUTEUR ; i++ ) {
8       tab[i] = (float*) malloc( LARGEUR * sizeof( float ) );
9  }
10 /* Parcours du tableau */
11 for( i = 0 ; i < HAUTEUR ; i++ ) {
12     for( j = 0 ; j < LARGEUR ; j++ ) {
13         if( i == j ) {
14             tab[i][j] = 1.0;
15         } else {
16             tab[i][j] = 0.0;
17         }
18     }
19 }
```

- On alloue les lignes du tableau (ligne 6) puis les colonnes (ligne 8)
- On parcourt le tableau grâce à deux boucles `for` imbriquées (ligne 11 et ligne 12)

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  }
```

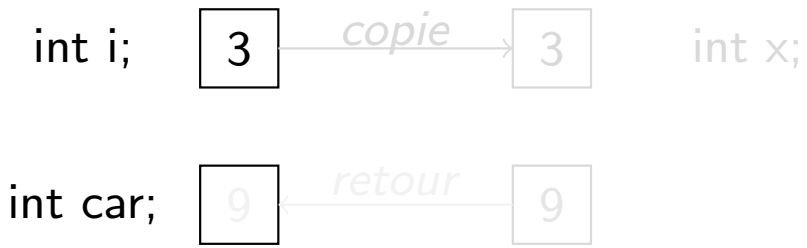
On sort de la procédure à la fin du bloc d'instruction où quand on rencontre le mot-clé `return`

Passage de paramètres

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

```
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

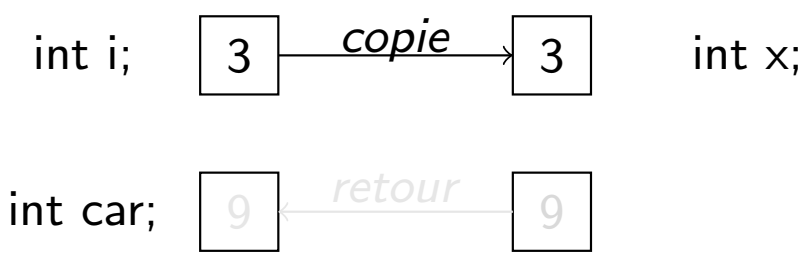
- Utilisation locale dans la fonction ou la procédure

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

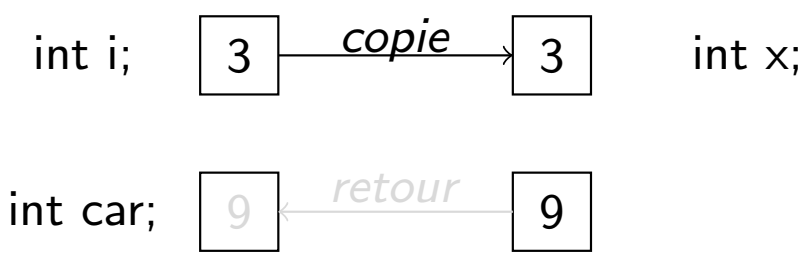
- Utilisation locale dans la fonction ou la procédure

Passage de paramètres

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

```
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

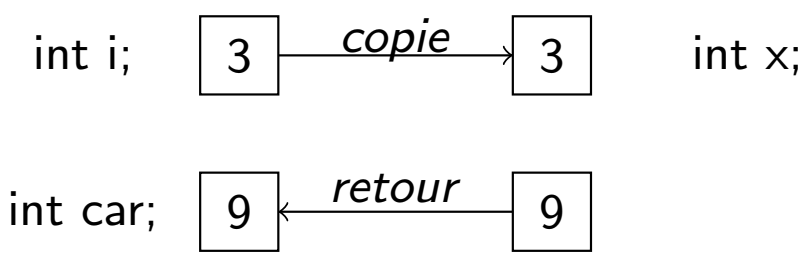
- Utilisation locale dans la fonction ou la procédure

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**

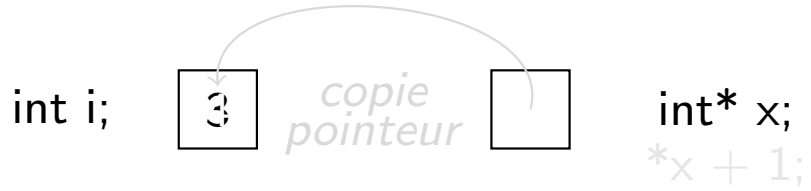
- Le pointeur n'est pas modifié
- La valeur pointée peut l'être

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



Utilisation

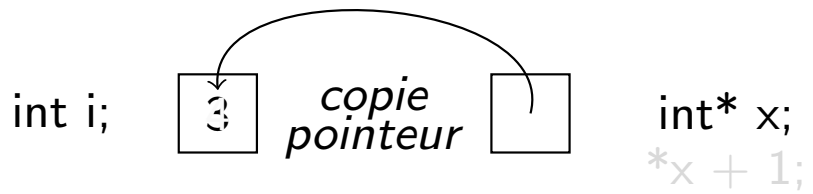
On doit passer les paramètres d'une fonction ou d'une procédure **par leur adresse** lorsque leur valeur est modifiée dans la fonction ou la procédure.

Passage de paramètres par pointeur

```
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



Utilisation

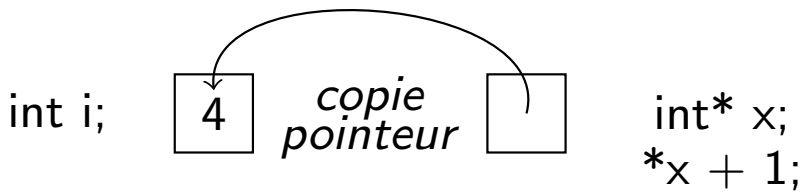
On doit passer les paramètres d'une fonction ou d'une procédure **par leur adresse** lorsque leur valeur est modifiée dans la fonction ou la procédure.

Passage de paramètres par pointeur

```
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



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]`) vaut `NULL`

```
1  int main( int argc, char** argv ) {
2    printf("Nb d\'arguments de %s : %d\n", argv[0], argc);
3    if( argc > 1 ) {
4      printf( "1er argument de %s : %d\n", argv[0], atoi( argv[1] );
5    }
6    return EXIT_SUCCESS;
7  }
```