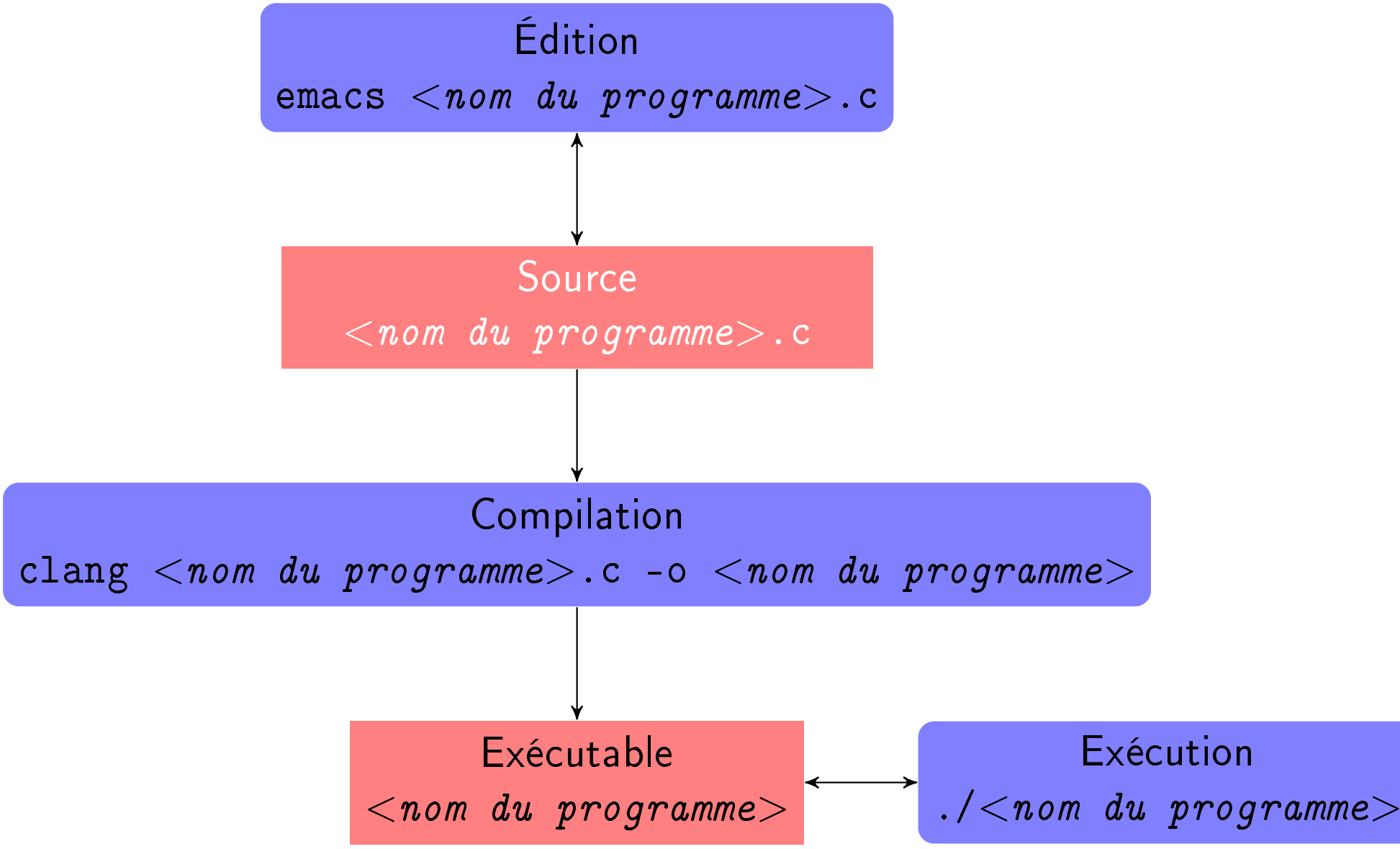
Introduction au C

INF304

2018/2019

Cycle de développement de programmes C



Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme : comment ?

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme :
  - sur un jeu d'essais : ensemble de tests, un test = une entrée possible pour le programme

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme :
  - sur un jeu d'essais : ensemble de tests, un test = une entrée possible pour le programme
  - pas de test «à la main» ! Les tests doivent être enregistrés dans un ou des fichiers, organisés et commentés. Penser à la redirection d'entrée.

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme :
  - sur un jeu d'essais : ensemble de tests, un test = une entrée possible pour le programme
  - pas de test «à la main» ! Les tests doivent être enregistrés dans un ou des fichiers, organisés et commentés. Penser à la redirection d'entrée.
  - couverture : penser au maximum de cas possibles

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme :
  - sur un jeu d'essais : ensemble de tests, un test = une entrée possible pour le programme
  - pas de test «à la main» ! Les tests doivent être enregistrés dans un ou des fichiers, organisés et commentés. Penser à la redirection d'entrée.
  - couverture : penser au maximum de cas possibles
- débogage (propriétés fonctionnelles) : recherche, identification et correction des erreurs
  - instrumentation : pour l'automatisation du test, pour l'identification

Test de programmes

Que faire une fois le programme écrit ?

- ~~rentrer chez soi ?~~
- tester le programme :
  - sur un jeu d'essais : ensemble de tests, un test = une entrée possible pour le programme
  - pas de test «à la main» ! Les tests doivent être enregistrés dans un ou des fichiers, organisés et commentés. Penser à la redirection d'entrée.
  - couverture : penser au maximum de cas possibles
- débogage (propriétés fonctionnelles) : recherche, identification et correction des erreurs
  - instrumentation : pour l'automatisation du test, pour l'identification
- évaluation (propriétés non fonctionnelles) : coût effectif en temps et en mémoire, réactivité, bande passante utilisée, etc.
  - instrumentation pour les mesures de coût

Structure d'une fonction C

type de retour

arguments

```
int nom_fonction (int n, float x) {
    int temp;
    int my_var = 42;
    temp = n + my_var;
    if (x > 0) {
      return temp + x;
    } else {
      return temp - x;
    }
}
```

Déclarations locales

instructions

Fonction principale

Un programme, pour être compilé en un exécutable, doit comporter une fonction `main` de profil :

```
int main(int argc, char ** argv) {
  ...
}
```

- Fonction exécutée au lancement du programme
- `argc` : nombre d'arguments de la ligne de commande (nom du programme inclus)
- `argv` : tableau de chaînes de caractères
  - `argv[i]` : i^e argument de la ligne de commande
  - `argv[0]` : nom du programme exécuté

Qu'est-ce qu'un type ?

Qu'est-ce qu'un type ?

- Abstraction d'une représentation machine : «25» plutôt que «00000000000000000000000000011001»
- Ensemble de valeurs possibles : entiers, entiers naturels, entiers positifs, flottants dans un intervalle borné...
- *Sémantique* (*sens*) donnée à cet ensemble, fournit *et* restreint les opérations possibles sur les valeurs de cet ensemble

Selon le langage de programmation, le typage peut être plus ou moins contraignant, statique (à la compilation) ou dynamique (à l'exécution)

En C : typage statique.

Les types de base en C

- `int` : entier signé (sur 32 bits, entiers dans l'intervalle $[-2^{31}, 2^{31} - 1]$).
- `float` : nombres flottants (représentation machine des nombres réels).
- `char` : caractères codés sur 8 bits.

Les tableaux

Un tableau est groupe d'éléments du même type. La taille d'un tableau est fixe : lorsqu'on déclare un tableau, il faut obligatoirement donner sa taille.

Déclaration d'un tableau

```
Telem Tab[N];
```

`Tab` est un tableau d'éléments de type `Telem` dont les indices varient de 0 à $N-1$. Les indices sont de type `int`.

Accéder à un élément d'un tableau

```
Tab[i] = x;
x = Tab[i];
```

Attention : si `i` n'appartient pas à l'intervalle $[0, N-1]$, le comportement n'est pas défini ! (arrêt du programme avec le message «*segmentation fault*», ou accès à une zone mémoire en-dehors du tableau $\longrightarrow$ erreurs difficiles à analyser)

Types énumérés

Un *type énuméré* permet de définir un ensemble de valeurs par extension (i.e., en donnant la liste des valeurs de l'ensemble).

Déclaration d'un type énuméré `Couleur`, comportant les valeurs `Rouge`, `Jaune`, `Vert` :

```
typedef enum {Rouge, Jaune, Vert} Couleur;

Couleur ma_couleur = Rouge;
```

Structures

Les *types structures* permettent de rassembler plusieurs valeurs de types (éventuellement) différents.

Déclaration d'un type structure `couple` contenant les champs `x` (de type `T1` et `y` (de type T2) :

```
typedef struct {
    T1 x;
    T2 y;
} couple;
```

Déclaration d'une variable de type `couple` :

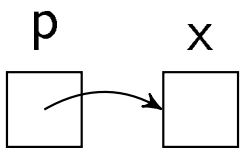
```
couple c;
```

Accès aux valeurs des champs : `c.x`, `c.y`.

Vocabulaire

- si T est un type, le type T $*$ est appelé «*type pointeur de T*». Une valeur de type T $*$ est un *pointeur*, pointant sur une valeur de type T.
- si x est de type T, &x est l'*adresse* de x et est une valeur de type T $*$. Cette valeur peut être affectée à un pointeur de T.
- si p est de type T $*$, alors la *valeur* pointée par p est $*$p, de type T. $*$p est appelé le *déréférencement* de p.

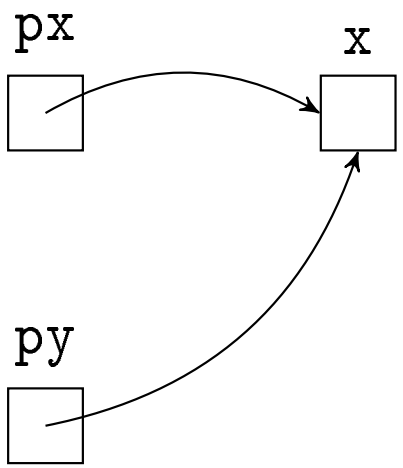
On représente de la manière suivante un pointeur p sur une valeur x :



Exemple

Une variable de type `int *` peut pointer sur une variable de type `int`, à l'aide de l'opérateur `&` :

```
int x;
int * px;
int * py;
x = 1;
px = &x;
py = &x;
```



Pointeurs et structures récursives

Pour la déclaration de types mutuellement récursifs (contenant des pointeurs vers ces types, pour les listes chaînées par exemple), on peut donner un nom à la structure récursive :

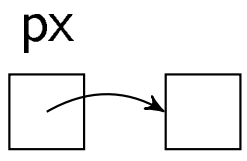
```
// Type cellule pour liste chaînée d'entiers
typedef struct s_cellule {
    int element; // élément courant
    struct s_cellule * suivant; // pointeur vers la cellule suivante
} Cellule;

// Une liste est un pointeur vers la cellule de tête
typedef Cellule * Liste;
```

Allocation mémoire

- Allocation à l'aide de la fonction `malloc`
- Paramètre de `malloc` : taille de la zone mémoire à allouer (utiliser la fonction `sizeof(type)`)
- Valeur de retour : l'adresse mémoire dynamique allouée

```
px = (int *)malloc(sizeof(int));
```

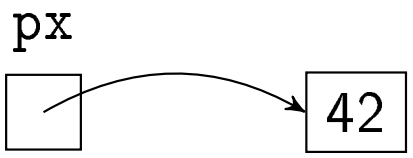


Une zone mémoire contenant un entier est créée, mais non définie (comme pour une déclaration).

Contrairement aux déclarations locales, cette zone mémoire existe encore à la sortie du bloc courant. Elle existe jusqu'à sa libération.

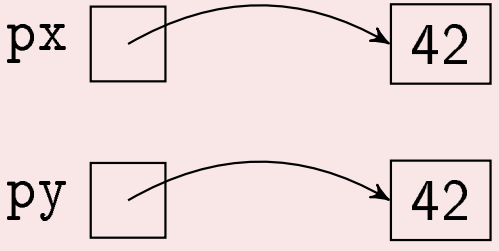
Accès à la valeur pointée

```
*px = 42;
```



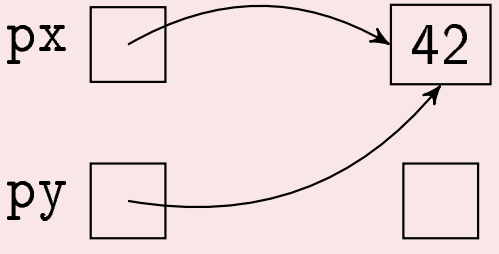
Ne pas confondre

```
*py = *px;
```



et

```
py = px;
```



Libération de la mémoire

Libérer un bloc mémoire = le «rendre» au système, faire en sorte que la zone mémoire puisse à nouveau être utilisée (à l'occasion d'une nouvelle demande d'allocation).

Tout bloc mémoire alloué (avec `malloc` ou équivalent) doit être libéré !
$\longrightarrow$ attention aux *fuites mémoires* : mémoire allouée et jamais libérée...

Si `p` est un pointeur, `free(p)` libère la zone mémoire pointée par `p`. La valeur de `p` doit être une valeur retournée par une primitive d'allocation mémoire : il n'est pas possible de libérer une partie seulement d'un bloc alloué (la moitié d'un tableau par exemple).

Attention : après libération, les valeurs contenues dans une zone mémoire ne doivent plus être accédées (sinon, arrêt du programme avec l'erreur «*segmentation fault*»)

Pointeurs et paramètres résultats

Les paramètres des fonctions C sont passés par valeur (la *valeur* est transmise à la fonction et non la *référence*).

Pour qu'une fonction puisse modifier un paramètre, il faut fournir l'adresse de la valeur à modifier, c'est-à-dire un pointeur sur cette valeur :

```
void echanger(int * x, int * y) {
  int aux;
  aux = *x;
  *x = *y;
  *y = aux;
}

int a = 42;
int b = 3;
echanger(&a, &b);
```

si/alors/sinon

```
if (condition) {
  <instructions>
} else if (condition2) {
  <instructions>
  ...
} else {
  <instructions>
}
```

Exemple

```
if (x < y) {
    printf("x est plus petit que y");
} else {
    printf("x est plus grand que y");
}
```

Switch/case

Exemple

```
switch (x) {
  case 0:
    printf("x est nul");
    break;
  case 1:
    printf("x vaut 1");
    break;
  default:
    printf("x vaut autre chose que 0 ou 1");
}
```

Boucle «tant que» (while)

```
while (condition) {
  <instructions>
}
```

Les instructions sont exécutées tant que la `condition` (booléenne) est vraie

Exemple

```
int tab[N];
int i = 0;
while (i < N) {
  tab[i] = f(i);
  i = i + 1;
}
```

Boucle «pour» (for)

```
for (<initialisation>, <condition>, <mise à jour>) {
  <instructions>
}
```

équivalent à :

```
<initialisation>;
while (<condition>) {
  <instructions>
  <mise à jour>;
}
```

Exemple

```
int tab[N];
int i;
for (i = 0; i < N; i++) {
  tab[i] = f(i);
}
```

Références

The GNU C Reference Manual.
`https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.html`

C Programming. *Wikibooks, The Free Textbook Project*.
`http://en.wikibooks.org/wiki/C_Programming`