Gestion de la mémoire et langage C

Damien Rouhling (Lycée Victor Hugo)

10 mars 2022

Allocation de mémoire

Deux manières d'allouer de la mémoire pour un programme compilé :

1. Statiquement : le compilateur réserve la mémoire à l'avance.
   Exemple : les variables globales/locales d'un programme, si leur taille est statiquement connue.

Allocation de mémoire

Deux manières d'allouer de la mémoire pour un programme compilé :

1. Statiquement : le compilateur réserve la mémoire à l'avance.
   Exemple : les variables globales/locales d'un programme, si leur taille est statiquement connue.
2. Dynamiquement : une demande d'allocation de mémoire est effectuée lors de l'exécution du programme.
   Exemple : un tableau dont la taille dépend de l'état du programme au moment de l'exécution.

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

- Allocation statique :

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

- Allocation statique :
  - Variables globales initialisées : dans une zone spécifique chargée avec le binaire.

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

- Allocation statique :
  - Variables globales initialisées : dans une zone spécifique chargée avec le binaire.
  - Variables globales non initialisées : allouées et initialisées à 0 au moment du chargement du binaire, après les variables globales initialisées.

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

- Allocation statique :
  - Variables globales initialisées : dans une zone spécifique chargée avec le binaire.
  - Variables globales non initialisées : allouées et initialisées à 0 au moment du chargement du binaire, après les variables globales initialisées.
  - Variables locales et paramètres des fonctions : dans une zone appelée pile, parfois dans les registres du processeur.

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

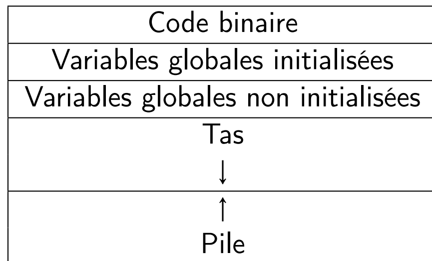
- Allocation statique :
  - Variables globales initialisées : dans une zone spécifique chargée avec le binaire.
  - Variables globales non initialisées : allouées et initialisées à 0 au moment du chargement du binaire, après les variables globales initialisées.
  - Variables locales et paramètres des fonctions : dans une zone appelée pile, parfois dans les registres du processeur.
- Allocation dynamique : dans une zone appelée tas.

Structure de la mémoire d'un programme

L'emplacement de la mémoire allouée dépend du type d'allocation :

- Allocation statique :
  - Variables globales initialisées : dans une zone spécifique chargée avec le binaire.
  - Variables globales non initialisées : allouées et initialisées à 0 au moment du chargement du binaire, après les variables globales initialisées.
  - Variables locales et paramètres des fonctions : dans une zone appelée pile, parfois dans les registres du processeur.
- Allocation dynamique : dans une zone appelée tas.
- Dans tous les cas : dans la RAM.

Vision schématique de la mémoire d'un programme



Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.
  - L'adresse de retour.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.
  - L'adresse de retour.
  - L'adresse du bloc prédécent.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.
  - L'adresse de retour.
  - L'adresse du bloc prédécent.
  - Une zone pour le résultat de la fonction.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.
  - L'adresse de retour.
  - L'adresse du bloc prédécent.
  - Une zone pour le résultat de la fonction.
  - Les variables locales de la fonction.

Gestion de la pile d'exécution

- Les allocations dans la pile se font dynamiquement et automatiquement.
- Fonctionne comme une pile dépendant des appels de fonction : on empile un bloc à chaque appel et on le dépile lorsque l'appel de fonction se termine.
- On empile un bloc d'activation qui contient :
  - Les paramètres de la fonction.
  - L'adresse de retour.
  - L'adresse du bloc prédécent.
  - Une zone pour le résultat de la fonction.
  - Les variables locales de la fonction.
- La pile peut être optimisée pour les fonctions récursives terminales.

Gestion du tas

- Automatique en OCaml, grâce au garbage collector.

Gestion du tas

- Automatique en OCaml, grâce au garbage collector.
- Explicite en C, via des fonctions d'allocation et de libération de la mémoire.

Gestion du tas

- Automatique en OCaml, grâce au garbage collector.
- Explicite en C, via des fonctions d'allocation et de libération de la mémoire.
- On fait référence aux blocs de mémoire par leur adresse (pointeurs).

Le langage C

- Développé à Bell Labs en 1972 par Dennis Ritchie.

Le langage C

- Développé à Bell Labs en 1972 par Dennis Ritchie.
- Paradigme impératif structuré.

Le langage C

- Développé à Bell Labs en 1972 par Dennis Ritchie.
- Paradigme impératif structuré.
- Langage compilé, comme OCaml.

Le langage C

- Développé à Bell Labs en 1972 par Dennis Ritchie.
- Paradigme impératif structuré.
- Langage compilé, comme OCaml.
- Langage de plus bas niveau qu'OCaml.

Le langage C

- Développé à Bell Labs en 1972 par Dennis Ritchie.
- Paradigme impératif structuré.
- Langage compilé, comme OCaml.
- Langage de plus bas niveau qu'OCaml.
- Typage statique faible sans inférence.

Structure d'un programme en C

Comme en OCaml, un programme est une suite de déclarations.

Structure d'un programme en C

Comme en OCaml, un programme est une suite de déclarations.

MAIS

- Il y a un point d'entrée, sous la forme d'une fonction principale nommée `main`.

Structure d'un programme en C

Comme en OCaml, un programme est une suite de déclarations.

MAIS

- Il y a un point d'entrée, sous la forme d'une fonction principale nommée `main`.
- Exécuter le programme revient à exécuter les instructions de la fonction `main` (qui peuvent dépendre des déclarations précédentes).

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...
- **float**/**double** : `1.`, `2.5 + 3.2e5`, `a / b`...

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...
- **float** / **double** : `1.`, `2.5 + 3.2e5`, `a / b`...
- `bool` : `true`, `b1 && b2`, `b1 || b2`, `!b`, `x == y`...

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...
- **float** / **double** : `1.`, `2.5 + 3.2e5`, `a / b`...
- `bool` : `true`, `b1 && b2`, `b1 || b2`, `!b`, `x == y`...
- **char** : `'a'`, `'b'`...

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...
- **float** / **double** : `1.`, `2.5 + 3.2e5`, `a / b`...
- `bool` : `true`, `b1 && b2`, `b1 || b2`, `!b`, `x == y`...
- **char** : `'a'`, `'b'`...
- Pas de type `string` : on utilise des tableaux de caractères, mais on peut tout de même écrire `"bonjour"`.

Expressions

Voici certains types de base :

- **int** : `1`, `2 + 8`, `a / b`...
- **float**/**double** : `1.`, `2.5 + 3.2e5`, `a / b`...
- `bool` : `true`, `b1 && b2`, `b1 || b2`, `!b`, `x == y`...
- **char** : `'a'`, `'b'`...
- Pas de type `string` : on utilise des tableaux de caractères, mais on peut tout de même écrire `"bonjour"`.
- **void** : joue un rôle similaire au type `unit` d'OCaml, mais il est vide.

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.
- Entrées/sorties :

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.
- Entrées/sorties :
  - Sorties :
    `printf(<chaîne de format>, <arguments éventuels>);`
    Exemple : `printf("Bonjour.\n");`, `printf("x vaut %d", x);`
    Formats : `%d`, `%f`, `%lf`, `%c`, `%s`...

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.
- Entrées/sorties :
  - Sorties :
    `printf(<chaîne de format>, <arguments éventuels>);`
    Exemple : `printf("Bonjour.\n");`, `printf("x vaut %d", x);`
    Formats : `%d`, `%f`, `%lf`, `%c`, `%s`...
  - Entrées :
    `scanf(<chaîne de format>, <adresses éventuelles>);`
    Exemple : `scanf("%d", &x);`, `scanf("%s", s);`

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.
- Entrées/sorties :
  - Sorties :
    `printf(<chaîne de format>, <arguments éventuels>);`
    Exemple : `printf("Bonjour.\n");`, `printf("x vaut %d", x);`
    Formats : `%d`, `%f`, `%lf`, `%c`, `%s`...
  - Entrées :
    `scanf(<chaîne de format>, <adresses éventuelles>);`
    Exemple : `scanf("%d", &x);`, `scanf("%s", s);`
- Instructions conditionnelles : **if** `(<expr>) <instr>` [**else** `<instr>`].
  On peut délimiter des blocs d'instructions à l'aide d'accolades.

Instructions

- Déclaration : `<type> <nom>;` ou `<type> <nom> = <expr>;`.
- Affectation : `<nom> = <expr>;`.
- Incrémentation/décrémentation : `<nom>++;` ou `<nom>--;`.
- Entrées/sorties :
  - Sorties :
    `printf(<chaîne de format>, <arguments éventuels>);`
    Exemple : `printf("Bonjour.\n");`, `printf("x vaut %d", x);`
    Formats : `%d`, `%f`, `%lf`, `%c`, `%s`...
  - Entrées :
    `scanf(<chaîne de format>, <adresses éventuelles>);`
    Exemple : `scanf("%d", &x);`, `scanf("%s", s);`
- Instructions conditionnelles : **if** `(<expr>) <instr>` [**else** `<instr>`].
  On peut délimiter des blocs d'instructions à l'aide d'accolades.
- Boucles :

Instructions

- Déclaration : <type> <nom>; ou <type> <nom> = <expr>;.
- Affectation : <nom> = <expr>;.
- Incrémentation/décrémentation : <nom>++; ou <nom>--;.
- Entrées/sorties :
  - Sorties :
    printf(<chaîne de format>, <arguments éventuels>);
    Exemple : printf("Bonjour.\n");, printf("x vaut %d", x);
    Formats : %d, %f, %lf, %c, %s...
  - Entrées :
    scanf(<chaîne de format>, <adresses éventuelles>);
    Exemple : scanf("%d", &x);, scanf("%s", s);
- Instructions conditionnelles : **if** (<expr>) <instr> [**else** <instr>].
  On peut délimiter des blocs d'instructions à l'aide d'accolades.
- Boucles :
  - **while** (<expr>) <instr>

Instructions

- Déclaration : <type> <nom>; ou <type> <nom> = <expr>;.
- Affectation : <nom> = <expr>;.
- Incrémentation/décrémentation : <nom>++; ou <nom>--;.
- Entrées/sorties :
  - Sorties :
    printf(<chaîne de format>, <arguments éventuels>);
    Exemple : printf("Bonjour.\n");, printf("x vaut %d", x);
    Formats : %d, %f, %lf, %c, %s...
  - Entrées :
    scanf(<chaîne de format>, <adresses éventuelles>);
    Exemple : scanf("%d", &x);, scanf("%s", s);
- Instructions conditionnelles : **if** (<expr>) <instr> [**else** <instr>].
  On peut délimiter des blocs d'instructions à l'aide d'accolades.
- Boucles :
  - **while** (<expr>) <instr>
  - **for** ([<instr>]; [<expr>]; [<instr>]) <instr>

Fonctions

- Déclaration :

```
<type_res> <nom_fonction> (<type_arg1> <nom_arg1>,
  ..., <type_argn> <nom_argn>)
  <instr>
```

Fonctions

- Déclaration :

```
<type_res> <nom_fonction> (<type_arg1> <nom_arg1>,
  ..., <type_argn> <nom_argn>)
  <instr>
```

- Pour renvoyer un résultat/interrompre la fonction :

```
return <expr>;
return;
```

Fonctions

- Déclaration :

```
<type_res> <nom_fonction> (<type_arg1> <nom_arg1>,
  ..., <type_argn> <nom_argn>)
  <instr>
```

- Pour renvoyer un résultat/interrompre la fonction :

```
return <expr>;
return;
```

- Usage d'une fonction : `<nom_fonction>(<arg1>, ..., <argn>)`

Fonctions

- Déclaration :

```
<type_res> <nom_fonction> (<type_arg1> <nom_arg1>,
  ..., <type_argn> <nom_argn>)
  <instr>
```

- Pour renvoyer un résultat/interrompre la fonction :

```
return <expr>;
return;
```

- Usage d'une fonction : `<nom_fonction>(<arg1>, ..., <argn>)`
- Les fonctions dont le résultat est de type **void** peuvent être utilisées comme des instructions (avec un point-virgule).

Modification des arguments et résultats multiples

On voudrait pouvoir :

- Modifier la valeur des arguments d'une fonction.
  Fonction incorrecte :

```
void swap (int x, int y) {
  int z = x;
  x = y;
  y = z;
}
```

Modification des arguments et résultats multiples

On voudrait pouvoir :

- Modifier la valeur des arguments d'une fonction.
  Fonction incorrecte :

```
void swap (int x, int y) {
  int z = x;
  x = y;
  y = z;
}
```

- Renvoyer plusieurs résultats : pas de type des *n*-uplets ni de possibilité de renvoyer un tableau déclaré localement.

Modification des arguments et résultats multiples

On voudrait pouvoir :

- Modifier la valeur des arguments d'une fonction.
  Fonction incorrecte :

```
void swap (int x, int y) {
  int z = x;
  x = y;
  y = z;
}
```

- Renvoyer plusieurs résultats : pas de type des *n*-uplets ni de possibilité de renvoyer un tableau déclaré localement.

$\rightsquigarrow$ on utilise en paramètres les adresses de cases mémoire que l'on pourra modifier.

Pointeurs

- Un pointeur est un entier représentant l'adresse d'une case mémoire.

Pointeurs

- Un pointeur est un entier représentant l'adresse d'une case mémoire.
- Type d'un pointeur : `<type_valeur>*`.
  Exemple : **int***, **char****, ...

Pointeurs

- Un pointeur est un entier représentant l'adresse d'une case mémoire.
- Type d'un pointeur : `<type_valeur>*`.
  Exemple : **int** *, **char** **, ...
- Pointeur sur rien : `NULL`.

Pointeurs

- Un pointeur est un entier représentant l'adresse d'une case mémoire.
- Type d'un pointeur : `<type_valeur>*`.
  Exemple : **int**∗, **char**∗∗, ...
- Pointeur sur rien : `NULL`.
- Opérateur d'adressage : `&<expr>`.

```
int x = 3;
int *p = &x;
```

Pointeurs

- Un pointeur est un entier représentant l'adresse d'une case mémoire.
- Type d'un pointeur : `<type_valeur>*`.
  Exemple : **int**∗, **char**∗∗, . . .
- Pointeur sur rien : `NULL`.
- Opérateur d'adressage : `&<expr>`.

```
int x = 3;
int *p = &x;
```

- Opérateur de déréférencement : `*<expr>`.

```
*p = 42;
printf("%d\n", x);
```

Retour à nos problèmes

- Modification des paramètres :

```
void swap (int* x, int* y) {
  int z = *x;
  *x = *y;
  *y = z;
}
```

Retour à nos problèmes

- Modification des paramètres :

```
void swap (int* x, int* y) {
  int z = *x;
  *x = *y;
  *y = z;
}
```

- Plusieurs valeurs de retour :

```
void min_max (int x, int y, int* min, int* max) {
  if (x <= y) {
    *min = x;
    *max = y;
  } else {
    *min = y;
    *max = x;
  }
}
```

Tableaux

- Déclaration de tableaux statiquement alloués :

```
int t[12];
int t[3] = {1, 2, 3};
```

Tableaux

- Déclaration de tableaux statiquement alloués :

```
int t[12];
int t[3] = {1, 2, 3};
```

- Accès et écritures : on utilise `<nom>[<expr>]`.

```
t[0] = 0;
t[i + 1] = t[i] - 1;
```

Tableaux

- Déclaration de tableaux statiquement alloués :

```
int t[12];
int t[3] = {1, 2, 3};
```

- Accès et écritures : on utilise `<nom>[<expr>]`.

```
t[0] = 0;
t[i + 1] = t[i] - 1;
```

- Les tableaux sont des pointeurs.

```
int t[23], *p;
p = t;
*(p + 1) = 2;
```

Tableaux

- Déclaration de tableaux statiquement alloués :

```
int t[12];
int t[3] = {1, 2, 3};
```

- Accès et écritures : on utilise `<nom>[<expr>]`.

```
t[0] = 0;
t[i + 1] = t[i] - 1;
```

- Les tableaux sont des pointeurs.

```
int t[23], *p;
p = t;
*(p + 1) = 2;
```

- Tableaux dynamiquement alloués : on alloue explicitement un pointeur sur un objet de taille `<longueur> * <taille d'un élément>`.

Allocation dynamique

- Fonction d'allocation :

```
void* malloc(size_t size)
```

Allocation dynamique

- Fonction d'allocation :

```
void* malloc(size_t size)
```

- Fonction de libération de la mémoire :

```
void free(void* ptr)
```

Allocation dynamique

- Fonction d'allocation :

```c
void* malloc(size_t size)
```

- Fonction de libération de la mémoire :

```c
void free(void* ptr)
```

- Exemple :

```c
int* t = malloc(n * sizeof(int));
for (int i = 0; i < n; i++)
  t[i] = i + 1;
free(t);
```

Structures et déclarations de types

- Déclaration de type : **typedef** `<type> <nom_type>;`

```
typedef int booleen;
booleen x = 0;
```

Structures et déclarations de types

- Déclaration de type : **typedef** `<type> <nom_type>;`

```
typedef int booleen;
booleen x = 0;
```

- Déclaration de structures :

```
struct personne {
  char nom[20];
  char prenom[20];
}
```

```
struct personne moi;
strcpy(moi.nom, "Rouhling");
strcpy(moi.prenom, "Damien");
```

Structures et déclarations de types

- Déclaration de type : **typedef** `<type> <nom_type>;`

```
typedef int booleen;
booleen x = 0;
```

- Déclaration de structures :

```
struct personne {
  char nom[20];
  char prenom[20];
}
```

```
struct personne moi;
strcpy(moi.nom, "Rouhling");
strcpy(moi.prenom, "Damien");
```

- Combinaison des deux :

```
typedef struct {
  float abscisse;
  float ordonnee;
} point;
point x = {1.5, 2.3};
```

Structures inductives

On utilise des pointeurs vers la structure :

```
typedef struct t_cellule {
  int valeur;
  struct t_cellule *suivant;
} cellule;

typedef struct {
  cellule* tete;
} liste;
```

Le pointeur `NULL` représente alors la liste vide.

Des questions ?