Introduction au Langage C

Vincent Vidal

Maître de Conférences
Enseignements : IUT Lyon 1 - pôle AP - Licence ESSIR - bureau 2ème étage
Recherche : Laboratoire LIRIS - bât. Nautibus
E-mail : vincent.vidal@univ-lyon1.fr

Supports de cours et TPs : `http://claroline connect.univ-lyon1.fr` espace d'activités "M1102 M1103 C - Introduction au langage C"

46H prévues $\approx$ 42H de cours+TPs, 2H - interros, et 2H - examen final

Évaluation : Contrôle continu + examen final + Bonus/Malus TP

Plan

Plan

Les variables automatiques et statiques (1/2)

- *Automatiques* :
  - les variables locales usuelles (variables intermédiaires) ;
  - les arguments/paramètres d'une fonction.
- *Statiques* :
  - les variables globales ;
  - les variables locales statiques (mot clef `static`).

Les variables automatiques et statiques (1/2)

- *Automatiques* :
  - les variables locales usuelles (variables intermédiaires) ;
  - les arguments/paramètres d'une fonction.
- *Statiques* :
  - les variables globales ;
  - les variables locales statiques (mot clef `static`).

Les variables automatiques et statiques (1/2)

- *Automatiques* :
  - les variables locales usuelles (variables intermédiaires) ;
  - les arguments/paramètres d'une fonction.
- *Statiques* :
  - les variables globales ;
  - les variables locales statiques (mot clef `static`).

Les variables automatiques et statiques (1/2)

- *Automatiques* :
  - les variables locales usuelles (variables intermédiaires) ;
  - les arguments/paramètres d'une fonction.
- *Statiques* :
  - les variables globales ;
  - les variables locales statiques (mot clef **static**).

Les variables automatiques et statiques (2/2)

Les variables automatiques et statiques sont gérées (allocation et libération) **automatiquement par le compilateur**.

Les **variables statiques** existent pendant toute la durée d'exécution du programme : elles ont un emplacement mémoire réservé.

Les **variables automatiques** sont gérées en fonction des appels aux fonctions à l'aide d'une *pile* : allocation au moment de l'appel à la fonction et libération à la fin de l'exécution de la fonction.

Les variables automatiques et statiques (2/2)

Les variables automatiques et statiques sont gérées (allocation et libération) **automatiquement par le compilateur**.

Les **variables statiques** existent pendant toute la durée d'exécution du programme : elles ont un emplacement mémoire réservé.

Les **variables automatiques** sont gérées en fonction des appels aux fonctions à l'aide d'une *pile* : allocation au moment de l'appel à la fonction et libération à la fin de l'exécution de la fonction.

Les variables automatiques et statiques (2/2)

Les variables automatiques et statiques sont gérées (allocation et libération) **automatiquement par le compilateur**.

Les **variables statiques** existent pendant toute la durée d'exécution du programme : elles ont un emplacement mémoire réservé.

Les **variables automatiques** sont gérées en fonction des appels aux fonctions à l'aide d'une *pile* : allocation au moment de l'appel à la fonction et libération à la fin de l'exécution de la fonction.

Les variables dynamiques gérées dans le tas

Les variables dynamiques sont gérées (allocation et libération) **par le programmeur via des commandes explicites** :

1. **allocation dynamique dans le *tas*** :
   fonction *malloc*(nombre d'octets à réserver)
2. **libération dynamique dans le *tas*** :
   fonction *free*(@ début du bloc mémoire à libérer)

Intérêts ?

- La mémoire allouée pour un tableau est minimale ;
- certaines structures de données *dynamiques* (listes chaînées, arbres binaires...), elles seront vues en SDD.

Les variables dynamiques gérées dans le tas

Les variables dynamiques sont gérées (allocation et libération) **par le programmeur via des commandes explicites** :

1. **allocation dynamique dans le *tas*** :
   fonction *malloc*(nombre d'octets à réserver)
2. **libération dynamique dans le *tas*** :
   fonction *free*(@ début du bloc mémoire à libérer)

Intérêts ?

- La mémoire allouée pour un tableau est minimale ;
- certaines structures de données *dynamiques* (listes chaînées, arbres binaires...), elles seront vues en SDD.

Les variables dynamiques gérées dans le tas

Les variables dynamiques sont gérées (allocation et libération) **par le programmeur via des commandes explicites** :

1. **allocation dynamique dans le *tas*** :
   fonction *malloc*(nombre d'octets à réserver)
2. **libération dynamique dans le *tas*** :
   fonction *free*(@ début du bloc mémoire à libérer)

Intérêts ?

- La mémoire allouée pour un tableau est minimale ;
- certaines structures de données *dynamiques* (listes chaînées, arbres binaires...), elles seront vues en SDD.

Les variables dynamiques gérées dans le tas

Les variables dynamiques sont gérées (allocation et libération) **par le programmeur via des commandes explicites** :

1. **allocation dynamique dans le *tas*** :
   fonction *malloc*(nombre d'octets à réserver)
2. **libération dynamique dans le *tas*** :
   fonction *free*(@ début du bloc mémoire à libérer)

Intérêts ?

- La mémoire allouée pour un tableau est minimale ;
- certaines structures de données *dynamiques* (listes chaînées, arbres binaires...), elles seront vues en SDD.

Les variables dynamiques gérées dans le tas

Les variables dynamiques sont gérées (allocation et libération) **par le programmeur via des commandes explicites** :

1. **allocation dynamique dans le *tas*** :
   fonction *malloc*(nombre d'octets à réserver)
2. **libération dynamique dans le *tas*** :
   fonction *free*(@ début du bloc mémoire à libérer)

Intérêts ?

- La mémoire allouée pour un tableau est minimale ;
- certaines structures de données *dynamiques* (listes chaînées, arbres binaires...), elles seront vues en SDD.

La fonction malloc (1/2)

#include <stdlib.h>
prototype : **void*** malloc(size_t *taille*) ;

- **void*** est une adresse générique, l'adresse du début d'un bloc mémoire alloué ; le bloc mémoire alloué fait *taille* octets successifs ; si l'allocation a échoué, malloc retourne NULL.
- *taille* est un entier naturel qui désigne le nombre d'octets nécessaires pour stocker nos données ;

  pour stocker 1 **int**, **sizeof(int)** octets sont nécessaires ;
  pour stocker n **int** successifs (un tableau de **int** de taille n), n***sizeof(int)** octets sont nécessaires.

La fonction malloc (1/2)

#include <stdlib.h>
prototype : **void*** malloc(size_t *taille*) ;

- **void*** est une adresse générique, l'adresse du début d'un bloc mémoire alloué ; le bloc mémoire alloué fait *taille* octets successifs ; si l'allocation a échoué, malloc retourne NULL.
- *taille* est un entier naturel qui désigne le nombre d'octets nécessaires pour stocker nos données ;

pour stocker 1 **int**, **sizeof(int)** octets sont nécessaires ;
pour stocker n **int** successifs (un tableau de **int** de taille n), n***sizeof(int)** octets sont nécessaires.

La fonction malloc (1/2)

#include <stdlib.h>
prototype : **void*** malloc(size_t *taille*) ;

- **void*** est une adresse générique, l'adresse du début d'un bloc mémoire alloué ; le bloc mémoire alloué fait *taille* octets successifs ; si l'allocation a échoué, malloc retourne NULL.
- *taille* est un entier naturel qui désigne le nombre d'octets nécessaires pour stocker nos données ;

  pour stocker 1 **int**, **sizeof(int)** octets sont nécessaires ;
  pour stocker n **int** successifs (un tableau de **int** de taille n), n***sizeof(int)** octets sont nécessaires.

La fonction malloc (2/2)

A vous de jouer : combien d'octets sont nécessaires pour stocker un tableau de 10 **double** ? De 15 **char** ?

La fonction malloc (2/2)

A vous de jouer : combien d'octets sont nécessaires pour stocker un tableau de 10 **double** ? De 15 **char** ?

Ecrivez les lignes de code pour le faire.

La fonction calloc

#include <stdlib.h>
prototype : **void*** calloc(size_t *nb_blocs*, size_t *taille*) ;

calloc alloue l'emplacement pour *nb_blocs* successifs, chacun ayant une taille de *taille* octet(s).

calloc met à zéro chaque octet de la zone allouée.

calloc permet d'allouer au max "size_t * size_t" octets alors que malloc est limité à "size_t".

La fonction calloc

#include <stdlib.h>
prototype : **void*** calloc(size_t *nb_blocs*, size_t *taille*) ;

calloc alloue l'emplacement pour *nb_blocs* successifs, chacun ayant une taille de *taille* octet(s).

calloc met à zéro chaque octet de la zone allouée.

calloc permet d'allouer au max "size_t * size_t" octets alors que malloc est limité à "size_t".

La fonction calloc

#include <stdlib.h>
prototype : **void*** calloc(size_t *nb_blocs*, size_t *taille*) ;

calloc alloue l'emplacement pour *nb_blocs* successifs, chacun ayant une taille de *taille* octet(s).

calloc met à zéro chaque octet de la zone allouée.

calloc permet d'allouer au max "size_t * size_t" octets alors que malloc est limité à "size_t".

La fonction realloc

#include <stdlib.h>
prototype : **void*** realloc(**void*** *ptr*, size_t *taille*) ;

realloc modifie la taille d'un bloc mémoire déjà alloué par malloc, calloc ou realloc.

ptr désigne l'adresse du début de la zone anciennement allouée. *taille* représente la nouvelle taille désirée pour le nouveau bloc mémoire réservé.

Le contenu de l'ancienne zone est préservé, quitte à le recopier si l'adresse du début de la zone mémoire allouée change.

La fonction realloc

#include <stdlib.h>
prototype : **void*** realloc(**void*** *ptr*, size_t *taille*) ;

realloc modifie la taille d'un bloc mémoire déjà alloué par malloc, calloc ou realloc.

ptr désigne l'adresse du début de la zone anciennement allouée. *taille* représente la nouvelle taille désirée pour le nouveau bloc mémoire réservé.

Le contenu de l'ancienne zone est préservé, quitte à le recopier si l'adresse du début de la zone mémoire allouée change.

La fonction realloc

#include <stdlib.h>
prototype : **void*** realloc(**void*** *ptr*, size_t *taille*) ;

realloc modifie la taille d'un bloc mémoire déjà alloué par malloc, calloc ou realloc.

ptr désigne l'adresse du début de la zone anciennement allouée. *taille* représente la nouvelle taille désirée pour le nouveau bloc mémoire réservé.

Le contenu de l'ancienne zone est préservé, quitte à le recopier si l'adresse du début de la zone mémoire allouée change.

La fonction free

#include <stdlib.h>
prototype : **void** free(**void*** *ptr*) ;

- *ptr* est une adresse générique, l'adresse du début d'un bloc mémoire alloué par malloc, calloc ou realloc.

La fonction free

#include <stdlib.h>
prototype : **void** free(**void*** *ptr*) ;

- *ptr* est une adresse générique, l'adresse du début d'un bloc mémoire alloué par malloc, calloc ou realloc.

Gestion de la mémoire pour un élément

```c
#include <stdio.h>
#include <stdlib.h> /* malloc et free */

int main(void) {
   /* ALLOCATION DYNAMIQUE */
   int * pMyInt = NULL;

   pMyInt = (int *) malloc( sizeof(int) ); /* remarquez le cast de pointeur */
   if( pMyInt == NULL )
   {
      printf("Allocation memoire echouee.\n");
      exit(EXIT_FAILURE);
   }
   /* INSTRUCTION(S) */
   *pMyInt = 4 ;
   printf("%d\n", *pMyInt) ;

   /* LIBERATION DYNAMIQUE : pour chaque malloc un free, sinon fuite mémoire */
   free(pMyInt); /* ATTENTION : on libère des pointeurs différents de NULL! */

   return EXIT_SUCCESS;
}
```

La fonction C exit

La fonction exit termine le programme "proprement", c'est-à-dire que toutes les ressources utilisées par le programme sont proprement rendues au système.

La constante EXIT_SUCCESS signale que le programme s'est déroulé normalement. La constante EXIT_FAILURE signale l'échec du déroulement du programme.

La fonction C exit

La fonction exit termine le programme "proprement", c'est-à-dire que toutes les ressources utilisées par le programme sont proprement rendues au système.

La constante EXIT_SUCCESS signale que le programme s'est déroulé normalement. La constante EXIT_FAILURE signale l'échec du déroulement du programme.

Gestion de la mémoire pour NB éléments contiguës (1/3)

```c
#include <stdio.h>
#include <stdlib.h> /* malloc et free */


int main(void) {
   /* ALLOCATION DYNAMIQUE DE TABLEAU 1D */
   const int NB = 5 ; /* nb cases d'un tableau 1D */
   int * pMyInts = (int *) malloc( NB * sizeof(int) ); /* nombre de cases x taille d'une case */
   if( pMyInts == NULL )
   {
      printf("Allocation memoire echouee.\n");
      exit(EXIT_FAILURE);
   }
   /* INSTRUCTION(S) */
   pMyInts[2] = 5 ; ...
   /* LIBERATION DYNAMIQUE : pour chaque malloc un free, sinon fuite mémoire */
   free(pMyInts);

   return EXIT_SUCCESS;
}
```

Gestion de la mémoire pour NB éléments contiguës (2/3)

```c
#include <stdio.h>
#include <stdlib.h> /* malloc et free */
#define TYPE float /* une approche générique */

int main(void) {
   /* ALLOCATION DYNAMIQUE DE TABLEAU 1D */
   const int NB = 5 ; /* nb cases d'un tableau 1D */
   TYPE * pElms = (TYPE *) malloc( NB * sizeof(TYPE) );
   if( pElms == NULL )
   {
      printf("Allocation memoire echouee.\n");
      exit(EXIT_FAILURE);
   }
   /* INSTRUCTION(S) */
   pElms[2] = (TYPE)5 ; ...
   /* LIBERATION DYNAMIQUE : pour chaque malloc un free, sinon fuite mémoire */
   free(pElms);

   return EXIT_SUCCESS;
}
```

Gestion de la mémoire pour NB éléments contiguës (3/3)

```c
#include <stdio.h>
#include <stdlib.h> /* malloc et free */
#define TYPE float /* une approche générique */

int main(void) {
   /* ALLOCATION DYNAMIQUE DE TABLEAU 1D ET INIT A 0 DES CASES */
   const int NB = 5 ; /* nb cases d'un tableau 1D */
   TYPE * pElms = (TYPE *) calloc( NB, sizeof(TYPE) );
   if( pElms == NULL )
   {
      printf("Allocation memoire echouee.\n");
      exit(EXIT_FAILURE);
   }
   /* INSTRUCTION(S) */
   pElms[2] = (TYPE)5 ; ...
   /* LIBERATION DYNAMIQUE : pour chaque calloc un free, sinon fuite mémoire */
   free(pElms);

   return EXIT_SUCCESS;
}
```

```c
#include <stdio.h>
#include <stdlib.h> /* malloc et free */

int main(void) {
   int i, j ;
   const int NB_L = 5, NB_C = 7 ;
   int ** pMyInts = (int **) malloc( NB_L * sizeof(int*) ); /* TAB 1D DE POINTEURS */
   if( pMyInts == NULL ){
      printf("Allocation memoire echouee.\n"); exit(EXIT_FAILURE);
   }
   for(i=0; i<NB_L; ++i){
      pMyInts[i] = (int *) malloc( NB_C * sizeof(int) ); /* TAB 1D D'ENTIERS */
      if( pMyInts[i] == NULL ){
         for(j=0; j<i; ++j){
            free(pMyInts[j]);
            pMyInts[j]=NULL;
         }
         free(pMyInts);
         printf("Allocation memoire echouee.\n"); exit(EXIT_FAILURE);
      }
   }
   /* manipulation du tableau et ensuite libération */
   for(i=0; i<NB_L; ++i){
      free(pMyInts[i]);
      pMyInts[i]=NULL;
   }
   free(pMyInts);
   return EXIT_SUCCESS;
}
```

Déclaration et utilisation d'une variable entière

Automatique

```
typeRetour fonction( liste arg )
{
   int v; /* v variable locale
             automatique*/


   v = 5 ; /* utilisation */
   scanf("%d", &v);
   printf("%d", v);

           /* libération automatique
              à la fin de fonction */
}
```

Dynamique

```
typeRetour fonction( liste arg )
{
   int* ad_v; /* ad_v variable locale automatique */
   ad_v = (int *)malloc(sizeof(int)) ;
   /* *ad_v est une variable dynamique */

   *ad_v = 5 ;  /* utilisation */
   scanf("%d", ad_v);
   printf("%d", *ad_v);

   free(ad_v) ; /* libération pas automatique */

}
```