

Initiation au C

cours n°8

Antoine Miné

École normale supérieure

12 avril 2007

Plan du cours

- Conversions de types : l'opérateur `(type)`.
- Allocation dynamique de mémoire : `malloc`, `realloc`, `free`.

Conversions de types

Rappels sur les types entiers

Types entiers sur un Intel 32-bit

type	sizeof	ensemble représenté
char, signed char	1	$[-128, 127]$
unsigned char	1	$[0, 255]$
short	2	$[-32768, 32767]$
unsigned short	2	$[0, 65535]$
int, long	4	$[-2^{31}, 2^{31} - 1]$
unsigned, unsigned long	4	$[0, 2^{32} - 1]$
long long	8	$[-2^{63}, 2^{63} - 1]$
unsigned long long	8	$[0, 2^{64} - 1]$

Calculs dans les entiers :

- non-signés : calcul modulo $2^{8 \times \text{sizeof}}$,
- signés : résultat aléatoire en cas de dépassement de capacité,
- erreur à l'exécution en cas de division par 0 (/ ou %).

Rappels sur les types flottants

Types flottants sur un Intel 32-bit

type	sizeof	min	max	précision
float	4	10^{-45}	3×10^{38}	10^{-7}
double	8	5×10^{-324}	10^{308}	10^{-16}
long double	12	10^{-4933}	10^{4912}	10^{-19}

Calculs dans les flottants :

- calculs arrondis à la précision du type,
- nombres spéciaux : $+\infty$, $-\infty$, *NaN*.

Choix d'un type arithmétique

Le type donne un compromis entre l'expressivité et le coût (mémoire et temps).

Le type définit la sémantique des opérations.

Exemple

```
int a,b,c;  
double x,y,z;  
  
c = a / b; /* division entière (troncature) */  
z = x / y; /* division flottante */  
  
c = a % b; /* modulo entier */  
z = x % y; /* ERREUR */
```

La sémantique est donnée par le type des **arguments** de l'opérateur.

Conversions explicites de types

Opérateur de **conversion de type** (*cast* en anglais).

Syntaxe

(type) *expr*

Effet : convertit la valeur de *expr* dans le type précisé.

Exemple

```
int x,y;  
double d;  
d = (double)x / (double)y; /* division flottante */
```

Attention à la forte priorité de l'opérateur (type) :

Exemple : (int)x+y signifie ((int) x)+y et non (int)(x+y).

Effet sur les valeurs

Règles de conversion :

- si la valeur est représentable exactement dans type,
⇒ la valeur est **inchangée**,
- en cas de conversion flottant → entier,
⇒ la valeur est **tronquée**,
- en cas de dépassement de capacité :
 - si type est flottant : le résultat est $+\infty$ ou $-\infty$,
 - si type est non signé : le résultat est pris **modulo $2^{8 \times \text{sizeof}}$** ,
 - si type est signé : le résultat **dépend de la machine**.

Exemples

```
(int) (1.5 + 0.5)    /* donne 2 */  
(int) (1.0 + 0.5)    /* donne 1 */  
(unsigned char) (-1) /* donne 255 */
```

Conversions implicites de types

Conversions implicites : ajoutées par le C.

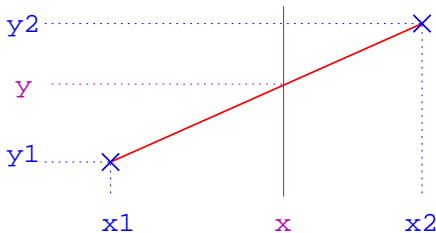
- lors des **affectations**, **initialisations** : vers le type de la *lvalue*,

Exemple

```
double d = 1;    conversion de 1 de int vers double  
int x = d+0.5;  troncature de d+0.5 de double vers int
```

- lors des appels de fonctions : vers le **type des arguments**,
Exemple : `fputc(2.5, f)`; (exception : `printf`)
- lors des opérations binaires : **promotion vers un type commun**,
`int < long < long long < float < double < long double`
Exemple : `int x=6; x=x*0.5;` multiplication en double.

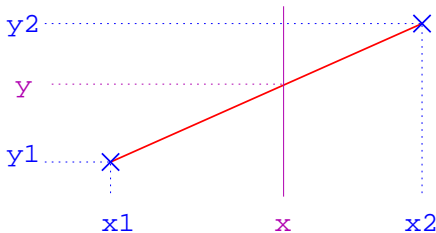
Exemple : interpolation linéaire



Calculs en entiers

```
int interpol(int x1, int y1, int x2, int y2, int x)
{
    return y1 + ((y2-y1) / (x2-x1)) * x;
}
```

Exemple : interpolation linéaire



Calculs en flottants

```
int interpol(int x1, int y1, int x2, int y2, int x)
{
    return y1 + ( (double) (y2-y1) / (x2-x1) ) * x;
}
```

Conversions de types pointeurs

On peut convertir d'un type pointeur en un autre type pointeur, avec la même syntaxe :

Syntaxe

`(type) expr`

Effet :

- ne change pas l'adresse pointée,
- change l'interprétation des opérateurs :
 - +, -, etc. décalent par unité de `sizeof(type)` octets,
 - * extrait un objet de type `type`.

Applications des conversions de types pointeurs

Deux applications principales :

- conversion avec le type générique `void*`,
Exemple : allocation de mémoire dynamique,
- conversion avec le type `unsigned char*`,
⇒ permet l'accès à la représentation mémoire en octets.

Exemple

```
int x = 386, i;  
unsigned char* p = (unsigned char*) &x;  
for ( i=0; i<sizeof(x); i++ ) printf("%i ", p[i]);  
donne sur un Intel 32-bit : 130 1 0 0.
```

Allocation dynamique de mémoire

Rappels sur la gestion de la mémoire

Variables : méthode la plus simple de réservation de mémoire.

Caractéristiques :

- variables globales : accessibles par tout le programme, variables locales : détruites en fin de bloc,
- la taille est fixée à la déclaration (par le **type**),
- tableaux globaux : la taille est une constante fixée “en dur”, tableaux locaux : la taille peut être une expression.

⇒ pas toujours suffisamment flexible !

Exemple : comment avoir un tableau

- dont la taille est calculée dans une fonction,
- dont la durée de vie dépasse celle de la fonction,
- dont la taille peut être changée *a posteriori*?

Les blocs de mémoire dynamique

Solution : **mémoire dynamique**.

- blocs de mémoire de taille arbitraire,
- alloués et libérés explicitement par l'utilisateur,
- redimensionnables (explicitement),
- accessibles par **pointeur**.

Tas (*heap*) = zone mémoire où est allouée la mémoire dynamique.

Allocation d'un bloc avec malloc

Syntaxe

```
#include <stdlib.h>  
void * malloc(size_t size);
```

Effet :

- alloue un nouveau bloc de mémoire de **size octets**,
- renvoie un pointeur sur le début (1ère octet) du bloc,
- renvoie **NULL** en cas d'erreur (mémoire insuffisante).

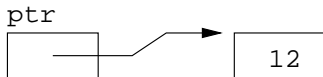
Comment utiliser malloc :

- calculer la taille à allouer grâce à **sizeof()**,
- vérifier que la valeur de retour n'est pas NULL,
- stocker le pointeur dans une **variable pointeur du bon type**,
- accéder au bloc grâce au pointeur : *****, **[]**.

Exemple d'allocation

Exemple

```
double* ptr = malloc( sizeof( double ) );  
assert( ptr!=NULL );  
*ptr = 12.0;  
*ptr += 2.0;
```



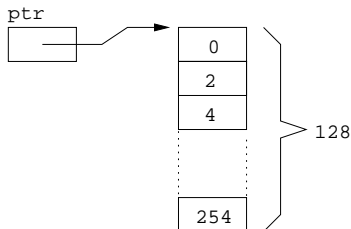
Notes :

- la conversion `void* → double*` est implicite,
- on utilise `assert` pour vérifier que `ptr` n'est pas `NULL`.

Exemple d'allocation de tableau

Exemple

```
int i;  
double* ptr = malloc( sizeof( double ) * 128 );  
assert(ptr);  
for ( i=0; i<128; i++ ) ptr[i] = i * 2.0;
```



Attention

Ne pas dépasser de la taille allouée !

Durée de vie d'un bloc

Le bloc reste alloué en mémoire jusqu'à être libéré explicitement.

Tableau local

```
int* truc(int n)
{
    int x[n];
    ...
    return &x[0];
}
```

FAUX!

Tableau dynamique

```
int* truc(int n)
{
    int* y = malloc(sizeof(int)*n);
    ...
    return y;
}
```

Correct.

Ne pas confondre

Durée de vie du bloc et durée de vie du pointeur y.

Allocation dynamique de chaînes

Fonction

```
#include <string.h>
char* strdup(const char* s);
```

Effet : renvoie une copie de `s` dans un bloc alloué dynamiquement, ou `NULL` (mémoire insuffisante).

Fonction équivalente à `strdup`

```
char* mon_strdup(const char* s)
{
    char* x = ???
    ???
    return x;
}
```

Allocation dynamique de chaînes

Fonction

```
#include <string.h>
char* strdup(const char* s);
```

Effet : renvoie une copie de `s` dans un bloc alloué dynamiquement, ou `NULL` (mémoire insuffisante).

Fonction équivalente à `strdup`

```
char* mon_strdup(const char* s)
{
    char* x = malloc( strlen(s) + 1 );
    if (x) strcpy( x, s );
    return x;
}
```

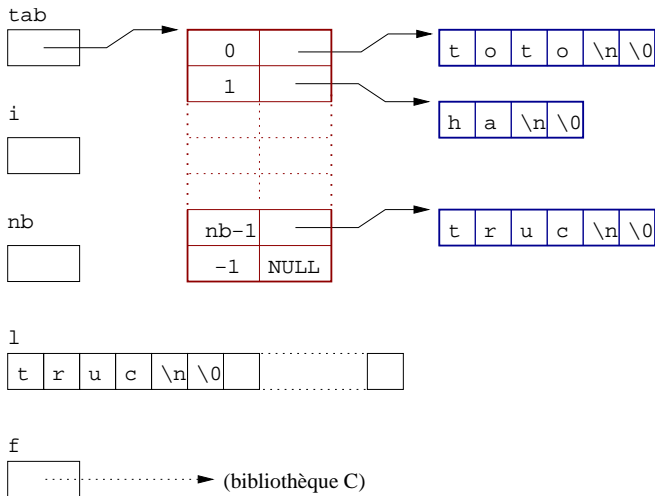
Exemple complexe

Exemple

```
typedef struct { int n; char* s; } ligne;

ligne* lit( FILE* f )
{
    char l[256];
    int i, nb;
    ligne* tab;
    fscanf( f, "%i", &nb );
    tab = malloc( sizeof(ligne) * (nb+1) );
    for ( i=0; i<nb; i++ ) {
        fgets( l, 256, f );
        tab[i].n = i; tab[i].s = strdup( l );
    }
    tab[i].n = -1; tab[i].s = NULL;
    return tab;
}
```

Illustration de l'exemple complexe



Libération d'un bloc avec free

La mémoire est libérée automatiquement en fin du programme.
On peut toutefois libérer un bloc manuellement :

Syntaxe

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

Effet :

- ptr doit pointer sur un bloc créé par malloc,
- après free, le bloc n'est plus utilisable,
- la mémoire libérée est recyclée et réutilisable par malloc.

Exemple d'utilisation de free

Exemple

```
typedef struct { int n; char* s; } ligne;  
  
void libere_tab( ligne* tab )  
{  
    int i;  
    for ( i=0; tab[i].nb > -1; i++ ) free( tab[i].s );  
    free( tab );  
}
```

Si on oublie de libérer `tab[i].s`, on a des **fuites de mémoire** !

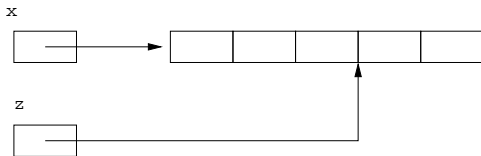
Notion d'*aliasing*

Aliasing

Pointeurs qui pointent dans le même bloc.

Exemple

```
int* x = malloc( sizeof(int) * 5 );  
int* z = x+3;  
free(x);  
*z = 12; /* erreur */
```



Redimensionnement de bloc avec realloc

Syntaxe

```
void* realloc(void *ptr, size_t size);
```

Effet :

- ptr doit pointer sur un bloc créé par malloc, (ou retournée par realloc),
- change la taille du bloc en size,
- renvoie la nouvelle adresse du bloc (ou NULL),
⇒ l'ancienne adresse ptr ne doit plus être utilisée,
- le contenu du bloc n'est pas changé.

Exemple d'utilisation de realloc

Exemple

```
int  taille = 128;
int* tab = malloc( sizeof(int) * taille );
...
if ( i >= taille ) {
    taille = i + 1;
    tab = realloc( tab, sizeof(int) * taille );
}
tab[i] = 12;
```

Note : il n'y a aucun moyen de connaître la taille d'un bloc,
⇒ on s'en souvient dans une variable annexe `taille`.

L'outil valgrind

valgrind = détecte (entres autres) les erreurs de mémoire :

- les accès en dehors des bornes d'un bloc dynamique,
- les free et realloc incorrects,
- les fuites de mémoire.

Utilisation

```
$ gcc toto.c -Wall -Wextra -g  
$ valgrind ./a.out
```

Notes :

- comme pour gdb, l'option de compilation `-g` est conseillée,
- contrairement à gdb, valgrind est non interactif.