Partie 2.2 - Le langage C

ECE_3TC31_TP/INF107

Guillaume Duc

2025

Expressions, suite

Typage et conversions implicites

Lorsque le type d'un opérande d'un opérateur n'est pas celui attendu, une conversion implicite est réalisée automatiquement.

Ces conversions sont standardisées. Les règles sont relativement complexes (et dépassent le cadre de ce cours) : https://en.cppreference.com/w/c/language/conversion

Opérateurs de conversion de type (*cast*)

Il est possible d'effectuer une conversion de type explicite :

'(' <Type> ')' <Expr>

- Convertit le résultat de l'expression vers le type indiqué
- Exemple

```c
float f = .5;
int main(int argc, char *argv[]) {
  return (int)f;     // Conversion explicite
}
```

https://en.cppreference.com/w/c/language/cast

Opérateur d'accès à un tableau

```
<Array> '[' <Index> ']'
```

- L'expression `Index` doit être de type entier
- Les éléments d'un tableau sont indexés à partir de 0
  - Pour un tableau de taille n, $n-1$ est l'indice du dernier élément
- Accéder à un index excédant la taille du tableau est un comportement non défini
- Exemple

```c
int array[] = {1, 2, 3, 4};
int main(int argc, char *argv[]) {
  return array[2];          // Accès à un tableau
}
```

https://en.cppreference.com/w/c/language/operator_member_access#Subscript

Opérateurs arithmétiques de base

Quelques particularités de `+`, `-` , `*`, `/` et `%`

- Dépassement (*overflow*)
  - Les opérations sur des entiers non signés sont effectuées modulo 2^n
  - Pour des opérations sur des entiers signés, le résultat est non défini
- La division entière est tronquée vers 0
  - La division par 0 est non définie
  - Le reste (`%`) est du même signe que le dividende
  - Exemple : $-11/3 = -3.666667$
    - `-11/3` vaut `-3`
    - `-11%3` vaut `-2`

https://en.cppreference.com/w/c/language/operator_arithmetic

Opérateurs de comparaison

Les opérateurs `==`, `!=`, `<`, `<=`, `>`, `>=` comparent les valeurs de leurs deux opérandes.

Si la comparaison est vraie, l'expression vaut 1, 0 sinon.

```
int a = (5 > 4);  // a = 1
int b = (1 == 2); // b = 0
```

Note : l'opérateur unaire NON logique est équivalent à un test par rapport à la zéro. `!a` est équivalent à `a == 0`.

⚠ L'opérateur de comparaison *égal* s'écrit `==` (avec 2 `=`).

https://en.cppreference.com/w/c/language/operator_comparison
https://en.cppreference.com/w/c/language/operator_logical

Opérateur d'affectation

⚠ L'opérateur d'affectation = ne doit pas être confondu avec l'opérateur de comparaison ==

```
<modifieable lvalue> = <expression>
```

- Assigne la valeur de l'expression de droite `<expression>` à `<lvalue>`
- L'expression `<lvalue>` doit être modifiable (variable, case d'un tableau…)
- Les types à gauche et à droite doivent être compatibles (une conversion implicite peut avoir lieu)
- L'expression complète `<modifieable lvalue> = <expression>` est évaluée à la valeur du membre de droite

```c
int a = (b = 2) + 3; // a = 5 et b = 2
```

https://en.cppreference.com/w/c/language/operator_assignment

Opérateurs d'affectation composés

`*=`, `/=`, `%=`, `+=`, `-=`, `<<=`, `>>=`, `&=`, `^=`, `|=`

- Ces opérateurs sont des raccourcis d'écriture
- Ils réalisent une opérations arithmétique ou logique dont le premier opérande est le membre de gauche et le second le membre de droite, puis le résultat de cette opération est affecté au membre de gauche
- Exemple : `a += b` est équivalent à `a = a + b`

```
int a = 1;
a += 1;     // a = a + 1 (= 2)
```

https://en.cppreference.com/w/c/language/operator_assignment

Opérateurs de pré/post incrémentation/décrémentation

(1) `a++` (2) `a--` (3) `++a` (4) `--a`

Pour les cas 1 et 3 la valeur de a est incrémentée de 1 et pour 2 et 4 elle est décrémentée de 1.

La différence réside dans la valeur de l'expression elle-même :

- 1 et 2 sont des opérateurs de **post**-incrémentation/décrémentation : l'expression `a++` vaut la valeur de a avant l'incrémentation
- 3 et 4 sont des opérateurs de **pré**-incrémentation/décrémentation : l'expression `a++` vaut la valeur de a après l'incrémentation

```
int a = 1;
int b = a++; // b = 1, a = 2
int c = ++a; // c = 3, a = 3
```

https://en.cppreference.com/w/c/language/operator_incdec

Appel de fonction

```
<Function name> '(' <Values of arguments> ')'
```

- Les valeurs des différents arguments (s'il y en a) sont séparées par des virgules
- Exemple

```
int add(int a, int b) { return a + b; }
void f(int c) {
   int a = add(2 * 3, c - 1); // Appel de la fonction add
}
```

https://en.cppreference.com/w/c/language/operator_other

Appel de fonction (2)

⚠ ⚠ **Ne confondez pas la syntaxe de définition d'une fonction et la syntaxe pour l'appeler !** ⚠ ⚠

Il s'agit, pour 3TC31, de l'une des erreurs les plus fréquentes en C !

Si la déclaration de la fonction add est

```
int add(int a, int b);
```

Pour l'appeler avec les valeurs 3 et 4, il suffit d'écrire (dans un contexte qui a du sens) :

```
add(3, 4); // Appel de la fonction add avec les arguments 3 et 4
```

⚠ Vous ne devez pas répéter le type de retour ou le type des arguments

```
int add(3, 4);          // FAUX !!!
int add(int 3, int 4); // FAUX !!!
add(int 3, int 4);      // FAUX !!!
```

Ordre d'évaluation

L'ordre d'évaluation des termes dans une expression n'est généralement pas spécifié, ce qui peut avoir un impact en cas d'effet de bord (exemple : appel de fonctions)

```
f1() + f2() * f3()
```

- Les règles de priorité et d'associativité permettent de déterminer l'ordre d'application de la multiplication et de l'addition : `f1() + (f2() * f3())`
- Les valeurs de retour de `f2()` et `f3()` sont nécessaires pour la multiplication
- La valeur de retour de `f1()` et le résultat de la multiplication sont nécessaires pour l'addition
- Dans quel ordre sont appelées les fonctions ? Ce n'est **pas spécifié**. Elles peuvent l'être dans n'importe quel ordre

https://en.cppreference.com/w/c/language/eval_order

Ordre d'évaluation : cas particulier non définis

Certains cas sont **non définis** (c'est mal, ne les écrivez pas)

- C'est le cas si un effet de bord sur une variable n'est pas ordonné via-à-vis d'un autre effet de bord sur la même variable.

```
i = ++i + i++;      // undefined behavior
i = i++ + 1;        // undefined behavior
f(++i, ++i);        // undefined behavior
f(i = -1, i = -1); // undefined behavior
```

- Ou lorsqu'un effet de bord sur une variable n'est pas ordonné avec un calcul utilisant la valeur de cette variable.

```
f(i, i++);  // undefined behavior
a[i] = i++; // undefined behavior
```

Opérateurs logiques (avec court-circuit)

`&&` et `||`

- `&&` retourne 1 si et seulement si ses deux opérandes sont différents de 0, sinon il retourne 0
- `||` retourne 0 si et seulement si ses deux opérandes égaux à 0, sinon il retourne 1
- L'ordre d'évaluation des opérandes est spécifié pour ces deux opérateurs
  - L'opérande de gauche est toujours évalué en premier
  - Pour `&&`, si ce membre de gauche vaut 0, l'opérande de droite **n'est pas** évalué et l'opérateur retourne 0
  - Pour `||`, si ce membre de gauche ne vaut pas 0, l'opérande de droite **n'est pas** évalué et l'opérateur retourne 1
- Exemple : `0 && f1()`, la fonction `f1()` n'est pas appelée

https://en.cppreference.com/w/c/language/operator_logical

Opérateur ternaire

`<Cond> '?' <Expr-true> ':' <Expr-false>`

- Si `<Cond>` est évaluée à une valeur différente de 0, l'opérateur retourne la valeur de `<Expr-true>`, sinon la valeur de `<Expr-false>`
- À noter : l'expression qui n'est pas utilisée n'est pas évaluée

```
a = (f == 1) ? f1() : f2();
// Équivalent à
if (f == 1) a = f1();
else a = f2();
```

https://en.cppreference.com/w/c/language/operator_other

Organisation mémoire

Organisation de la mémoire

Vu au premier cours : architecture Von Neumann, les instructions et les données d'un programme sont stockées dans la même mémoire.

On va donc trouver en mémoire :

- Les instructions du programme (code machine issu de la compilation du corps des fonctions)
- Les données du programme (toutes les variables)

Données

Concernant les données, celles-ci sont stockées dans trois zones distinctes

- Une zone pour toutes les variables globales (ou locales statiques)
  - Cette zone est subdivisée pour des raisons de protection et d'optimisation (cf. 4SE03)
  - Le compilateur (en pratique son *éditeur de liens*) calcule la taille de cette zone en fonction des déclarations des variables concernées. Elle est constante tout au long de l'exécution.
- Une zone nommée le **tas** (*heap*)
  - Le tas est la zone dans laquelle la mémoire allouée dynamiquement (avec la fonction `malloc`, voir prochain cours) est prise
  - Sa taille est nulle au début de l'exécution et varie au gré des allocations et désallocations explicites
  - Elle est gérée par la bibliothèque standard et le système d'exploitation
- Une zone nommée la **pile** (*stack*)
  - La pile permet de stocker toutes les valeurs temporaires liées à l'exécution des fonctions (arguments, variables locales, valeurs de retour)
  - Sa taille est nulle (ou presque) au début de l'exécution du programme et varie au gré des appels et retour de fonctions
  - Elle est gérée par des instructions ajoutées par le compilateur au début et à la fin des fonctions

Utilité de la pile (1)

```c
int f(int a) {
   int b = a - 1;
   if (b == 0)
      return 0;
   else
      return f(b) + 1;
}
```

Combien de mémoire faut-il réserver pour exécuter cette fonction ?

Réponse naïve

- 8 octets pour stocker la valeur de l'argument `a` (en supposant qu'un `int` fait 8 octets)
- 8 octets pour stocker la variable locale `b`
- 8 octets pour stocker la valeur de retour de la fonction

Utilité de la pile (2)

```c
int f(int a) {
   int b = a - 1;
   if (b == 0)
      return 0;
   else
      return f(b) + 1;
}
```

- Mais à cause de la récursion, à un instant donné, plusieurs appels imbriqués à `f` peuvent être en cours d'exécution.
- Chacun de ces appels a ses propres arguments, sa propre variable locale `b` et sa propre valeur de retour.
- Dans le cas général, le compilateur ne peut donc pas savoir à la compilation combien de mémoire réserver pour l'exécution de cette fonction.

La pile

Les données liées à l'exécution des fonctions sont donc en général stockées dans une zone de la mémoire nommée **la pile**.

La taille de cette zone croit lors de l'appel d'une fonction pour y stocker les informations liées à son exécution. Elle décroît une fois la fonction terminée supprimant ainsi les données qui ne sont plus utiles.

Ce sont des instructions insérées par le compilateur au début et à la fin des fonctions, ainsi qu'au niveau des appels de fonction qui permettent de gérer cette zone.

On va trouver classiquement (les détails peuvent varier en fonction des architectures et des optimisations)

- La valeur des arguments
- Les variables locales (sauf celles `static`)
- Des informations sauvegardées pour permettre la reprise de l'exécution à la fin de la fonction (par exemple l'adresse de retour)
- La valeur de retour de la fonction

La pile : exemple simplifié (1)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);
    return e;
}

int main(int argc, char *argv[]) {
    // <-- Ici
    int t = f(4, 5);
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à l'entrée de la fonction `main`

Adresse de la pile	Valeur	Description
7fffd97c	xxxxxxxx	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
	...	

La pile : exemple simplifié (2)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);    // <-- Ici
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile lors de la préparation de l'appel à la fonction `f`

Adresse de la pile	Valeur	Description
7fffd970	00000005	Deuxième argument de `f`
7fffd974	00000004	Premier argument de `f`
7fffd978	0007f354	Adresse de retour (dans `main`)
7fffd97c	xxxxxxxx	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
	…	

La pile : exemple simplifié (3)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    // <-- Ici
    int e;
    e = g(c, d);
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à l'entrée de la fonction `f`

Adresse de la pile	Valeur	Description
7fffd96c	xxxxxxxx	Variable locale `e`
7fffd970	00000005	Valeur de `d`
7fffd974	00000004	Valeur de `c`
7fffd978	0007f354	Adresse de retour (dans `main`)
7fffd97c	xxxxxxxx	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
		…

La pile : exemple simplifié (4)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);      // <-- Ici
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à lors de la préparation de l'appel à la fonction `g`

Adresse de la pile	Valeur	Description
7fffd960	00000005	Deuxième argument de `g`
7fffd964	00000004	Premier argument de `g`
7fffd968	0007f498	Adresse de retour (dans `f`)
7fffd96c	xxxxxxxx	Variable locale `e`
7fffd970	00000005	Valeur de `d`
7fffd974	00000004	Valeur de `c`
7fffd978	0007f354	Adresse de retour (dans `main`)
7fffd97c	xxxxxxxx	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
	…	

La pile : exemple simplifié (5)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    // <-- Ici
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à l'entrée de la fonction `g`

Adresse de la pile	Valeur	Description
7fffd960	00000005	Valeur de b
7fffd964	00000004	Valeur de a
7fffd968	0007f498	Adresse de retour (dans f)
7fffd96c	xxxxxxxx	Variable locale e
7fffd970	00000005	Valeur de d
7fffd974	00000004	Valeur de c
7fffd978	0007f354	Adresse de retour (dans main)
7fffd97c	xxxxxxxx	Variable locale t
7fffd980	7fffda88	Valeur de argv
7fffd984	00000001	Valeur de argc
7fffd988	0007f29d	Adresse de retour
		…

La pile : exemple simplifié (6)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);
    // <-- Ici
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à la sortie de la fonction `g`

Adresse de la pile	Valeur	Description
7fffd96c	00000014	Variable locale `e`
7fffd970	00000005	Valeur de `d`
7fffd974	00000004	Valeur de `c`
7fffd978	0007f354	Adresse de retour (dans `main`)
7fffd97c	xxxxxxxx	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
		…

La pile : exemple simplifié (7)

```c
#include <stdio.h>
#include <stdlib.h>

int g(int a, int b) {
    return a * b;
}

int f(int c, int d) {
    int e;
    e = g(c, d);
    return e;
}

int main(int argc, char *argv[]) {
    int t = f(4, 5);
    // <-- Ici
    printf("%d\n", t);
    return EXIT_SUCCESS;
}
```

État de la pile à la sortie de la fonction `f`

Adresse de la pile	Valeur	Description
7fffd97c	00000014	Variable locale `t`
7fffd980	7fffda88	Valeur de `argv`
7fffd984	00000001	Valeur de `argc`
7fffd988	0007f29d	Adresse de retour
	...	

Aparté : passage par valeur des arguments (1)

Lors d'un appel de fonction, les valeurs des arguments sont placées sur la pile (les détails peuvent varier en fonction des architectures…). La fonction appelée a accès (et peut modifier) ces valeurs sur la pile.

Lors de la fin d'une fonction, ces arguments sont supprimés de la pile. Toute modification faite par la fonction sur ses arguments est donc perdue. La fonction appelante n'a pas été affectée.

Les arguments des fonctions sont passés **par valeur**.

Aparté : passage par valeur des arguments (2)

```c
void swap(int a, int b) {
    // Échange les valeurs de a et b
    int c = b;
    b = a;
    a = c;
}

int main(int argc, char *argv[]) {
    int a = 1;
    int b = 2;
    swap(a, b);
    // a vaut toujours 1, b vaut toujours 2
}
```

⚠ C'est un piège classique pour les débutants.

Comment résoudre le problème (élégamment) ? En utilisant des pointeurs, voir plus loin…

Adresses, tailles et alignement

Chaque objet (fonction, variable…) a une adresse qui lui est assigné par le compilateur (plus précisément son éditeur de liens).

Ces adresses sont globales pour les fonctions et les variables globales. Elles sont relatives au sommet de la pile pour les variables locales.

Ces adresses ne changent pas durant la vie de l'objet.

De plus, chaque objet a une **taille** connue.

Enfin, chaque objet a une notion d'**alignement** : son adresse doit être multiple de cet alignement.

https://en.cppreference.com/w/c/language/object#Alignment

Opérateurs `sizeof` et `_Alignof`

Ces opérateurs permettent de déterminer la taille et l'alignement

- **Alignement :** `'_Alignof' '(' <Type> ')'` Retourne l'alignement minimum d'un type
  - Alternative `alignof` (requiert `#include <stdalign.h>`)
- **Taille :** `'sizeof' '(' <Type> ')'` or `'sizeof' <Expr>` Retourne la taille d'un type ou d'une expression
- Le type de retour de ces deux opérateurs est `size_t`.

https://en.cppreference.com/w/c/language/sizeof
https://en.cppreference.com/w/c/language/_Alignof
https://en.cppreference.com/w/c/types/size_t

Taille et alignement : exemple (1)

```c
#include <stdalign.h>
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("char:\t\talignment:%zd size:%zd\n", _Alignof(char), sizeof(char));
  printf("short:\t\talignment:%zd size:%zd\n", _Alignof(short), sizeof(short));
  printf("int:\t\talignment:%zd size:%zd\n", _Alignof(int), sizeof(int));
  printf("long:\t\talignment:%zd size:%zd\n", _Alignof(long), sizeof(long));
  printf("long long:\talignment:%zd size:%zd\n", _Alignof(long long), sizeof(long long));
  printf("float:\t\talignment:%zd size:%zd\n", _Alignof(float), sizeof(float));
  printf("double:\t\talignment:%zd size:%zd\n", _Alignof(double), sizeof(double));
  printf("long double:\talignment:%zd size:%zd\n", _Alignof(long double), sizeof(long double));

  return EXIT_SUCCESS;
}
```

size-alignment.c.

Taille et alignement : exemple (2)

Sur une machine de TP (Linux, x86-64) :

```
tp-5b07-26:~/tmp> ls
size-alignment.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g size-alignment.c -o size-alignment

tp-5b07-26:~/tmp> ls
size-alignment size-alignment.c

tp-5b07-26:~/tmp> size-alignment
char:           alignment:1 size:1
short:          alignment:2 size:2
int:            alignment:4 size:4
long:           alignment:8 size:8
long long:      alignment:8 size:8
float:          alignment:4 size:4
double:         alignment:8 size:8
long double:    alignment:16 size:16
```

Types dérivés

Types dérivés (*Derived Types*)

En plus des types de base vus précédemment, le C permet de construire des types dérivés :

- Pointeurs
- Énumérations
- Structures

Pointeurs

Un pointeur contient l'**adresse en mémoire d'un autre objet**.

La déclaration du pointeur indique le type de l'objet pointé.

```
int *pointerToInt; // Un pointeur vers un int
void *pointer;     // Un pointeur vers un type non spécifié
```

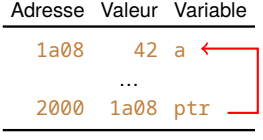
⚠ La définition d'une variable de type pointeur comme ci-dessus alloue de la mémoire pour stocker la valeur du pointeur (une adresse) mais pas de mémoire pour l'objet pointé par ce pointeur ! ⚠

https://en.cppreference.com/w/c/language/pointer

Opérateur & (adresse de)

L'opérateur unaire & permet de récupérer l'adresse d'un objet.

```c
int a = 0x42;  // Définit une variable a de type int
int *ptr = &a; // Définit un pointeur vers un int
               //  et initialise sa valeur à l'adresse de a
```



La valeur retournée est de type pointeur vers le type de l'objet. Exemples :

- Appliqué sur un objet de type `int`, la valeur retournée sera de type `int *`
- Appliqué sur un objet de type `float *`, la valeur retournée sera de type `float **` (type pointeur vers un pointeur vers un `float`)

https://en.cppreference.com/w/c/language/operator_member_access#Address_of

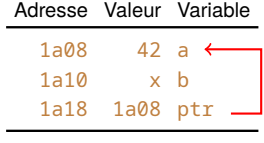
Opérateur * : déréférencement (1)

L'opérateur * est l'opérateur de **déréférencement** :

'*' <Pointer-Expression>

- <Pointer-Expression> doit être une expression de type pointeur
- Considère la valeur de l'expression <Pointer-Expression> comme une adresse et accède, en lecture ou en écriture, à la mémoire située à cette adresse

```
int a, b;       // Définit deux variables de type int
a = 0x42;       // Stocke 0x42 dans la variable a
int *ptr = &a; // Définit un pointeur vers un int
                //  et initialise sa valeur à l'adresse de a
b = *ptr;       // Lit l'int stocké à l'adresse contenu dans ptr
                //  et affecte cette valeur à b (0x42)
*ptr = 0x43;    // Stocke la valeur 0x43 à l'adresse contenu
                //  dans ptr
```



https://en.cppreference.com/w/c/language/operator_member_access#Dereference

Opérateur * : déréférencement (2)

L'opérateur * est l'opérateur de **déréférencement** :

'*' <Pointer-Expression>

- <Pointer-Expression> doit être une expression de type pointeur
- Considère la valeur de l'expression <Pointer-Expression> comme une adresse et accède, en lecture ou en écriture, à la mémoire située à cette adresse

```
int a, b;       // Définit deux variables de type int
a = 0x42;       // Stocke 0x42 dans la variable a
int *ptr = &a; // Définit un pointeur vers un int
                //  et initialise sa valeur à l'adresse de a
b = *ptr;       // Lit l'int stocké à l'adresse contenu dans ptr
                //  et affecte cette valeur à b (0x42)
*ptr = 0x43;    // Stocke la valeur 0x43 à l'adresse contenu
                //  dans ptr
```

Adresse	Valeur	Variable
1a08	42	a
1a10	42	b
1a18	1a08	ptr

https://en.cppreference.com/w/c/language/operator_member_access#Dereference

Opérateur * : déréférencement (3)

L'opérateur * est l'opérateur de **déréférencement** :

'*' <Pointer-Expression>

- <Pointer-Expression> doit être une expression de type pointeur
- Considère la valeur de l'expression <Pointer-Expression> comme une adresse et accède, en lecture ou en écriture, à la mémoire située à cette adresse

```
int a, b;       // Définit deux variables de type int
a = 0x42;       // Stocke 0x42 dans la variable a
int *ptr = &a; // Définit un pointeur vers un int
                //  et initialise sa valeur à l'adresse de a
b = *ptr;       // Lit l'int stocké à l'adresse contenu dans ptr
                //  et affecte cette valeur à b (0x42)
*ptr = 0x43;    // Stocke la valeur 0x43 à l'adresse contenu
                //  dans ptr
```

Adresse	Valeur	Variable
1a08	43	a
1a10	42	b
1a18	1a08	ptr

https://en.cppreference.com/w/c/language/operator_member_access#Dereference

Opérateur * : pièges classiques (1)

⚠ **Écrire** `ptr = xxx` **change le pointeur lui-même** (la valeur du pointeur) :

```
int a, b;
int *ptr = &a; // ptr pointe vers a (sa valeur est l'adresse de a)
ptr = &b; // ptr pointe mintenant vers b (sa valeur est l'adresse de b)
          // a n'a pas changé de valeur
```

⚠ **Écrire** `xxx = ptr` **accède à la valeur du pointeur** :

```
int a;
int *ptr = &a; // ptr pointe vers a (sa valeur est l'adresse de a)
int *ptr2 = ptr; // la valeur de ptr2 est celle de ptr (l'adresse de a)
                 // ptr2 pointe donc également vers a
```

⚠ **Écrire** `xxx = *ptr` **déréférence le pointeur** (accède à la case mémoire dont l'adresse est la valeur du pointeur) :

```
int a = 0x42;
int *ptr = &a; // ptr pointe vers a (sa valeur est l'adresse de a)
int b = *ptr;  // déréférence le pointeur ptr, b prend la valeur 0x42
b = 0x43;      // b prend la valeur 0x43, a n'est pas modifié (0x42)
```

Opérateur * : pièges classiques (2)

⚠ **Vous ne pouvez déréférencer un pointeur que s'il contient une adresse autorisée.** Pour simplifier :

- L'adresse d'une variable allouée par le compilateur (globale ou locale si respect de sa durée de vie)
- L'adresse d'une zone allouée dynamiquement par vos soins (voir prochain cours sur `malloc`)

```c
int *ptr;    // Déclaration de ptr mais pas de valeur initiale !
*ptr = 0x42; // NON !!! Vous ne savez pas vers quoi pointe ptr
// ...
int *ptr2 = (int *)1024; // Déclare un pointeur valant 1024
                         //  i.e. pointant vers l'adresse 1024
*ptr2 = 0x42; // NON !!! Sauf si vous savez exactement ce que vous faites
```

Opérateur * : type et pointeurs void *

`*ptr` est du type de l'objet pointé par `ptr`. Exemples :

- Si `ptr` est de type `int *`, `*ptr` est de type `int`
- Si `ptr` est de type `float **`, `*ptr` est de type `float *`

Vous ne pouvez pas déréférencer une expression de type `void *`.

NULL

Il peut être utile d'indiquer qu'un pointeur ne contient pas d'adresse valide.

La macro NULL peut être utilisée pour cela :

```
int *ptr = NULL;
```

NULL est définie dans plusieurs fichiers d'en-tête de la bibliothèque standard (`#include <stddef.h>`, `#include <stdlib.h>`...).

La valeur de NULL dépend de l'implémentation (mais le plus souvent vaut `0`).

⚠ Un pointeur non initialisé n'est pas automatiquement initialisé à NULL.

https://en.cppreference.com/w/c/types/NULL

Aparté : passage par adresse des arguments (1)

Comment corriger notre fonction `swap` ?

```c
void swap(int a, int b) {
    // Échange les valeurs de a et b
    int c = b;
    b = a;
    a = c;
}

int main(int argc, char *argv[]) {
    int a = 1;
    int b = 2;
    swap(a, b);
    // a vaut toujours 1, b vaut toujours 2
}
```

Aparté : passage par adresse des arguments (2)

`swap` va maintenant prendre non pas la valeur des arguments mais les adresses auxquelles se trouvent les deux entiers à échanger.

```c
void swap(int *a, int *b) {
    int c = *b;
    *b = *a;
    *a = c;
}

int main(int argc, char *argv[]) {
    int a = 1;
    int b = 2;
    swap(&a, &b);
    // a vaut maintenant 2, b vaut 1
}
```

C'est un passage d'argument **par adresse**.

Pointeurs et tableaux

Les tableaux et les pointeurs sont proches en C.

Notamment :

- Le nom d'un tableau se convertit naturellement si besoin en un pointeur **constant** vers son premier élément (un pointeur dont la valeur constante est l'adresse du premier élément du tableau)
- L'opérateur `[ ]` d'accès à un élément d'un tableau peut être utilisé sur un pointeur

Exemple : Pointeurs et tableaux (1)

```c
int tab[] = {1, 2, 3, 4, 5};

int *ptr = &tab[0];
int *ptr = tab;
```

Les deux déclarations de `ptr` sont équivalentes. `ptr` est un pointeur et sa valeur est l'adresse de la première case du tableau (qui contient `1`).

```c
tab[2] = 42;
ptr[2] = 42;
```

Ces deux statements sont identiques. Ils modifient tous les deux la troisième case du tableau `tab` (celle qui contenait la valeur `3`).

Exemple : Pointeurs et tableaux (2)

```c
int a;
int tab[] = {1, 2, 3, 4, 5};
int *ptr = tab;

ptr = &a; // OK
tab = &a; // NON !
```

`tab` se convertit naturellement en un pointeur **constant** vers son premier élément. La valeur de ce pointeur n'est donc pas modifiable.

Chaînes de caractères

Les chaînes de caractères en C sont composés de caractères qui se suivent en mémoire (comme un tableau).

Contrairement à certains langages (comme Java, Python, Rust…), la taille de la chaîne (le nombre de caractères) n'est pas stockée explicitement.

La fin de la chaîne est indiquée par un caractère spécial, le caractère **null** (à ne pas confondre avec le pointeur `NULL`), de valeur 0, souvent écrit `'\0'`).

Exemple

```c
const char messagePointer[] = "Hello World"; // <-- '\0' implicite à la fin
```

En mémoire :

'H'	'e'	'l'	'l'	'o'	' '	'W'	'o'	'r'	'l'	'd'	'\0'

ou en décimal :

72	101	108	108	111	32	87	111	114	108	100	0

Structures

Une structure permet de regrouper plusieurs valeurs (avec des types potentiellement différents) en un seul nouveau type dédié.

```c
struct coord {
  float x;
  float y;
};
```

Cette déclaration introduit un nouveau type `struct coord`. Pour le moment, aucune variable de ce type n'est encore créée.

⚠ Attention, le nom du type est bien `struct coord` et non pas simplement `coord`.

https://en.cppreference.com/w/c/language/struct

Structures : initialisation

Il est possible d'initialiser les membres de la structure lors de la définition

```
struct coord c1 = { 1.0, 2.0 };
struct coord c2 = { .x = 1.0, .y = 2.0 };
```

Structures : accès à un membre

L'opérateur `.` est utilisé pour accéder à un membre d'une structure.

```
struct coord c1;

c1.x = 1.0;
c1.y = 2.0;
```

Structures : accès à un membre via un pointeur (1)

Il est souvent utile d'accéder à un membre d'une structure à partir d'un pointeur vers cette structure. Il faut donc déréférencer le pointeur, puis utiliser l'opérateur `.` vu précédemment.

```
struct coord c1 = { 1.0, 2.0 };
struct coord *ptr = &c1;

(*ptr).y = 3.0;
```

Les parenthèses sont obligatoires à cause des priorités respectives des opérateurs `*` et `.`.

Structures : accès à un membre via un pointeur (2)

La construction précédente est peu élégante. Il existe un opérateur `->` qui permet de réaliser le déréférencement et l'accès à un membre.

Exemple

```
struct coord c1 = { 1.0, 2.0 };
struct coord *ptr = &c1;

ptr->y = 3.0; // Équivalent à (*ptr).y = 3.0
```

Énumérations `enum`

Les énumérations permettent de créer un nouveau type pouvant prendre une valeur parmi un ensemble de constantes nommées.

`'enum' <Identifier> '{' <Enumerators> '}'`

`<Enumerators>` sont séparés par des virgules et peuvent être de deux formes :

(1) `<Identifier>`

(2) `<Identifier> '=' <Constant-Expression>`

- Chaque `<Enumerator>` introduit un nouvel identifiant
  - Cet identifiant doit être unique
  - Il est associé à une valeur numérique
  - Si aucune valeur n'est fournie
    - La valeur est celle de l'identifiant précédent plus 1
    - Ou 0 si c'est le premier

⚠ Comme pour les structures, le type est `enum identifier` et non simplement `identifier`.

https://en.cppreference.com/w/c/language/enum

Énumérations enum : exemple

```c
enum color {RED, BLUE, GREEN};

enum color c1;
c1 = RED;

if (c1 == GREEN) { /* ... */ }
```

Définition de type

Il est possible de créer un alias pour un type existant.

`'typedef' <Type> <Identifier>`

- Introduit un nouvel identifiant (`<Identifier>`) qui sera utilisé comme alias pour le type `<Type>`.
- Les deux types seront synonymes.
- Cela est souvent utilisé pour raccourcir le nom d'un type structure ou énumération.

Exemple :

```c
struct coord { float x; float y; };
typedef struct coord coord_s;

// coord_s est maintenant un alias pour struct coord

coord_s c1;
c1.x = 3.0;
```

https://en.cppreference.com/w/c/language/typedef

Listes chaînées

Liste chaînées (1)

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct node node_t;                    // Declare shortcut.
struct node {
  void *data;
  node_t *next;                                // Use shortcut.
};

int main(int argc, char *argv[]) {
  static int data[] = {4, 067, 0xfffffff9};
  node_t node2 = {&data[2], NULL};             // Null pointer marks list end.
  node_t node1 = {(void*)&data[1], &node2};    // Explicit pointer cast int* to void*.
  node_t node0 = {&data[0], &node1};           // Explicit cast not needed for void*.

  for (node_t *tmp = &node0; tmp; tmp = tmp->next) {   // Address of operator.
    int *ptr = (int*)tmp->data;                // Dereferencing and member access + pointer cast.
    printf("%d\n", *ptr);                      // Dereferencing.
  }
  return EXIT_SUCCESS;
}
```

linked-list.c

Liste chaînées (2)

```
tp-5b07-26:~/tmp> ls
linked-list.c

tp-5b07-26:~/tmp> gcc -Wall -pedantic -std=c11 -O0 -g linked-list.c -o linked-list

tp-5b07-26:~/tmp> ls
linked-list  linked-list.c

tp-5b07-26:~/tmp> ./linked-list
4
55
-7
```

Annexes

Crédits

- Le cours se base sur celui écrit par Florian Brandner pour 3TC31