INF3135
Construction et maintenance de logiciels

Chapitre 1: Bases du langage C

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Types

Types numériques

Valeurs entières

- `char`: $\geq$ **1 octet**
- `short`: $\geq$ **2 octets**
- `int`: **2** ou **4 octets**
- `long`: $\geq$ **4 octets**
- `long long`: $\geq$ **8 octets**
- **Variantes:** `signed` (par défaut) ou `unsigned`

Valeurs flottantes

- `float`: $\geq$ **4 octets**
- `double`: $\geq$ **8 octets**
- `long double`: **10**, **12** ou **16 octets**
- Toujours **signées**

Exemple (entiers signés)

```c
#include <stdio.h>

int main(void) {
    char  c = '+';
    short s = 67;
    int   i = -1;
    long  l = -28;
    // %c: character, %d: signed decimal, %l: long
    printf("c = %c %d\n", c, c);
    printf("s = %c %d\n", s, s);
    printf("i = %d\n", i);
    printf("l = %ld\n", l);
    return 0;
}
```

Résultat:

```
c = + 43
s = C 67
i = -1
l = -28
```

Exemple (entiers non signés)

```c
#include <stdio.h>

int main(void) {
    unsigned char  c = -1;
    unsigned short s = -1;
    unsigned int   i = -1;
    unsigned long  l = -1;
    // %d: signed decimal, %u: unsigned decimal, %l: long
    printf("c = %d %u\n", c, c);
    printf("s = %d %u\n", s, s);
    printf("i = %d %u\n", i, i);
    printf("l = %ld %lu\n", l, l);
    return 0;
}
```

Résultat:

```
c = 255 255
s = 65535 65535
i = -1 4294967295
l = -1 18446744073709551615
```

Type booléen

Avant C99

- Pas de type booléen **natif**
- 0: **faux**
- $\neq$0: **vrai**

Depuis C99

- Ajout de la bibliothèque `stdbool.h`
- Définit le **type** `bool` (entier non signé)
- Et définit les **constantes** `true` et `false`

```
#define true 1
#define false 0
```

Exemple (booléens)

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

int main(void) {
    int  u = 0;
    char c = 'A';
    bool t = true;
    bool f = false;
    if (u) printf("u ");
    if (c) printf("c ");
    if (t) printf("t ");
    if (f) printf("f ");
    if (c && t) printf("c&&t ");
    if (c == t) printf("c==t\n");
    return 0;
}
```

Résultat:

```
c t c&&t
```

Types énumératifs

- Une des façons de définir des **constantes**
- Par **défaut**: valeurs 0, 1, 2, etc.
- Une variable de type `enum` est traitée comme un `int`
- **Aucune vérification** n'est faite

```c
#include <stdio.h>

enum day {MON, TUE, WED, THU, FRI, SAT, SUN};
enum http_code {
    HTTP_CONTINUE          = 100, HTTP_OK              = 200,
    HTTP_MULTIPLE_CHOICES  = 300, HTTP_BAD_REQUEST     = 400,
    HTTP_FORBIDDEN         = 403, HTTP_NOT_FOUND       = 404
};

int main(void) {
    enum day d1 = MON, d2 = SAT;
    enum http_code code = HTTP_NOT_FOUND;
    printf("%d %d %d\n", d1, d2, code);
}
```

Résultat:

```
0 5 404
```

Types complexes

Tableaux

- Permet de **concaténer** plusieurs valeurs de même type
- Les types doivent être **homogènes**

Structures (produit)

- Permet de **concaténer** des types
- Les types peuvent être **hétérogènes**

Unions (coproduit)

- Permet de proposer une **alternative** entre types
- Les types peuvent être **hétérogènes**

Autres types (plus tard)

Type vide

- Identifié par le mot `void`
- Définit le type d'une fonction **sans valeur** de retour
- Aussi la valeur **nulle** pour les pointeurs
- On va y **revenir** plus tard

Pointeurs

- `char*`: pointeur vers `char`
- `int*`: pointeur vers `int`
- `int**`: pointeur vers `int*`
- `double***`: pointeur vers `double**`
- `void*`: pointeur « générique »
- On va y **revenir** plus tard

Synonymes

- À l'aide de l'instruction `typedef`
- Aucune **vérification**

```
// Déclaration des synonymes
typedef unsigned int jour;
typedef float distance;
typedef enum {PIQUE, COEUR, CARREAU, TREFLE} couleur;

// Utilisation
jour lundi = 0, mardi = 1, samedi = 5;
distance d = 123.4;

void afficher_couleur(couleur c) {
    switch (c) {
        case PIQUE:   printf("pique");   break;
        case COEUR:   printf("coeur");   break;
        case CARREAU: printf("carreau"); break;
        case TREFLE:  printf("trefle");  break;
    }
}
```

Variables et constantes

Affectation

Syntaxe

Expression de la forme `L = R;`

Left-value ou *lvalue*

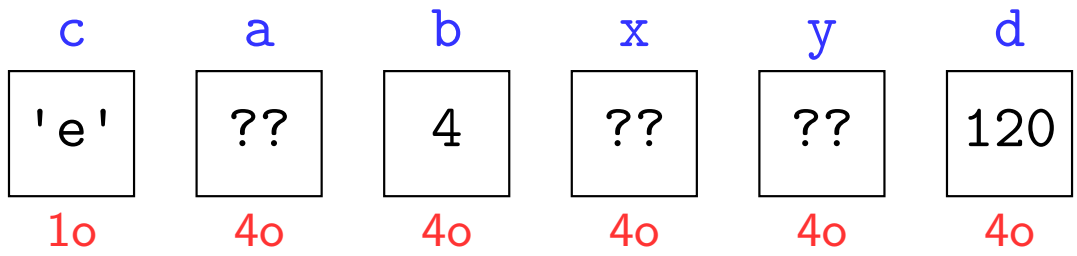
- Toute **expression** qui peut être placée à **gauche** de l'opérateur =
- Doit avoir un espace mémoire **réservé**
- **Exemples**: variable, champ d'une structure ou d'une union, pointeur

Right-value ou *rvalue*

- Toute autre **expression** valide
- Ayant une **valeur**

Variables

```
char c = 'e';
int a, b = 4;
float x, y;
unsigned int d = factorial(5);
```



Règles

- **Déclarée** avant son utilisation
- **Visible** seulement dans le bloc où elle est déclarée
- Peut être **initialisée** lors de la déclaration
- **Non initialisée** a une valeur indéterminée

Type de variables (1/2)

Automatiques (`auto`)

- Aussi appelées variables **locales**
- **Portée**: attachée au bloc qui la contient
- Mémoire **allouée** à la déclaration
- Et **libérée** à la fin du bloc
- Mot réservé `auto` **optionnel**

Statiques (`static`)

- **Portée**: attachée à la fonction qui la contient
- Mot réservé `static` **requis**
- Mémoire **permanente**: **allouée** au début du programme
- Et **libérée** à la fin du programme
- On va y revenir **plus tard**

Type de variables (2/2)

Externes (`extern`)

- Aussi appelées variables **globales**
- **Portée**: tout le fichier qui la contient
- Les **fonctions** sont externes par défaut
- Mot réservé `extern` **optionnel**
- Permet de communiquer entre plusieurs **modules**
- Mémoire **allouée** dans un seul module
- Mais **déclarations multiples** permises
- Utile pour **interfacer** avec d'autres langages
- On va y revenir **plus tard**

Constantes

Trois mécanismes:

- `#define`: instruction au préprocesseur pour définir une **macro**
- `const`: **empêche** de modifier une variable
- `enum`: définition d'un type **énumératif**

```
#define PI 3.141592654

const float PI = 3.141592654;

enum sign {
    NEG  = -1,
    ZERO = 0,
    POS  = 1
};
```

Valeurs littérales

```c
unsigned int ui = 34u;
long l = 34L;
char c = 52, d = 064, e = 0X34, f = '4';
// %u: unsigned decimal, %l: long, %d: decimal
printf("%u %ld\n", ui, l);
printf("%d %d %d %d\n", c, d, e, f);
// Affiche :
// 34 34
// 52 52 52 52
```

- **Suffixe** `u` ou `U`: valeur **non signée**
- **Suffixe** `l` ou `L`: valeur **longue**
- **Préfixe** `0`: valeur **octale**
- **Préfixe** `0x` valeur **hexadécimale**

$$\begin{aligned} 064 &= 6 \times 8^1 + 4 \times 8^0 = 52 \\ 0X34 &= 3 \times 16^1 + 4 \times 16^0 = 52 \\ '4' &= 52 \quad \text{(code ASCII)} \end{aligned}$$

Caractères spéciaux

Quelques caractères utiles:

- `\n`: fin de ligne
- `\t`: tabulation
- `\\`: contre-oblique
- `\'`: apostrophe
- `\"`: guillemets

```c
#include <stdio.h>

int main(void) {
    char c = '\'';
    printf("\"\\\t%c\"\n", c);
}
```

Résultat:

```
"\	'"
```

Structures de contrôle

Instruction `for`

```
for (<initialisation>; <condition>; <incrementation>) {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}
```

- `<initialisation>` est évaluée une seule fois, au début
- `<condition>` est évaluée au début de chaque tour de boucle

$$\texttt{<condition>} \text{ est considérée } \begin{cases} \textbf{faux}, & \text{si } \texttt{<condition> == 0;} \\ \textbf{vrai}, & \text{sinon.} \end{cases}$$

- `<incrémentation>` est évaluée à la fin de chaque tour de boucle

Déclaration et initialisation (1/2)

– On ne peut déclarer le type de l'itérateur dans l'**initialisation** d'une boucle `for` avec le standard **ANSI**:

```c
/* Fichier for.c */
#include <stdio.h>

int main(void) {
    for (unsigned int i = 0; i < 10; ++i) {
        printf("%d ", i);
    }
    return 0;
}
```

```
$ gcc -ansi for.c
code/for.c: In function ‘main’:
code/for.c:5:5: error: ‘for’ loop initial declarations
    are only allowed in C99 or C11 mode
[...]
```

Déclaration et initialisation (2/2)

Deux corrections possibles:

– Compiler selon un **standard** plus récent:

```
$ gcc -std=c99 for.c
$ gcc -std=c11 for.c
```

– Ou **réécrire** le programme:

```c
/* Fichier for-ansi.c */
#include <stdio.h>

int main(void) {
    unsigned int i;
    for (i = 0; i < 10; ++i) {
        printf("%d ", i);
    }
    return 0;
}
```

Instructions `if`, `else if` et `else`

```
if (<condition>) {
    <suite instructions>
}

if (<condition>) {
    <suite instructions 1>
} else {
    <suite instructions 2>
}

if (<condition 1>) {
    <suite instructions 1>
} else if (<condition 2>) {
    <suite instructions 2>
}
```

- Branchement `else` optionnel
- **Accolades** optionnelles si instruction **unique**
- Attention aux structures **fortement imbriquées**

Instruction `switch`

```
switch (<variable>) {
    case <valeur 1> : <instruction 1>
    case <valeur 2> : <instruction 2>
    ...
    case <valeur n> : <instruction n>
    default : <instruction n + 1>
}
```

- Chaque expression `case` est examinée dans l'**ordre**
- Jusqu'à **correspondance** ou jusqu'à `default`
- L'instruction est alors **exécutée**
- Ainsi que toutes les instructions **suivantes**
- **Tant que** le mot réservé `break` n'est pas rencontré
- Le cas `default` est **optionnel**

Exemple avec switch

```c
#include <stdio.h>

void print_case(char c) {
    switch (c) {
        case 'A': printf("A");
        case 'B': printf("B"); break;
        case 'C': printf("C");
        default : printf("default");
    }
    printf("\n");
}

int main(void) {
    print_case('A'); print_case('B');
    print_case('C'); print_case('D');
    return 0;
}
```

Résultat:

```
AB
B
Cdefault
default
```

Boucles `while` et `do-while`

Syntaxe:

```
while (<condition>) {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
}

do {
    <instruction 1>
    <instruction 2>
    ...
    <instruction n>
} while (<condition>);
```

- `break` permet de sortir de la boucle courante
- `continue` permet de passer immédiatement à l'itération suivante
- Utiliser `break`/`continue` seulement si simplifie la **lecture** ou
- Améliore l'**efficacité** du programme

Opérateurs

Opérateurs arithmétiques

- +: addition, -: soustraction
- *: multiplication, /: division
- %: modulo

Division entière

- entier / entier → **division entière**
- entier / flottant ou flottant / entier → **division flottante**

Modulos

```
#include <stdio.h>

int main(void) {
    printf("%d %d %d %d\n",
            5 % 3, (-5) % 3, 5 % (-3), (-5) % (-3));
}
```

Résultat:

```
2 -2 2 -2
```

Représentation interne

– Représentation par le **complément à deux** (cours INF2171):

	signe							
127 =	0	1	1	1	1	1	1	1
2 =	0	0	0	0	0	0	1	0
1 =	0	0	0	0	0	0	0	1
0 =	0	0	0	0	0	0	0	0
-1 =	1	1	1	1	1	1	1	1
-2 =	1	1	1	1	1	1	1	0
-127 =	1	0	0	0	0	0	0	1
-128 =	1	0	0	0	0	0	0	0

$\rightarrow (128-1)_2 = (127)_2 = \mathbf{0}1111111$

$\rightarrow (128-2)_2 = (126)_2 = \mathbf{0}1111110$

$\rightarrow (128-127)_2 = (1)_2 = \mathbf{0}0000001$

$\rightarrow (128-128)_2 = (0)_2 = \mathbf{0}0000000$

– S'il y a **débordement** (*overflow*), il n'y a pas d'erreur

```
signed char c = 127, c1 = c + 1;
printf("%d %d\n", c, c1);
// Affiche 127 -128
```

Opérateurs de comparaison et logiques

Opérateurs de comparaison

- `==`: égalité
- `!=`: inégalité
- `>`: stricte supériorité
- `>=`: supériorité
- `<`: stricte infériorité
- `<=`: infériorité

Opérateurs logiques

- `!`: négation
- `&&`: et
- `||`: ou (inclusif)
- L'évaluation est **paresseuse** pour `&&` et `||`

Opérateurs d'affectation et de séquençage

- =, +=, -=, *=, /=, %=

```
int x = 1, y, z, t;
t = y = x;     // Equivaut à t = (y = x)
x *= y + x;    // Equivaut à x = x * (y + x)
```

- Incrémentation et décrémentation: ++ et --

```
int x = 1, y, z;
y = x++;      // y = 1, x = 2
z = ++x;      // z = 3, x = 3
```

- (rarement utilisé) opérateur de séquençage ,: évalue les expressions dans l'ordre et retourne le résultat de la dernière

```
int a = 1, b;
b = (a++, a + 2);
printf("%d\n", b);
// Affiche 4
```

Opérateur ternaire

```
<condition> ? <valeur si vrai> : <valeur si faux>
```

```c
#include <stdio.h>

void print_room(unsigned int n) {
    printf("There %s %d pe%s in this room\n",
           n <= 1 ? "is" : "are",
           n,
           n <= 1 ? "rson" : "ople");
}

int main(void) {
    print_room(0);
    print_room(1);
    print_room(2);
}
```

Résultat:

```
There is 0 person in this room
There is 1 person in this room
There are 2 people in this room
```

Opérations bit à bit

- `&`: et
- `|`: ou inclusif
- `^`: ou exclusif (xor)

Utilité?

- Pour **optimiser** certains calculs (programmation vectorielle)
- Ou pour **combiner** des options (*flags*)
- Par exemple, la fonction `SDL_Init`:

```
[...]
if (SDL_Init(SDL_INIT_VIDEO | SDL_INIT_AUDIO) < 0) {
    fprintf(stderr, "SDL failed to initialize: %s\n",
            SDL_GetError());
    return NULL;
}
[...]
```

L'opérateur `sizeof`

Retourne le **nombre d'octets** (`size_t`) utilisés par

- un **type** de données: `sizeof(int)`
- une valeur **littérale**: `sizeof("bonjour")`
- une **variable**: `sizeof(i)`
- un tableau de taille **fixe**: `sizeof(a)` (on va y revenir)

Remarque

L'expression est évaluée à la **compilation**

Utilité

- Code **plus portable**
- **Allocation dynamique** (on va y revenir)

Taille des types entiers

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

int main(void) {
    printf("sizeof(bool)      = %ld\n", sizeof(bool));
    printf("sizeof(char)      = %ld\n", sizeof(char));
    printf("sizeof(short)     = %ld\n", sizeof(short));
    printf("sizeof(int)       = %ld\n", sizeof(int));
    printf("sizeof(long)      = %ld\n", sizeof(long));
    printf("sizeof(long long) = %ld\n", sizeof(long long));
}
```

Résultat (varie selon l'architecture):

```
sizeof(bool)      = 1
sizeof(char)      = 1
sizeof(short)     = 2
sizeof(int)       = 4
sizeof(long)      = 8
sizeof(long long) = 8
```

Taille des types flottants

```c
#include <stdio.h>

int main(void) {
    printf("sizeof(float)       = %ld\n", sizeof(float));
    printf("sizeof(double)      = %ld\n", sizeof(double));
    printf("sizeof(long double) = %ld",   sizeof(long double));
}
```

Résultat (varie selon l'architecture):

```
sizeof(float)       = 4
sizeof(double)      = 8
sizeof(long double) = 16
```

Taille de variables et de tableaux

```c
#include <stdio.h>

int main(void) {
    int i;
    char c;
    double xs[4];
    printf("sizeof i      = %ld\n", sizeof i);
    printf("sizeof c      = %ld\n", sizeof c);
    printf("sizeof(xs[0]) = %ld\n", sizeof xs[0]);
    printf("sizeof(xs)    = %ld\n", sizeof xs);
}
```

Résultat (varie selon l'architecture):

```
sizeof i      = 4
sizeof c      = 1
sizeof(xs[0]) = 8
sizeof(xs)    = 32
```

Conversions

Conversions des types numériques

- Souvent, on applique un opérateur **binaire**
- Sur deux valeurs de **types différents**
- Il y a alors **promotion**, ou **conversion** (*cast*) automatique
- Entre valeurs **entières**:

$$\texttt{bool} \rightarrow \texttt{char} \rightarrow \texttt{short} \rightarrow \texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{long long}$$

- Entre valeurs **flottantes**:

$$\texttt{float} \rightarrow \texttt{double} \rightarrow \texttt{long double}$$

- Promotion automatique **entier** $\rightarrow$ **flottant**
- Règles plus complexes pour types **signés** et **non signés**
- Éviter de **mélanger** les types dans une même opération
- Sauf cas **idiomatiques**
- Montrer les conversions de façon **explicite**

Conversions implicites

Attention aux conversions implicites entre types signés et non signés:

```c
#include <stdio.h>

int main(void) {
    char x = -1, y = 20, v;
    unsigned char z = 254;
    unsigned short t;
    unsigned short u;

    t = x;
    u = y;
    v = z;
    printf("%d %d %d\n", t, u, v);
    // Affiche 65535 20 -2
}
```

Conversion explicites

```c
#include <stdio.h>

int main(void) {
    unsigned char x = 255;
    printf("%d\n", x);
    // Affiche 255
    printf("%d\n", (signed char)x);
    // Affiche -1
    int y = 3, z = 4;
    printf("%d %f\n", z / y, ((float)z) / y);
    // Affiche 1 1.333333
}
```

Priorité des opérateurs

Arité	Associativité	Par priorité décroissante
2	$\rightarrow$	`( )`, `[ ]`
2	$\rightarrow$	`->`, `.`
1	$\leftarrow$	`!`, `++`, `--`, `+`, `-`, `(int)`, `*`, `&`, `sizeof`
2	$\rightarrow$	`*`, `/`, `%`
2	$\rightarrow$	`+`, `-`
2	$\rightarrow$	`<`, `<=`, `>`, `>=`
2	$\rightarrow$	`==`, `!=`
2	$\rightarrow$	`&&`
2	$\rightarrow$	`||`
3	$\rightarrow$	`?  :`
1	$\leftarrow$	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
2	$\rightarrow$	`,`

Tableaux

Tableaux

- Collection de données **homogènes** (de même type)
- Stockées de façon **contiguë** en mémoire

```
// Déclaration seulement

// Réserve un espace mémoire de taille 8 * sizeof(int)
int t1[8];
// Réserve un espace mémoire de taille n * sizeof(double)
// Seulement avec -std=c99 ou -std=c11
// Allocation sur la pile et non sur le tas (heap)
double t2[n];

// Définition et initialisation

// Réserve un espace mémoire de taille 8 * sizeof(int)
int t3[] = {5,2,0,1,3,4,7,6};
// Réserve un espace mémoire de taille 10 * sizeof(int)
// Deux premières valeurs initialisées à 1 et 1
// Autres valeurs indéterminées si variables automatiques
int t4[6] = {1,1};
```

Représentation schématique de la mémoire

Tas

`t1`

0	1	2	3	4	5	6	7
??	??	??	??	??	??	??	??

$8 \times 4o = 32o$

`t3`

0	1	2	3	4	5	6	7
5	2	0	1	3	4	7	6

$8 \times 4o = 32o$

`t4`

0	1	2	3	4	5
1	1	??	??	??	??

$6 \times 4o = 24o$

Pile

`t2`

0	1	2	3	4	...	n-1
??	??	??	??	??	...	??

$n \times 8o = (8n)o$

Opérateur []

```c
#include <stdio.h>
#define ALPHABET_SIZE 26
#define SENTENCE "the quick brown fox jumps over a lazy dog"

int main(void) {
    // Déclaration
    unsigned int num_occurrences[ALPHABET_SIZE];
    // Initialisation
    for (unsigned int i = 0; i < ALPHABET_SIZE; ++i)
        num_occurrences[i] = 0;
    // Écriture
    char c;
    for (int i = 0; (c = SENTENCE[i]) != '\0'; ++i)
        if (c >= 'a' && c <= 'z')
            ++num_occurrences[c - 'a'];
    // Lecture
    for (int i = 0; i < ALPHABET_SIZE; ++i) {
        printf("%c: %d  ", i + 'a', num_occurrences[i]);
        if (i == 12) printf("\n");
    }
}
```

Résultat:

```
a: 2  b: 1  c: 1  d: 1  e: 2  f: 1  g: 1  h: 1  i: 1  j: 1  k: 1  l: 1  m: 1
n: 1  o: 4  p: 1  q: 1  r: 2  s: 1  t: 1  u: 2  v: 1  w: 1  x: 1  y: 1  z: 1
```

Attention!

- Aucune **vérification** s'il y a dépassement
- Autant pour la **lecture** que pour l'**écriture**
- Source fréquente d'erreur de segmentation (*segfault*)

```c
#include <stdio.h>

int main(void) {
    int a[] = {12, 24, 36, 48};
    int b[] = {60, 72};
    // %-3: alignement à gauche (-) sur 3 caractères (3)
    for (unsigned int i = 0; i <= 4; ++i)
        printf("a[%d] = %-3d   ", i, a[i]);
    printf("\n");
    a[1] = -24; a[4] = -60;
    for (unsigned int i = 0; i <= 4; ++i)
        printf("a[%d] = %-3d   ", i, a[i]);
}
```

Résultat:

```
a[0] = 12    a[1] = 24    a[2] = 36    a[3] = 48    a[4] = -1140900672
a[0] = 12    a[1] = -24   a[2] = 36    a[3] = 48    a[4] = -60
```

Chaînes de caractères

- Cas **particulier** de tableau
- Ses éléments sont de type `char`
- Chaînes **littérales** délimitées par des guillemets " "
- Chaîne **bien formée**: doit terminer par le caractère \0

```c
#include <stdio.h>

int main(void) {
    // En mémoire, ['l','i','n','u','x','\0']
    char s[] = "linux";
    for (unsigned int i = 0; i < 6; ++i) {
        printf("%d %c %d\n", i, s[i], s[i]);
    }
    return 0;
}
```

Résultat:

```
0 l 108
1 i 105
2 n 110
3 u 117
4 x 120
5   0
```

Bibliothèque `string.h`

Suppose que les chaînes sont **bien formées** (terminent par `\0`)

Plusieurs **fonctions** disponibles:

- `strlen`: longueur d'une chaîne
- `strcat`/`strncat`: concaténation de deux chaînes
- `strcmp`/`strncmp`: comparaison de deux chaînes
- `strcpy`/`strncpy`: copie d'une chaîne dans une autre
- `strstr`: première occurrence d'une sous-chaîne dans une chaîne
- `strtok`: segmentation (*tokenization*) d'une chaîne selon délimiteurs
- ...

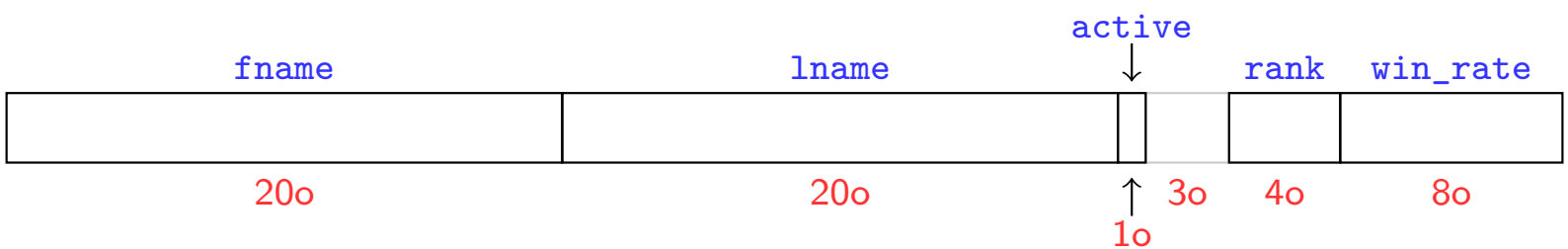
On va y revenir plus tard (pointeurs)

Structures et unions

Structure

- Aussi appelée **enregistrement**
- Regroupe en un même bloc des données **hétérogènes**
- Définit un **nouveau type** de données
- Déclarée à l'aide du mot réservé `struct`
- **Contigu** en mémoire
- **Alignement**: compilateur décale selon l'architecture
- Rend l'adressage plus **efficace**

```
struct Player {
    char fname[20];
    char lname[20];
    bool active;
    unsigned int rank;
    double win_rate;
};
```



Déclaration, initialisation, lecture, écriture

```c
#include <stdio.h>

// Déclaration d'un nouveau type
struct point2d {
    double x;
    double y;
};

int main(void) {
    // Déclaration d'une variable non initialisée
    struct point2d p1;
    // Déclaration et initialisation
    struct point2d p2 = {2.0, -1.2};
    // Écriture dans champs avec opérateur .
    p1.x = 3.6;
    p1.y = -4.9;
    // Lecture des champs avec opérateur .
    // %f: nombre flottant (float ou double)
    printf("p1 = (%f,%f)\n", p1.x, p1.y);
    printf("p2 = (%f,%f)\n", p2.x, p2.y);
}
```

Résultat:

```
p1 = (3.600000,-4.900000)
p2 = (2.000000,-1.200000)
```

Affectation (*compound literal*)

- Affectation en bloc possible
- Depuis standard **C99**

```c
#include <stdio.h>

struct Rectangle {
    float x;
    float y;
    float width;
    float height;
};

int main(void) {
    struct Rectangle r = {1.0, 2.0, 5.0, 6.0};
    // Affectation en bloc (conversion obligatoire)
    r = (struct Rectangle){3.0, 8.0, 9.0, 7.0};
    float a = 0.0, b = 0.0, c = 1.0, d = 2.0;
    // Affectation en bloc avec champs nommés
    r = (struct Rectangle){.x      = a,
                           .y      = d,
                           .width  = b,
                           .height = c};
    return 0;
}
```

Copie de structures

- Opérateur = sur les structures: copie les **champs** un par un
- Lors d'appel de fonctions, la structure est **copiée**

```c
#include <stdio.h>

struct point2d { double x; double y; };

void print_point(struct point2d p) {
    printf("point(%f,%f)", p.x, p.y);
}

int main(void) {
    struct point2d p1 = {3.2, -1.4};
    struct point2d p2 = p1;
    p2.y *= -1;
    print_point(p1); printf("\n"); print_point(p2);
    return 0;
}
```

Résultat:

```
point(3.200000,-1.400000)
point(3.200000,1.400000)
```

Structures imbriquées

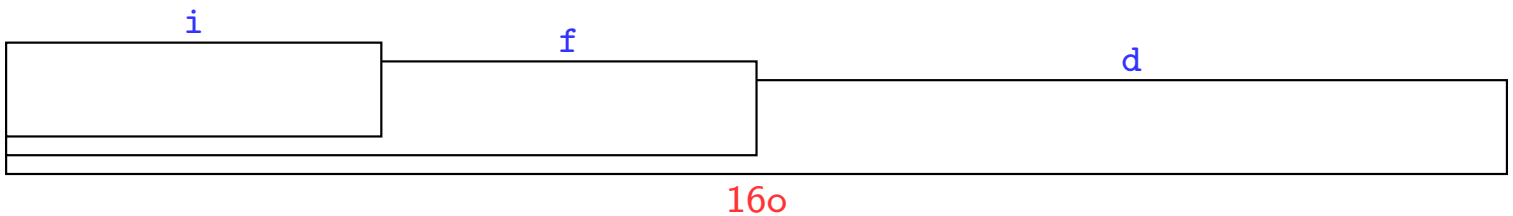
- Les structures peuvent être **imbriquées**
- Elles peuvent aussi être composées avec des **tableaux**
- Et avec des **unions** et des **pointeurs** (plus tard)

```
struct point2d {
    double x;
    double y;
};

struct segment {
    struct point2d p;
    struct point2d q;
};

struct triangle {
    struct point2d points[3];
};

struct carre {
    struct point2d points[4];
};
```

Unions

- Contient des données de **types différents**
- Une seule donnée peut exister à un **instant donné**
- Même **syntaxe** que pour les structures
- Mémoire réservée: $\geq$ `sizeof` du type le **plus volumineux**
- **Utilité**: factorisation de code, économie d'espace
- **Difficulté**: il faut savoir quel type est le bon

```
// Un nombre entier ou flottant
union nombre {
    int    i;
    float  f;
    double d;
};
```



Exemple

```c
#include <stdio.h>

int main(void) {
    union nombre {
        int    i;
        float  f;
        double d;
    };
    union nombre n;
    n.i = 152;
    printf("%d %f %f\n", n.i, n.f, n.d);
    n.f = 87.31;
    printf("%d %f %f\n", n.i, n.f, n.d);
    n.d = -999.999;
    printf("%d %f %f\n", n.i, n.f, n.d);
}
```

Résultat: (peut varier)

```
152 0.000000 0.000000
1118740152 87.309998 0.000000
-206158430 -288821517785131196433866225090 56.000000 -999.999000
```

Structures, unions et enum anonymes

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

struct choice {
    bool is_number;                     // marqueur de type de donnée
    union {                             // union anonyme
        float number;
        enum {YES, NO, MAYBE} answer; // enum anonyme
    };
};

void print_choice(struct choice c) {
    if (c.is_number) {
        printf("%lf\n", c.number);
    } else {
        switch (c.answer) {
            case YES:   printf("yes");   break;
            case NO:    printf("no");    break;
            case MAYBE: printf("maybe");
        }
        printf("\n");
    }
};

int main() {
    struct choice c = {false, .answer = YES}; print_choice(c);
    c = (struct choice){true, 3.14};          print_choice(c);
}
```

Résultat:

```
yes
3.140000
```

Utilisation de `typedef`

- On peut éviter de réécrire `struct`, `union` et `enum`
- En utilisant l'instruction `typedef`
- **Avantage**: syntaxe plus proche de Java et C++

```c
#include <stdio.h>

typedef struct {
    double x;
    double y;
} point2d;

int main(void) {
    point2d p = {2.0, -1.5};
    printf("point(%f,%f)\n", p.x, p.y);
    return 0;
}
```

- Mais considéré **abusif** (voir discussion, en particulier cette réponse)
- Donc à **éviter** dans le cours

Fonctions

Utilité des fonctions

- **Unité de base** d'un programme (avec les types)
- Effectue une tâche **précise**
- Doit préférablement être **courte**
- Permet de **diviser** un problème complexe
- À la base de la **réutilisation**
- Et de la **factorisation** de code
- Fondamentales pour la **maintenance**
- Doivent être **bien nommées**
- Avec une syntaxe et une logique **uniforme**
- Devraient être **documentées**

Fonction `main`

- Fonction spéciale, point d'entrée du programme
- Communique à l'aide des **paramètres** `argc` et `argv`
- Et d'une **valeur de retour** de type `int`
- On va y revenir

Fonctions pures et effets de bord

Fonction pure

- Le résultat ne **dépend** que des arguments
- Retourne le **même résultat** si appelée avec les **mêmes arguments**
- Pas d'**effet** de bord
- Par exemple, les fonctions **mathématiques**
- Ou les fonctions de **lecture seule**

Fonction non pure ou à effets de bord

- Le résultat dépend de l'**environnement** ou le modifie
- Peut retourner un **résultat différent** même lorsque appelée avec les **mêmes arguments**
- **Exemples**: écriture sur `stdout`, lecture sur `stdin`, lecture/écriture fichier, allocation dynamique, chargement de données, ...

Arguments et paramètres

- **Paramètre**: nom de la variable dans la fonction
- **Argument**: valeur passée lors de l'appel
- Valeurs passées par **copie**
- Aussi possible par **adresse** (on va y revenir)

La valeur `void`

- `void` si pas de valeur retournée
- `f(void)`: fonction sans paramètre
- `f()`: fonction avec nombre inconnu de paramètres

Nombre variable de paramètres

- `f(int, ...)`: paramètre entier, suivi de paramètres optionnels
- **Exemple**: `printf` et `scanf`

Exemples

```c
#include <stdio.h>

// Aucun paramètre, aucune valeur retournée
void f1(void) {
    printf("f1()");
};

// Un paramètre, aucune valeur retournée
void f2(int a) {
    printf("f2(%d)", a);
}

// Aucun paramètre, une valeur retournée
int f3() {
    return 42;
}

// Un paramètre, une valeur retournée
int f4(int a) {
    return -a;
}

int main(void) {
    f1();  // Pas d'argument
    f2(4); // Argument obligatoire
    f3();  // Pas obligatoire de récupérer la valeur
    printf("%d\n", f4(4));
    return 0;
}
```

Résultat:

```
f1()f2(4)-4
```

Déclaration et définition (1/2)

- Attention à l'ordre de **déclaration** des fonctions
- Car la compilation se fait en **une** passe

```c
#include <stdio.h>

int a(int x) {
    return 42;
}

int b(int x) {
    return c(x - 1) + 1;
}

int c(int x) {
    return a(x + 1);
}
```

Avertissement à la compilation:

```
fonctiondd.c: In function 'b':
fonctiondd.c:8:12: warning: implicit declaration of function 'c'
    [-Wimplicit-function-declaration]
    return c(x - 1) + 1;
```

Déclaration et définition (2/2)

Solution: séparer les déclarations et les définitions

```c
#include <stdio.h>

// Déclarations
int a(int x);
int b(int x);
int c(int x);

// Définitions
int a(int x) {
    return 42;
}

int b(int x) {
    return c(x - 1) + 1;
}

int c(int x) {
    return a(x + 1);
}

int main(void) {
    return 0;
}
```

Fonction récursive

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

void print_reverse(char s[], int n) {
    // Cas de base: si n < 0, on ne fait rien
    if (n >= 0) {
        // Cas général: 1) on affiche la dernière lettre
        printf("%c", s[n]);
        // 2) puis on diminue la longueur de la chaîne
        print_reverse(s, n - 1);
    }
}

int main(void) {
    char s[] = "linux";
    char t[] = "esoperesteicietserepose";
    print_reverse(s, strlen(s) - 1); printf("\n");
    print_reverse(t, strlen(t) - 1); printf("\n");
    return 0;
}
```

Résultat:

```
xunil
esoperesteicietserepose
```

La fonction `printf` (1/3)

- Affichage (`print`) **formaté** (`f`)
- Disponible dans la **bibliothèque** `stdio.h`
- Affichage des **types** de base:

Code	Description
`%c`	caractère
`%d`	entier sous forme décimale
`%hd`	entier court sous forme décimale
`%ld`	entier long sous forme décimale
`%u`	entier non signé
`%o`	entier sous forme octale
`%x`	entier sous forme hexadécimale
`%e`	flottant en notation scientifique
`%f`	flottant en notation décimale
`%g`	flottant de façon compacte
`%lf`	double en notation décimale
`%L`	long double en notation décimale
`%s`	chaîne de caractères
`%p`	pointeur

La fonction `printf` (2/3)

- **Options** d'affichage:

Code	Description
`%-`	Alignement à gauche (par défaut à droite)
`%+`	Ajoute le symbole + aux nombres positifs
`%`	Ajoute un espace aux nombres positifs
`%#`	Ajoute un préfixe 0 ou 0X si octal ou hexadécimal
`%<n>`	Largeur d'affichage d'au moins `<n>`
`%*`	Largeur d'affichage variable
`%.<n>f`	Précision numérique ou remplissage de `<n>`

- Très pratique pour afficher des **tableaux** et des **grilles**
- Autres **options** disponibles (voir la documentation)

La fonction printf (3/3)

```c
#include <stdio.h>

int main(void) {
    // Fixe
    printf("|%11d|%11f|\n",      32, -1.4);
    printf("|%-11d|%-11f|\n",    32, -1.4);
    printf("|%.5d|%.5f|\n",      32, -1.4);
    printf("|%11.5d|%11.5f|\n", 32, -1.4);
    // Variable
    printf("|%*d|%*f|\n",      11,    32, 11,    -1.4);
    printf("|%-*d|%-*f|\n",    11,    32, 11,    -1.4);
    printf("|%.*d|%.*f|\n",    5,     32, 5,     -1.4);
    printf("|%*.*d|%*.*f|\n", 11, 5, 32, 11, 5, -1.4);
}
```

Résultat:

```
|          32|  -1.400000|
|32          |-1.400000  |
|00032|-1.40000|
|      00032|   -1.40000|
|          32|  -1.400000|
|32          |-1.400000  |
|00032|-1.40000|
|      00032|   -1.40000|
```

La fonction `main`

- Point d'**entrée** d'un programme C
- Trois **signatures** possibles

```c
// Aucun argument permis
int main(void);

// Avec arguments
// argc: nombre d'arguments, incluant le nom du programme
// argv: tableaux d'arguments
//     argv[0]: nom du programme
//     argv[1]: premier argument
//     argv[2]: deuxième argument
//     ...
int main(int argc, char* argv[]);

// Nombre d'arguments indéfinis (à éviter)
int main();
```

- `char* argv[]`: tableau de pointeurs vers `char` (on va y revenir)

Exemple avec arguments

- `argv[0]`: nom du programme **tel qu'invoqué**
- **Protection** avec " ou '
- Variables d'**environnement** disponibles

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    printf("argc = %d\n", argc);
    for (unsigned int i = 0; i < argc; ++i)
        printf("argv[%d] = %s\n", i, argv[i]);
    return 0;
}
```

Résultat:

```
$ ./prog alpha 1 "un deux" 'trois quatre' "$LANG"
argc = 6
argv[0] = ./prog
argv[1] = alpha
argv[2] = 1
argv[3] = un deux
argv[4] = trois quatre
argv[5] = en_CA.UTF-8
```

Fonction et variables statiques

Variables statiques

- On peut attacher des variables **statiques** à une fonction
- À l'aide du mot réservé `static`
- Mémoire réservée au **début** du programme
- Et **libéré** à la fin du programme
- **Initialisée** à 0 par défaut

Mémoïsation

- On considère n'importe quelle **fonction pure**
- On **« mémorise »** le résultat de la fonction
- Pour certains **arguments** donnés
- Lors d'un appel de fonction avec les **mêmes** arguments
- On n'a pas à **refaire** le calcul

Exemple: `fibo.c`

```c
#include <stdio.h>

unsigned long long fibonacci(unsigned int n) {
    if (n == 0 || n == 1)
        return 1;
    else
        return fibonacci(n - 1) + fibonacci(n - 2);
}

int main(void) {
    // %lld: long long decimal
    for (unsigned int n = 0; n < 40; ++n) {
        printf("%lld ", fibonacci(n));
    }
    printf("\n");
    return 0;
}
```

Problème?

On recalcule plusieurs fois les **mêmes** valeurs

Chronomètre

```
$ gcc fibo.c -o fibo
$ time ./fibo
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m1.452s
user    0m1.451s
sys 0m0.000s
```

- **Solution 1**: « dérécursifier » le programme
- Mais pas **facile** en général
- **Solution 2**: « mémoriser » les valeurs déjà calculées
- À l'aide d'une variable **statique**

Exemple: fibofast.c

```c
#include <stdio.h>

#define MAX_N 1000

unsigned long long fibonacci(unsigned int n) {
    // Le tableau f contient les MAX_N premières valeurs
    // On l'initialise avec {1,1,0,0,0,0,...}
    static unsigned long long f[MAX_N] = {1, 1};
    if (n >= MAX_N)
        // Tableau pas assez grand, on y va naïvement
        return fibonacci(n - 1) + fibonacci(n - 2);
    if (f[n] == 0) {
        // Tableau assez grand, on mémoïse
        f[n] = fibonacci(n - 1) + fibonacci(n - 2);
    }
    return f[n];
}

int main(void) {
    for (unsigned int n = 0; n < 40; ++n) {
        printf("%lld ", fibonacci(n));
    }
    printf("\n");
    return 0;
}
```

Chronomètre

```
$ gcc fibofast.c -o fibofast
$ time ./fibofast
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
2584 4181 6765 10946 17711 28657 46368 75025 121393
196418 317811 514229 832040 1346269 2178309 3524578
5702887 9227465 14930352 24157817 39088169 63245986
102334155
real    0m0.002s
user    0m0.002s
sys 0m0.000s
```

Compilation et Makefiles

Compilateur

- GCC = *GNU Compiler Collection*
- Cygwin GCC: chaîne d'outils Linux pour Windows
- Mingw GCC = *Minimalist GNU for Windows*
- Clang LLVM = *Low Level Virtual Machine*
- Plusieurs autres...

Remarque

Sur **MacOS**, gcc est un alias pour clang:

```
$ gcc --version
[...]
Apple LLVM version 10.0.0 (clang-1000.10.44.4)
Target: x86_64-apple-darwin17.7.0
Thread model: posix
[...]
```

GCC

- **Ensemble** de compilateurs
- Sous une **interface** commune
- Développée par GNU
- Plusieurs **langages** supportés:

C, C++, Objective-C, Fortran, Ada, Go, D

Plusieurs options

- `-c`: compilation seulement
- `-o FICHIER`: spécifier le nom du fichier en sortie
- `-Wall`: afficher tous les avertissements
- `-Wextra`: afficher encore plus d'avertissements
- `-std=STD`: spécifier le standard (`c99`, `c11`, etc.)
- Commande `man gcc` pour toutes les options

Cycle de compilation

Édition du programme source (`.c`)

À l'aide d'un éditeur de texte ou d'un EDD

Compilation (`.c` → `.o`)

- Vérification **syntaxique**
- Produit des fichiers (binaires) **objets**

Édition de liens (`.o` → `.out`) (*linking*)

- Fichiers `.o` assemblés pour former un binaire **exécutable**
- Extension `.out` par **défaut**

Simplifier la compilation

On a vu un peu plus tôt la compilation en deux étapes pour créer un exécutable

- On compile le fichier `.c` en un fichier `.o`

```
$ gcc -c maj.c
```

- On lie les fichiers `.o` en un seul fichier exécutable

```
$ gcc -o maj maj.o
```

- **Problème**: pénible de saisir la commande de compilation chaque fois qu'on apporte une modification au fichier source
- Encore plus avec les **options**:

```
$ gcc -c -Wall -Wextra maj.c
$ gcc maj maj.o
```

- **Solution**: utiliser un `Makefile`

Makefiles

- Existent depuis la fin des années '70
- Fichier **texte**
- Décrit les **dépendances** entre composantes d'un programme
- Automatisent la **compilation** en minimisant le nombre d'étapes
- Malgré qu'ils soient archaïques, ils sont encore **très utilisés**
- Certaines **limitations** corrigées par des outils comme Autoconf et CMake (on va y revenir)
- Préférer le nom `Makefile` (avec une majuscule) à `makefile`

Contenu d'un Makefile

- Des règles **explicites**
- Des règles **implicites** (on va y revenir)
- Des définitions de **variables**
- Des **directives** spécifiques à `make` (on va y revenir)
- Des commentaires, préfixés par #

Syntaxe d'une règle explicite

```
<cible>: <prérequis>
<tab><recette>
```

- `<cible>`: nom de **fichier** ou nom **personnalisé** ou nom **spécial**
- `<prérequis>`: noms de **fichier** ou autres **cibles**
- séparés par des **espaces**
- `<tab>`: caractère de **tabulation**, pas d'espaces
- `<recette>`: suite de **commandes** permettant de générer `<cible>`

Exemple

```
maj: maj.o
    gcc -o maj maj.o

maj.o: maj.c
    gcc -c -Wall -Wextra maj.c
```

Invocation d'un Makefile

- **Invocation** avec la commande `make`:

```
$ make
gcc -c -Wall -Wextra maj.c
gcc -o maj maj.o
```

- La commande `make` ne regarde que la **première** règle
- Et ses **dépendances**, si elles doivent être mises à jour
- Par défaut, les commandes sont **affichées**
- Possible de les faire taire avec `@` ou `-s|--silent`

Astuce Vim

- Associer les touches `,m` à l'invocation `:!make`
- Voir fichier `vimrc` en labo

Variables

Syntaxe

- **Déclaration**: `<nom variable> = <valeur>`
- **Utilisation**: `$(<nom variable>)`
- Variables **textuelles**

Exemple

```
exec = maj
CFLAGS = -Wall -Wextra

$(exec): $(exec).o
    gcc -o $(exec) $(exec).o

$(exec).o: $(exec).c
    gcc -c $(CFLAGS) $(exec).c
```

Cibles spéciales

- **Interprétées** de façon particulière par `make`
- Voir documentation
- En **majuscules**
- **Préfixées** par un point `.`

Exemples

`.PHONY`, `.SUFFIXES`, `.DEFAULT`, `.PRECIOUS`, `.INTERMEDIATE`, `.SECONDARY`, `.SECONDEXPANSION`, `.DELETE_ON_ERROR`, `.IGNORE`, `.SILENT`, `.POSIX`, …

`.PHONY`

- Permet de déclarer des cibles **personnalisées**
- Indique que ce ne sont pas des **noms** de fichier

Exemple avec .PHONY

```
exec = maj
readme = README
CFLAGS = -Wall -Wextra -std=c11

$(exec): $(exec).o
    gcc -o $(exec) $(exec).o

$(exec).o: $(exec).c
    gcc -c $(exec).c

.PHONY: clean html

clean:
    rm -f *.o
    rm -f $(exec)

html:
    pandoc -o $(readme).html -sc pandoc.css $(readme).md
```

Invocation d'une cible personnalisée

Syntaxe

```
$ make <cible>
```

Exemple

```
$ make
gcc -c -Wall -Wextra -std=c11 maj.c
gcc -o maj maj.o
$ make clean
rm -f *.o
rm -f maj
$ make html
pandoc -o README.html -sc pandoc.css README.md
```

Préprocesseur

Directives au préprocesseur

Précompilation

- **Interprétées** par le préprocesseur **avant** la compilation
- Symbole # au **début** de la ligne
- On peut insérer des **espaces** entre # et la directive

Directives permises

- `#include`: **inclusion** d'un fichier externe
- `#define`: définition d'un **symbole** ou d'une **macro**
- `#undef`: **annulation** d'un symbole ou d'une macro
- `#ifdef`/`#ifndef`: **vérifie** si une macro est définie ou non définie
- `#if`/`#else`/`#elif`/`#endif`: structure **conditionnelle**
- `#error`: indique une **erreur fatale**
- `#pragma`: pour des traitements plus **spécifiques**

La directive `#include`

- Permet d'**inclure** un fichier source externe
- Généralement, on inclut le fichier d'**en-tête** (extension `.h`)
- Inclusion d'un fichier `.c` possible, mais à **éviter**

Exemples

```c
// Bibliothèques standards
#include <stdio.h>
#include <stdbool.h>
#include <math.h>

// Modules locaux
#include "utils.h"
#include "constants.h"

// Bibliothèques tierces
#include <jansson.h>
#include <cairo.h>
#include <sdl2.h>
```

La directive #define

- **Syntaxe**: `#define <symbole> <valeur>`
- **Remplace** toutes les occurrences de `<symbole>` par `<valeur>`
- La valeur est donnée par **le reste de la ligne**
- Pour une valeur **multiligne**, utiliser le caractère \

Exemples

```
#define BUFFER_SIZE 100
#define ERROR_MSG "Error: wrong number of arguments"
#define USAGE "\
Usage: %s [-h|--help] [-n|--num-iterations VALUE]\n\
\n\
Simulates the propagation of a rumour.\n\
\n\
Optional arguments:\n\
  -h, --help                  Shows this help message and exit.\n\
  -n, --num-iterations VALUE  The number of iterations.\n\
                              The default value is 100.\n\
"
```

Autres directives

- `#undef`: **annulation** d'un symbole ou d'une macro
- `#ifdef`/`#ifndef`: **vérifie** si une macro est définie ou non définie
- `#if`/`#else`/`#elif`/`#endif`: structure **conditionnelle**

Utilité

- Empêcher les **inclusions** multiples de modules
- **Portabilité** du code

Plus tard

- Quand on va parler de **modules**
- Puis de **bibliothèques** (*libraries*)

INF3135
Construction et maintenance de logiciels

Chapitre 2: Outils de développement logiciel

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Style de programmation

Définition

Extrait de Wikipedia:

> *« Le style de programmation est un ensemble de règles ou de lignes directrices utilisées lors de l'écriture du code source d'un programme informatique. Il est souvent affirmé que suivre un style de programmation particulier aidera les programmeurs à lire et à comprendre le code source conforme au style, et aidera à éviter les erreurs. »*

- **Conventions**, ensemble de **règles**
- Pour l'écriture de **code source**
- Améliore la **lisibilité**
- Permet de réduire les **erreurs**

Style de programmation en C

- C a été **standardisé** dans les années 80 (ANSI C89/C90)
- Mais aucun standard de **programmation** proposé

Quelques exemples

- Indian Hill
- NASA
- Noyau Linux (Linus Torvalds)
- GNU
- GNOME

« The Single Most Important Rule »

« Check the surrounding code and try to imitate it. »

— Extrait du site de GNOME

Contenu d'un fichier

- Un **module** C devrait contenir les éléments suivants, dans l'ordre:

```c
/**
 * Documentation d'en-tête du fichier
 */

#include // inclusion des bibliothèques

#define  // et autres constantes

// Déclaration des types (struct, union, enum, typedef)

// Déclaration des fonctions (avec leur docstring)
// Regrouper les fonctions par thématique

// Implémentation des fonctions (sans les documenter)

// Fonction main
```

- **Bonne pratique**: utiliser des commentaires pour mettre en évidence la **structure** du fichier

Espacement

- Au plus **80 caractères** par ligne
- **Indentation**: 2, 4 ou 8
- **Tabulations** ou **espaces**: ne pas mélanger!
- **Aérer** autour des opérateurs et des délimiteurs:

```
for (int j = 3; j < 10; ++j) // Bien
for(int j=3;j<10;++j)        // À éviter
```

- Éviter les **suites** de lignes vides
- Aligner les **paramètres** lors d'un long appel de fonction

```
printf("L'équation\n  %.2lfx^2 %c %.2lfx %c %.2lf == 0\n",
       sol.equation.a,
       sol.equation.b >= 0 ? '+' : '-', fabs(sol.equation.b),
       sol.equation.c >= 0 ? '+' : '-', fabs(sol.equation.c));
```

Deux styles fréquents

Aéré:

```
if (valid)
{
    printf("Everything is fine\n.");
}
else
{
    printf("Something went wrong\n.");
}
```

Compact:

```
if (valid) {
    printf("Everything is fine\n.");
} else {
    printf("Something went wrong\n.");
}
```

Nomenclature (1/2)

Variables

- **Syntaxe** `camelCase` ou `snake_case`
- Plus la **portée** est importante, plus le nom devrait être **long**
- Préférer variables **courtes** pour indices d'un tableau: `i`, `j`, `k`

Types `struct`, `union` et `enum`

- **Syntaxe**: `PascalCase` ou `snake_case`
- Éviter `typedef` le plus possible

Orthographe

- Attention aux **fautes**
- Ne pas mélanger les **langues**

Nomenclature (2/2)

Fonctions

- **Syntaxe**: `camelCase` ou `snake_case`
- Si retourne `void`, utiliser verbe à l'**infinitif**: `parse_values`, `initialize_canvas`, `multiply_arrays`
- Si retourne nombre, utiliser **nom** correspondant: `num_nodes`, `size`, `win_ratio`, `average_income`
- Si retourne `bool`, utiliser verbe à l'**indicatif**: `is_valid`, `has_attribute`, `contains_point`
- Ne pas mettre **systématiquement** `get` et `set`

Fichiers

- **Syntaxe**: `snake_case.c`, `snake_case.h`
- Nom le plus **court** possible

Commentaires

- **Syntaxe**:

```
// Commentaire sur une ligne

/* Commentaire
   multiligne */

/**
 * Docstring
 */
```

- *Docstrings* suffisent la plupart du temps
- **Commenter** le code traduit souvent un mauvais **découpage**
- Ou une mauvaise **nomenclature**
- **Éviter** de paraphraser le code
- **Supprimer** le code en commentaire à la livraison

Valeurs magiques

- Valeur magique = valeur constante
- **Nombres**, mais aussi **chaînes** de caractères
- À **éviter** le plus possible
- **Critère**: dès qu'elles sont utilisées plus d'une fois
- **Exemples**: dimensions d'un tableau, bornes de valeurs permises, messages d'erreur

Valeurs littérales acceptables

- 0 ou 1, très souvent
- 2 dans une formule mathématique ou pour vérifier la parité
- Le caractère `'\0'`
- Une chaîne fréquent (`"yes"`, `"no"`)
- Une option (`-o|--output`, `-c|--count`)

Factorisation

- Éviter au maximum la **duplication** de code
- Selon les **possibilités** du langage

Quand factoriser?

- Au fur et à mesure
- Ne pas attendre à la fin

Mécanismes

- À l'aide de **fonctions**
- **Généraliser** fonction en ajoutant paramètre
- **Réduire** nombre de paramètres en déclarant **types**
- À l'aide de l'opérateur **ternaire**
- À l'aide des **pointeurs** (on va y revenir)
- L'affichage et la lecture **formaté** (`printf`, `scanf`, `sscanf`)

Documentation

Plusieurs types de documentation

Code source (*docstrings*)

- **Modules**: description, auteurs, license, version, etc.
- **Fonctions**: description, paramètres, valeur de retour, etc.

Utilisateur

- Guide de l'utilisateur
- Souvent dans un fichier `README`
- Tutoriels pour l'utilisateur

Développeur

- Documentation des **modifications** apportées
- Guide du développeur
- Tutoriels pour le développeur

Langage de balisage léger

Définition (extraite de Wikipedia):

> *« Un langage de balisage léger est un type de langage de balisage utilisant une syntaxe simple, conçu pour être aisé à saisir avec un éditeur de texte simple, et facile à lire dans sa forme non formatée. »*

Exemples:

- Markdown,
- ReStructuredText,
- Asciidoc, etc.

Contre-exemples:

- HTML, XML: pas légers!
- YAML, JSON, légers, mais plutôt pour structurer des données

Markdown

- **2004**: créé par John Gruber avec Aaron Swartz
- Peu modifié ensuite par les auteurs originaux
- Extension de fichier: `.md` ou `.markdown`
- Peut facilement être transformé en HTML ou en PDF
- grâce notamment au programme Pandoc
- Supporté sur plusieurs **plateformes** ou **forums**
- Pas de **standardisation** formelle
- Possibilité d'**enchasser** du HTML
- **Attention!** éviter s'il existe un équivalent Markdown!

Plusieurs variantes (*flavors*)

- Multimarkdown, qui est une extension
- GitHub Flavored Markdown, d'abord développé pour GitHub
- GitLab Flavored Markdown, développé pour GitLab

Formatage

- **Emphase** (balise en HTML): étoiles simples *mot* ou soulignés simples _mot_
- **Emphase forte** (balise): étoiles doubles **mot**
- **Souligner**: soulignés doubles __mot__
- **Bout de code** (balise <code>): apostrophes inversées `mot`
- **Citation** (balise <blockquote>): commencer par >

```
> Extrait d'une conversation qu'on souhaite commenter
> Peut être sur plusieurs lignes
```

- **Paragraphes** (balise <p>): il suffit de laisser une ligne vide

```
Premier paragraphe

Deuxième paragraphe
```

Bloc de code

– Trois **apostrophes inversées** (*backticks*), suivi du **langage**

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

int main(int argc, char *argv[]) {
    printf("Hello, world!\n");
    return 0;
}
```


```sh
$ sudo apt install pandoc
$ gcc -o maj maj.c
```
````

– **Sans langage** associé: indenter de 4 espaces

```
    Bout de texte qui apparaîtra comme du code
    Pour des extraits de fichiers texte, par exemple
```

Listes

- Liste **non ordonnée** (balise <ul>): étoiles ou tirets

```
* Premier élément
* Deuxième
    * Élément imbriqué (au moins 4 espaces)
* Troisième
```

- Liste **ordonnée** (balise <ol>): chiffre suivi d'un point

```
1. Premier élément
2. Deuxième élément
3. Troisième élément
```

- Liste **à cocher**: crochets avec x optionnel

```
- [ ] Finir TP1
- [ ] Relire notes de cours
- [x] Se reposer
```

Titres

- Pour **structurer** un document Markdown (balises `<h1>` à `<h6>`)

```
# Travail pratique 1

## Description

## Auteurs

## Exemples

### Exemple 1

### Exemple 2
```

- Attention de ne pas mettre **trop** de titres
- Ajuster la **profondeur** selon la taille du document

Liens

- Pour insérer un **hyperlien**:

```
[texte](lien relatif ou absolu)
```

- Pour insérer une **image**:

```
![texte](lien relatif ou absolu vers l'image)
```

Documentation

- Vers **sites officiels**
- Pour les **références** (Wikipedia, code, article, livre, etc.)

Mathématiques

- Supporté dans certaines **variantes**
- **Exemples**: Mattermost
- Ou encore GitLab Flavored Markdown
- **Dans le texte**: un dollar suivi d'une apostrophe inversée
- **Bloc mathématique**: comme pour le code, avec le mot `math`

````
Dans le texte, c'est comme ça: $`(x + 1, y - 2)`$.

Pour un bloc mathématique

```math
f(x,y) = x^2 + y^2 - 1
```
````

Documentation du code source

- Aucun standard de **documentation** officiel
- Selon le projet, le standard varie
- Dans le cours, nous allons utiliser le standard **Javadoc**

Étiquette	Description
`@author`	Auteur du module ou de la fonction
`@deprecated`	Indique que la fonction ou le module ne devrait plus être utilisé
`@exception`	Décrit le type d'exception qui peut être soulevée
`{@link}`	Insère un lien vers un autre module, fonction, etc.
`@param`	Une brève description d'un paramètre de fonction
`@return`	Une brève description de la valeur de retour d'une fonction
`@see`	Indique une fonction ou un module relié
`@version`	Indique le numéro de version de la fonction ou du module
etc.	

Documentation d'en-tête: *docstrings*

Fichier

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- **Auteur** (obligatoire)

Fonctions

- Description **générale** en une phrase (obligatoire)
- Description **détaillée** (optionnelle)
- **Exemples** d'utilisation (optionnels)
- Description de **chaque paramètre** (obligatoire)
- Description de la **valeur de retour**, s'il y en a une (obligatoire)

Exemples

```c
/**
 * Loads the company data from a JSON file
 *
 * @param company   The resulting company object
 * @param filename  The JSON filename path
 * @return          Error code indicating success or error
 */
enum error load_company_data(struct company *company,
                             const char *filename);

/**
 * Indicates if a triangle contains a given point
 *
 * @param t  The triangle
 * @param p  The point
 * @return   True if and only if the triangle contains the point
 */
bool triangle_contains_point(const struct triangle *t,
                             const struct point *p);
```

Documentation des modules

- Toujours documenter l'**en-tête des fichiers**:
- Utiliser le format **Markdown** (souvent reconnu)

```
/**
 * geometry.c
 *
 * Provides different data structures and functions for handling
 * 2-dimensional geometry.
 *
 * The basic type is `struct point`:
 *
 * ```c
 * struct point p = {1.5, -2.3};
 * print_point(&p);
 * ```
 *
 * [...]
 *
 * @author Alexandre Blondin Masse
 */
```

Bats

Bats

- *Bats = Bash Automated Testing System*
- **2011-2016**: créé et maintenu par Sam Stephenson
- **Ancien lien**: https://github.com/sstephenson/bats
- Depuis **2017**: maintenu par la communauté bats-core
- **Divergence** (*fork*) du projet original
- **Lien actuel**: https://github.com/bats-core/bats-core
- **Paquets**: Homebrew et NPM
- **Image**: DockerHub

Installation

```
# Avec apt - version pas à jour
$ sudo apt install bats

# À partir des fichiers sources - version récente
$ git clone https://github.com/bats-core/bats-core.git
$ cd bats-core
$ sudo ./install.sh /usr/local
```

Tests unitaires

Syntaxe

```
@test "Nom du test" {
    # Suite de commandes shell
    # Suite de tests (entre crochets [])
}
```

Commandes et variables spéciales

- `run`: exécute et remplit les variables `status`, `output` et `lines`
- `$status`: **code de retour** de la commande appelée
- `$output`: **sortie** résultante (`stdout` et `stderr` combinés)
- `${lines[i]}`: `i`-ème **ligne** de `$output`
- `skip`: permet de **sauter** un test

Exemples

```
@test "Addition" {
    resultat="$(echo $((1 + 2)))"    # Pas obligé d'utiliser run
    [ "$resultat" -eq 3 ]            # Teste si sortie de echo est 3
}

@test "Avec run" {
    run echo $((1 + 2))              # Avec run
    [ "$status" -eq 0 ]              # Teste code de retour de echo
    [ "$output" == "3" ]             # Teste si sortie de echo est 3
}

@test "Plusieurs lignes" {
    run echo -e "ligne 1\nligne 2"   # Avec run
    [ "${lines[0]}" == "ligne 1" ]   # Teste contenu de la 1re ligne
    [ "${lines[1]}" == "ligne2" ]    # Teste contenu de la 2e ligne
}

@test "Un test désactivé" {
    skip                             # On désactive le test
    run echo "un autre test"
    [ "$status" -eq 1 ]              # Teste si echo échoue
}
```

Invocation

Il suffit d'entrer la commande `bats <fichier>`:

```
$ bats tests.bats
 ✓ Addition
 ✓ Avec run
 ✗ Plusieurs lignes
   (in test file tests.bats, line 15)
     `[ "${lines[1]}" == "ligne2" ]   # Teste contenu de la 2e
         ligne' failed
 - Un test désactivé (skipped)

4 tests, 1 failure, 1 skipped
```

Plusieurs options (`bats --help`):

- `-c|--count`: compter le nombre de tests
- `-f|--filter`: lancer tests qui vérifient une ER
- `-F|--formatter`: préciser le format d'affichage
- `-t|--tap`: afficher selon le protocole TAP
- `-j|--jobs`: tester en parallèle

Protocole TAP

- **TAP** = *Tests Anything Protocol*
- Format **texte** simple
- **Site officiel**: `http://testanything.org/`
- Spécification
- Format utilisé par **GitLab-CI** (pas de caractères spéciaux):

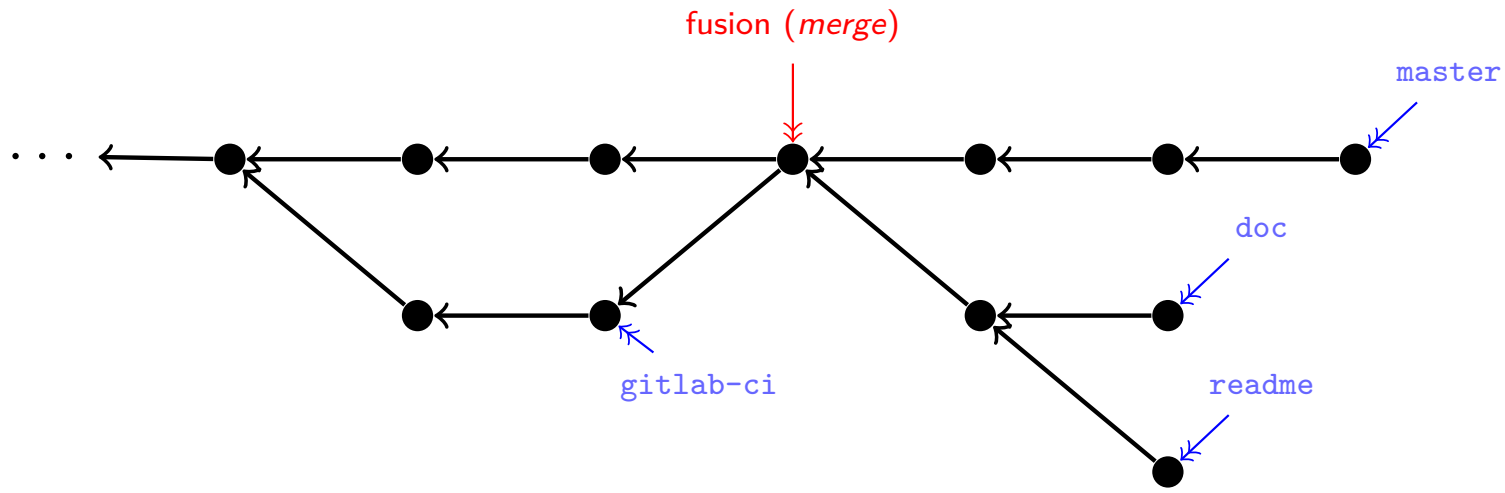
```
$ bats tests.bats --tap
1..4
ok 1 Addition
ok 2 Avec run
not ok 3 Plusieurs lignes
# (in test file tests.bats, line 15)
#   `[ "${lines[1]}" == "ligne2" ]   # Teste contenu de la 2e ligne
    ' failed
ok 4 Un test désactivé # skip
```

- **Plan**: `1..4`
- **Résultat**: `ok` ou `not ok`
- **Test ignoré**: `skip`
- **Commentaires**: avec #, etc.

Git

Historique d'un projet

Graphe orienté acyclique (DAG = *directed acyclic graph*)



4 types d'objets

- **Blob** = *Binary Large OBject*: données binaires
- **Arbre**: arborescence de blobs (≈ arborescence de fichiers)
- **Commit**: référence à un arbre
- **Annotation**: référence vers un *commit*

Commit

Métadonnées

- **Message**: décrit les modifications apportées
- **Auteur** (*author*): auteur original
- **Date de création**: date, heure, fuseau horaire
- **Livreur** (*committer*): personne qui a intégré le *commit*
- **Date de livraison**: date, heure, fuseau horaire

Identification

- **Clé**: 40 caractères hexadécimaux
- Produite par l'algorithme **SHA-1** (*Secure Hash Algorithm*)
- Probabilité que la clé soit **unique** $\rightarrow 1$

Exemple

- Souvent, **auteur** = **livreur**, mais pas toujours
- `git show`: voir informations sur un *commit*
- `--quiet`: cacher *diff*, `--pretty=fuller`: toutes les métadonnées

```
$ cd bats-core
$ git show --quiet --pretty=fuller 3b33a5a
commit 3b33a5ac6afd7f01ff4120659e2a72b851081178
Merge: 7b032e4 eb120d9
Author:     Sam Stephenson <sstephenson@gmail.com>
AuthorDate: Thu Oct 16 09:44:15 2014 -0500
Commit:     Sam Stephenson <sstephenson@gmail.com>
CommitDate: Thu Oct 16 09:44:15 2014 -0500

    Merge pull request #76 from jwerle/patch-1

    Update package.json

$ git show --quiet --pretty=fuller 3be8246 | head -n 5
commit 3be82466a7355b3a6f40f428d8c6520b63241593
Author:     Henrique Moody <henriquemoody@gmail.com>
AuthorDate: Wed Oct 30 22:10:00 2013 -0200
Commit:     Ross Duggan <rduggan@engineyard.com>
CommitDate: Wed Aug 13 14:32:35 2014 +0100
```

Références

Référence courante

- `HEAD`: référence vers l'état courant
- `HEAD~n`: référence vers le `n`-ième *commit* parent
- `detached HEAD`: référence à un *commit* non pointé par une branche

Branches

- **Référence** nommée vers un *commit*
- `master`: branche « principale » qui pointe vers le **1er** *commit*

Références distantes (*remote*)

- **Dépôt distant** nommé `origin` par défaut
- `origin/master`: branche « principale » du dépôt distant

Création d'un dépôt

Nouveau dépôt

```
git init
```

Copie d'un dépôt existant

- **Commande**: `git clone`
- *Fork*: gérée par l'hébergeur (GitLab, Github), pas par Git
- Deux **protocoles**: HTTPS et Git (SSH)

Répertoire spécial `.git`

- **Décentralisé**: tout l'historique y est contenu
- Chaque dépôt Git est **équivalent**
- Le répertoire qui contient le sous-répertoire `.git` peut donc être **déplacé** n'importe où

Consulter l'historique

Commandes fréquentes

- `git log`: journal détaillé des modifications
- `git log --graph --all --color`: historique en graphe
- `git show`: voir un *commit* spécifique
- `git diff`: différence entre deux versions

Astuce dans `.gitconfig`

Ajouter un synonyme (*alias*) pour la commande `git gr`:

```
[alias]
    gr = log --graph --full-history --all --color --pretty=tformat
        :"%x1b[31m%h%x09%x1b[32m%d%x1b[0m%x20%s%x20%x1b[33m(%an)%
        x1b[0m"
```

Exemple

```
$ cd bats
$ git gr
* c750877        (origin/issue-290-debian-fix) build: test parallel invocation in [...]
*   7f0b346      (HEAD -> master, origin/master, origin/HEAD) Merge pull request [...]
|\
| * 3e9fd9d      test: filter parallell warnings in --job ordering test (Andrew Martin)
| * 17ff3f1      fix: parallel invocation for debian (Martin Schulze)
* |   743b02b    Merge pull request #310 from dimo414/patch-2 (Andrew Martin)
|\ \
| * | aeeb090    Remove TODO leftover from 118391d8e (Michael Diamond)
|/ /
* |   c648a85    Merge pull request #291 from dimo414/patch-2 (Andrew Martin)
|\ \
| * | 220bb9d    docs: add to README `load` semantics (Michael Diamond)
* | |   9b0a9a5  Merge pull request #304 from dimo414/patch-3 (Andrew Martin)
|\ \ \
| |_|/
|/| |
| * | 0b57bca    Remove "See the Background section above" link (Michael Diamond)
|/ /
* |   b615ed8    Merge pull request #297 from martin-schulze-vireso/issue-290 [...]
|\ \
| * | 0deffcc    fix: parallel output (Martin Schulze)
|/ /
* |   90ce858    Merge pull request #296 from martin-schulze-vireso/issue-292 [...]
|\ \
| * | 338226a    fix: handle skip in teardown more gracefully (Martin Schulze)
| * | 0ec52d7    test: add failing test cases for `skip` (Andrew Martin)
| |/
* |   3a0d0d2    Merge pull request #288 from waterkip/fix-greadlinky (Andrew Martin)
[...]
```

État d'un projet

États possibles

- **Propre** (*clean*): aucun fichier versionné n'a été modifié
- **Modifié**: il y a eu certaines modifications
- **Indexé** (*staged*): certaines modifications ont été indexées

Connaître l'état d'un projet

```
$ git status      # Affichage long
$ git status -s   # Affichage compact
```

Astuce dans `.gitconfig`

Ajouter un synonyme pour `git st`:

```
[alias]
    st = status -s
```

Changer l'état du projet

Commande

```
git checkout
```

Astuce dans .gitconfig

```
[alias]
    co = checkout
```

Astuce dans .bashrc

```
# Retrieves current branch
function parse_git_branch_and_add_brackets {
    git branch --no-color 2> /dev/null | sed -e '/^[^*]/d' -e 's/*
        \(.*\)/\[\1\]\ /'
}
# Changes prompt
PS1="\e[36m\$(parse_git_branch_and_add_brackets)\e[32;1m\u\e[0m \e
    [33;1m[\w]\e[0m\n$ "
```

Exemples

```
$ git co master          # Se placer sur la branche principale

$ git co c648a85         # Se placer au commit c648a85
Note: checking out 'c648a85'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by performing another checkout.

$ git co origin/master  # Branche principale du dépôt distant
Note: checking out 'origin/master'.

You are in 'detached HEAD' state. You can look around, make experimental
[...]
```

- Possible d'**ajouter** des *commits*
- Et de créer des nouvelles **branches**
- Ou de **déplacer** des branches (*rebase*)
- On va y revenir plus tard

Préparer une sauvegarde (*commit*)

Cycle de développement de base:

1. `git st`: vérifier que l'état est **propre**
2. Apporter des modifications au projet (tâche **atomique**)
3. `git st`: vérifier l'état des fichiers modifiés
4. `git diff`: vérifier les modifications
5. `git add`: indexer les modifications **ou**
   `git add -p|--patch`: indexer des morceaux de modifications
6. `git commit`: valider la sauvegarde **ou**
   `git ci` (en ajoutant l'*alias* `ci = commit`)

Raccourci

- `git ci -a`: combiner l'indexation et la validation
- De **toutes** les modifications

Message de *commit*

Les 7 règles d'un bon message (plus de détails)

1. Limiter le sujet à **50 caractères**
2. Séparer le **sujet** du **corps** par une ligne vide
3. Commencer le sujet par une **majuscule**
4. Ne pas terminer le sujet avec un **point**
5. Utiliser l'**impératif** dans le sujet
6. Limiter à **72 caractères** la largeur du corps
7. Dans le corps, décrire **quoi?** et **pourquoi?** plutôt que *comment?*

– Ne pas **mélanger** les langues

Conventions

– Dans un projet existant, respecter **les conventions**
– **Exemple**: commits conventionnels (issu du projet Angular)

Réinitialiser l'état du projet

Fichier spécifique

`git checkout <fichier>`: annuler les modifications apportées à `<fichier>` depuis le dernier *commit*

Annuler l'indexation

`git reset`: annule les commandes `git add` précédentes

Annuler les modifications

- `git reset --hard`: restaure les fichiers avant modifications
- **Attention:** on ne peut pas revenir en arrière
- **Vérifier** avec `git diff` avant

Corriger un *commit*

Corriger le message

- `git ci --amend`
- Puis on réécrit le message

Corriger le contenu du dernier *commit*

- Apporter les **modifications** souhaitées normalement
- Puis faire `git ci --amend` plutôt que `git add`

Attention

- **Réécrit** l'historique
- À éviter si vous avez **partagé** des modifications

Récupérer et partager des modifications

Télécharger des historiques distants

`git fetch`

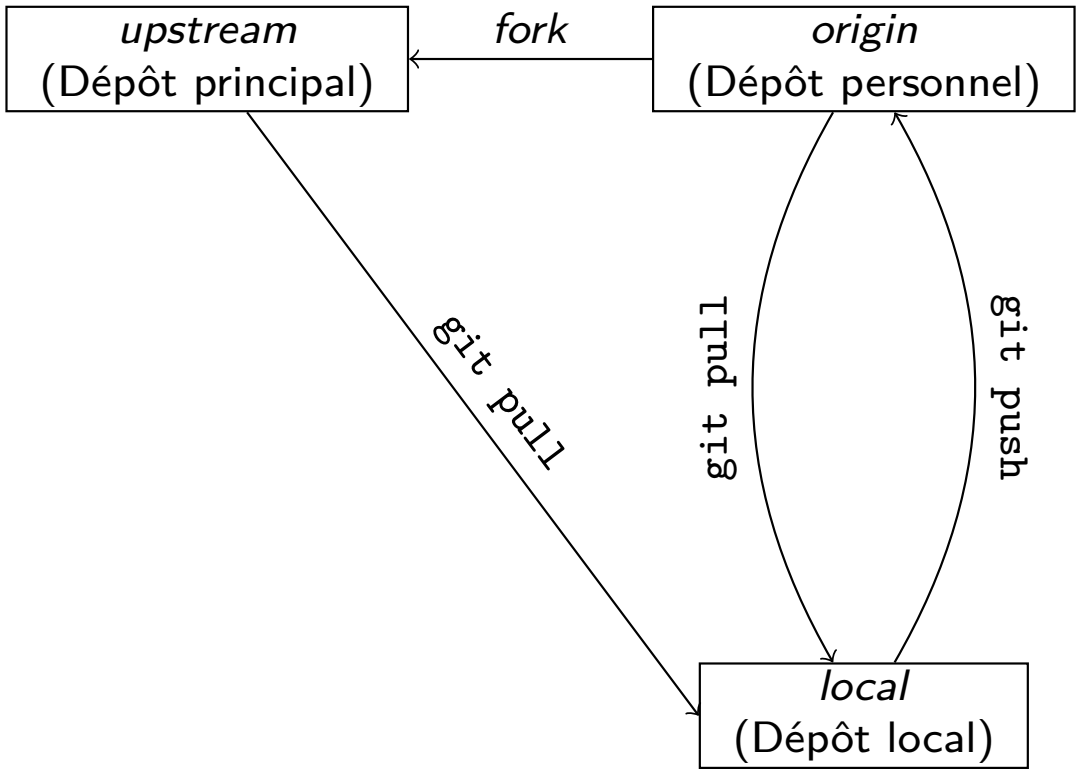
Fusionner deux historiques

- `git merge`: fusionne deux branches
- Peut entraîner des **conflits**
- On va y revenir
- `git pull`: tirer des modifications
- `git pull`: `git fetch` + `git merge`

Partager des modifications

- `git push`: « pousser » des modifications
- Doit souvent être précédé de `git pull` (conflits potentiels)

Se synchroniser avec des dépôts distants



- `git remote -v`: voir les informations sur dépôts distants
- `git remote add upstream <URL>`: ajouter un dépôt distant nommé `upstream`
- **Synchronisation** avec *upstream* sur `master`

GitLab-CI

Intégration continue

- En anglais, *continuous integration* (CI)
- Une **modification** à un logiciel ne devrait pas entraîner de régression
- Garantir la **compilation** (*build*)
- Mais aussi le **fonctionnement** (tests unitaires)
- Accepter seulement si **aucun test** n'échoue

Logiciels

- Jenkins: initialement pour Java, mais supporte plusieurs autres langages
- Travis CI: intégré directement à Github
- GitLab CI: intégré directement à GitLab

GitLab CI

- Documentation: `https://about.gitlab.com/gitlab-ci/`
- Tutoriel introductif: ici
- Mise en place: ajout d'un fichier nommé `.gitlab-ci.yml` qui respecte le format YAML et qui indique comment lancer les tests
- Lancés dans un « carré de sable » (*sand box*)
- Ce carré de sable est une **image** Docker
- Plusieurs images sont disponibles **par défaut**, mais vous pouvez aussi fournir vos propres images

Mise en place

- Ajout d'un fichier de configuration
- Et de scripts de vérification
- **Exemple**: lancement de la commande `make test`

Exemple

```
# Mise à jour de apt et installation de Bats
before_script:
    - apt-get update -qq
    - git clone "https://github.com/bats-core/bats-core.git" /tmp/bats
    - mkdir -p /tmp/local
    - bash /tmp/bats/install.sh /tmp/local
    - export PATH="$PATH:/tmp/local/bin"

# Pour vérifier la compilation
build:
    stage: build
    script:
        - make

# Tests unitaires
test:
    stage: test
    script:
        - make test
```

- Possible de spécifier l'**image** Docker
- Structuration des *pipelines*
- Peut conserver les **résultats** (artéfacts)
- On va y revenir

INF3135
Construction et maintenance de logiciels

Chapitre 3: Pointeurs

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Adresse et pointeur

Adresse

- Les données stockées en mémoire ont une **adresse**
- L'opérateur & (esperluette) retourne l'adresse d'une *left-value*

```c
#include <stdio.h>

int main(void) {
    int x = 210;
    printf("x           = %d\n", x);
    printf("&x          = %p\n", &x);
    printf("sizeof(x)   = %ld (= sizeof(int))\n", sizeof(x));
    printf("sizeof(&x) = %ld (4 ou 8, selon l'architecture)\n",
           sizeof(&x));
    return 0;
}
```

Résultat (le résultat peut varier):

```
x           = 210
&x          = 0x7ffef8a2c3e4
sizeof(x)   = 4 (= sizeof(int))
sizeof(&x) = 8 (4 ou 8, selon l'architecture)
```

Adresses dans une structure

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

struct player {
    char fname[20];
    char lname[20];
    bool active;
    unsigned int rank;
    double winrate;
};

int main(void) {
    struct player p = {"Novak", "Djokovic", true, 1, .95};
    printf("p.fname   = %-8s  &p.fname   = %p\n",  p.fname,   &p.fname);
    printf("p.lname   = %-8s  &p.lname   = %p\n",  p.lname,   &p.lname);
    printf("p.active  = %-8d  &p.active  = %p\n",  p.active,  &p.active);
    printf("p.rank    = %-8d  &p.rank    = %p\n",  p.rank,    &p.rank);
    printf("p.winrate = %-8lf  &p.winrate = %p\n", p.winrate, &p.winrate);
    return 0;
}
```

Résultat (le résultat peut varier):

```
p.fname   = Novak     &p.fname   = 0x7fff0eb44280
p.lname   = Djokovic  &p.lname   = 0x7fff0eb44294
p.active  = 1         &p.active  = 0x7fff0eb442a8
p.rank    = 1         &p.rank    = 0x7fff0eb442ac
p.winrate = 0.950000  &p.winrate = 0x7fff0eb442b0
```

Pointeur

- **Pointeur**: *left-value* qui contient une adresse
- **Déclaré** à l'aide du symbole *

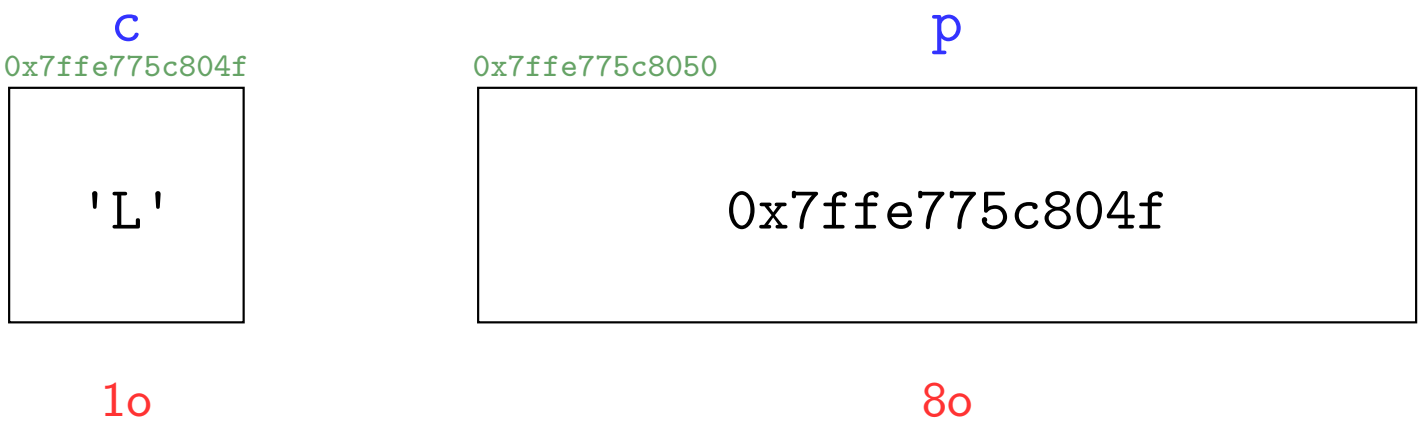
```c
#include <stdio.h>

int main(void) {
    char c = 'L', *p = &c;
    printf("c          = %c\n", c);
    printf("&c         = %p\n", &c);
    printf("p          = %p\n", p);
    printf("&p         = %p\n", &p);
    printf("sizeof(c) = %ld\n", sizeof c);
    printf("sizeof(p) = %ld\n", sizeof p);
    return 0;
}
```

Résultat (le résultat peut varier):

```
c          = L
&c         = 0x7ffe775c804f
p          = 0x7ffe775c804f
&p         = 0x7ffe775c8050
sizeof(c) = 1
sizeof(p) = 8
```

Représentation schématique

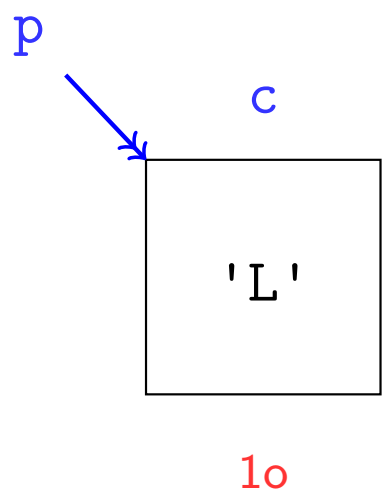
```
c           = L
&c          = 0x7ffe775c804f
p           = 0x7ffe775c804f
&p          = 0x7ffe775c8050
sizeof(c) = 1
sizeof(p) = 8
```



Avec variables

Représentation schématique

```
c           = L
&c          = 0x7ffe775c804f
p           = 0x7ffe775c804f
&p          = 0x7ffe775c8050
sizeof(c) = 1
sizeof(p) = 8
```



Avec flèches

Pointeurs typés

- **Exemples**: `char*`, `int*`, `double*`, `double**`, `enum answer*`, `struct player*`, `void*`, etc.
- **Intérêt?** détecter des erreurs à la compilation
- **Syntaxe**: coller `*` sur la variable et non le type:

```
int *p;         // Syntaxe préférée
int* p;         // Non recommandé
int *p1, *p2; // Deux pointeurs
int *p1, p2;  // p1 est un pointeur, p2 est un int
```

Plusieurs types de pointeurs

- **Pointeur nul**: identifié par la valeur `NULL`
- **Pointeur constant**: en lecture seule, écriture interdite
- **Pointeur générique**: `void*`
- **Pointeur de fonction**: permet de passer des fonctions en arguments

Espace mémoire

- Tous les pointeurs occupent le même **espace**
- Car contiennent des **adresses**

```c
#include <stdio.h>

enum answer { YES, NO, MAYBE };

int main(void) {
    char *c; int *i; double *d; enum answer *a;
    printf("sizeof c = %ld\n", sizeof c);
    printf("sizeof i = %ld\n", sizeof i);
    printf("sizeof d = %ld\n", sizeof d);
    printf("sizeof a = %ld\n", sizeof a);
    return 0;
}
```

Résultat (le résultat peut varier selon l'architecture):

```
sizeof c = 8
sizeof i = 8
sizeof d = 8
sizeof a = 8
```

Le qualificatif `const`

- `const <type> *p`: pointeur en lecture seule (*read-only*)
- Le contenu pointé peut être **lu**
- Mais on ne peut pas **écrire**
- **Intérêt?** détecter une écriture non souhaitée à la compilation

```
const int *p;  // Pointeur constant vers un entier
const char *s; // Chaîne de caractère non modifiable
```

- `<type> *const p`: pointeur constant
- La **valeur** du pointeur ne peut pas être modifiée
- **Tableau**: souvent converti (*decay*) en pointeur constant

```
// La signature suivante
void initialize_array(double a[], unsigned int n);
// est équivalente à
void initialize_array(double *const a, unsigned int n);
```

Pointeur « constant »?

- `const <type> *p`: pointeur en lecture seule
- `<type> *const q`: pointeur constant

```c
#include <stdio.h>

int main(void) {
    int i;
    const int *p = &i; // Le pointeur p ne doit pas modifier
                       // le contenu qu'il référence
    *p = 13;           // On tente de modifier le contenu de i
    int *const q;      // Le pointeur q ne doit pas être modifié
    q = &i;            // On tente de le modifier
    return 0;
}
```

Résultat:

```
pointeur_const.c: In function 'main':
pointeur_const.c:7:8: error: assignment of read-only location '*p'
     *p = 13;           // On tente de modifier le contenu de i
        ^
pointeur_const.c:9:7: error: assignment of read-only variable 'q'
     q = &i;            // On tente de le modifier
       ^
```

Opérations sur les pointeurs

Déréférencement

- **Déréférencement**: accès à la donnée « pointée »
- À l'aide de l'**opérateur** *

```c
#include <stdio.h>

int main(void) {
    double d = 1.5, *p = &d;
    printf("d      = %lf\n", d);
    printf("&d     = %p\n" , &d);
    printf("p      = %p\n" , p);
    printf("*p     = %lf\n", *p);
    printf("*(&d)  = %lf\n", *(&d));
    printf("&(*p)  = %p\n" , &(*p));
    return 0;
}
```

Résultat (le résultat peut varier):

```
d      = 1.500000
&d     = 0x7ffcbfa1e798
p      = 0x7ffcbfa1e798
*p     = 1.500000
*(&d)  = 1.500000
&(*p)  = 0x7ffcbfa1e798
```

Opérateur -> (sucre syntaxique)

- Soit `p` un pointeur vers une structure ayant un champ `champ`
- Alors on peut écrire `p->champ` plutôt que `(*p).champ`
- Toujours préférer `p->champ` à `(*p).champ`

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

struct point {
    double x;
    double y;
};

int main(void) {
    struct point p = {-1.5, 2.3};
    struct point *q = &p;
    printf("p.x = %-4lf  q->x = %-4lf  (*q).x = %-4lf\n",
           p.x, q->x, (*q).x);
    printf("p.y = %-4lf  q->x = %-4lf  (*q).y = %-4lf\n",
           p.x, q->x, (*q).x);
    return 0;
}
```

Résultat (le résultat peut varier):

```
p.x = -1.500000  q->x = -1.500000  (*q).x = -1.500000
p.y = -1.500000  q->x = -1.500000  (*q).y = -1.500000
```

Autres opérations

- =: affectation
- `(type*)`: conversion (implicite ou explicite)

Comparaison

- ==: égalité d'adresses
- !=: différence d'adresses
- <=, >=, <, >: comparaison d'adresses

Arithmétique

- +: pointeur décalé vers la « droite »
- -: pointeur décalé vers la « gauche »
- ++: incrémentation
- --: décrémentation

Affectation et conversion de pointeurs

- **Affectation**: `p = q`
- Si `p`, `q` sont de même type qualifié, fonctionne sans problème
- Si `p` est plus qualifié (`const`) que `q`, alors conversion **implicite**
- Sinon, une conversion **explicite** est requise

```c
#include <stdio.h>

int main(void) {
    int x = 8, *p = &x;      // p pointe vers l'entier 8
    double *q = p;           // Avertissement: pointeurs incompatibles
    double *r = (double*)p; // Conversion explicite (à éviter)
    return 0;
}
```

Résultat:

```
cast_pointer.c: In function 'main':
cast_pointer.c:5:17: warning: initialization from incompatible
    pointer type [-Wincompatible-pointer-types]
     double *q = p;      // Avertissement: pointeurs incompatibles
```

Prudence lors des conversions

Conversion de pointeur est à **éviter** sauf dans les cas suivants:

1. Vers un pointeur du **même type**, mais plus **qualifié** (`const`)

```
char *s = "linux"; // Pointeur vers une chaîne littérale
const char *t = s; // Conversion implicite (acceptable)
```

2. De/vers un pointeur `void*` (on va y revenir)
3. Vers la valeur `NULL`

```
int i = 8, *p = &i; // p pointe vers la valeur 8
p = NULL;           // p ne pointe vers aucune valeur
```

Tableaux et arithmétique des pointeurs

Adresse d'une valeur d'un tableau

On peut récupérer l'**adresse** d'une valeur dans un **tableau**

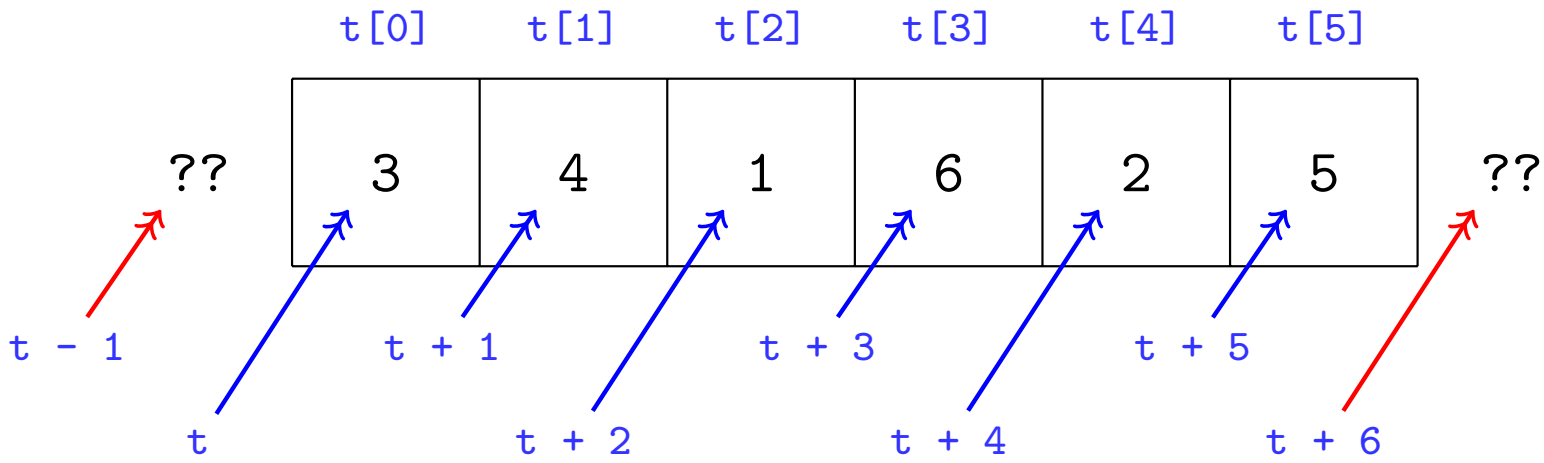
```c
#include <stdio.h>

int main(void) {
    int t[4] = {1, 2, 3, 4};
    for (unsigned int i = 0; i < 4; ++i)
        printf("t[%d] = %d, &t[%d] = %p\n",
               i, t[i], i, &t[i]);
    return 0;
}
```

Résultat (le résultat peut varier):

```
t[0] = 1, &t[0] = 0x7ffebfb89390
t[1] = 2, &t[1] = 0x7ffebfb89394
t[2] = 3, &t[2] = 0x7ffebfb89398
t[3] = 4, &t[3] = 0x7ffebfb8939c
```

Arithmétique des pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
```

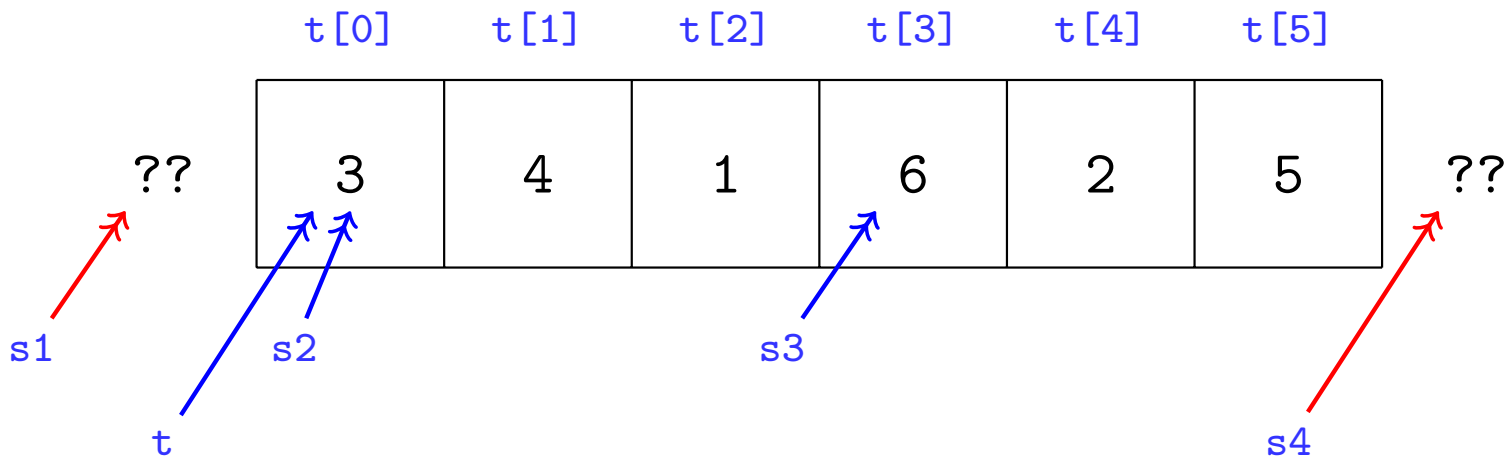


- `t`: pointe la première valeur du tableau
- `t + i`: pointe la `i`-ème valeur
- `t - 1`: pointe à « gauche » du tableau
- `t + 6`: pointe à « droite » du tableau
- Bref, `(t + i) == &t[i]`, ou `*(t + i) == t[i]`

Opérateurs de comparaison

- ==: indique si deux pointeurs contiennent la même adresse
- !=: négation de ==
- <=, >=, <, >: compare deux pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
int *s1 = t - 1, *s2 = t, *s3 = t + 3, *s4 = t + 6;
```



Les expressions `s2 == t`, `s1 != s2`, `s2 <= t`, `s3 >= s1`, `s3 < s4` et `s3 > t` sont **vraies**

Opérateurs + et −

- Soit `p` un pointeur de type `t*`
- `p + i`: décalé de `p` de `+ sizeof(t)` octets
- `p - i`: décalé de `p` de `- sizeof(t)` octets

```c
#include <stdio.h>

int main(void) {
    double values[] = {3.14, 2.71, 1.41, -0.5};
    double *p = &values[1];
    printf("values = [%lf, %lf, %lf, %lf]\n",
           values[0], values[1], values[2], values[3]);
    printf("p     = %p,  *p       = %lf\n", p, *p);
    printf("p + 1 = %p,  *(p + 1) = %lf\n", p + 1, *(p + 1));
    printf("p + 2 = %p,  *(p + 2) = %lf\n", p + 2, *(p + 2));
    printf("p - 1 = %p,  *(p - 1) = %lf\n", p - 1, *(p - 1));
    return 0;
}
```

Résultat:

```
values = [3.140000, 2.710000, 1.410000, -0.500000]
p     = 0x7ffea139f2c8,  *p       = 2.710000
p + 1 = 0x7ffea139f2d0,  *(p + 1) = 1.410000
p + 2 = 0x7ffea139f2d8,  *(p + 2) = -0.500000
p - 1 = 0x7ffea139f2c0,  *(p - 1) = 3.140000
```

Opérateurs ++ et --

- ++: incrémenter un pointeur vers le bloc suivant
- --: décrémenter le pointeur vers le bloc précédent
- Pratique pour **itérer** sur des tableaux

```c
#include <stdio.h>

int main(void) {
    char *s = "linux";
    double values[] = {3.14, 2.71, 1.41};
    for (char *p = s; // On initialise p au début de s
         *p != '\0';  // On répète tant qu'on ne rencontre pas '\0'
         ++p)         // On incrémente de sizeof(char) octet(s)
        printf("%c ", *p);
    printf("\n");
    for (double *p = &values[2]; // On initialise à la fin de values
         p >= values;            // On répète tant qu'on n'est pas au début
         --p)                    // On décrémente de sizeof(double) octets
        printf("%lf ", *p);
    return 0;
}
```

Résultat:

```
l i n u x
1.410000 2.710000 3.140000
```

Chaînes de caractères

Chaînes de caractères en C

- Cas **particulier** de tableau
- Ses éléments sont de type `char`
- Chaînes **littérales** délimitées par des guillemets `" "`
- Chaîne **bien formée**: doit terminer par le caractère `\0`
- Plusieurs **types** peuvent désigner une chaîne:

```
char s1[10];     // Tableau de caractères de taille fixe
char *s2;        // Pointeur vers début d'une chaîne
const char *s2; // Chaîne en lecture seule
```

Bibliothèques utiles

- `ctype.h`: manipulation de caractères
- `string.h`: manipulation de chaînes

La bibliothèque `ctype.h`

- `int isalpha(c)`: retourne une valeur $\neq 0$ ssi `c` est **alphabétique**
- `int isupper(c)`: retourne une valeur $\neq 0$ ssi `c` est **majuscule**
- `int islower(c)`: retourne une valeur $\neq 0$ ssi `c` est **minuscule**
- `int isdigit(c)`: retourne une valeur $\neq 0$ ssi `c` est un **chiffre**
- `int isalnum(c)`: retourne `isalpha(c) || isdigit(c)`
- `int isspace(c)`: retourne une valeur $\neq 0$ ssi `c` est un **espace**, un **saut de ligne**, un caractère de **tabulation**, etc.
- `int isprint(c)`: retourne une valeur $\neq 0$ ssi `c` est **affichable**
- `char toupper(c)`: retourne la lettre **majuscule** correspondant à `c`
- `char tolower(c)`: retourne la lettre **minuscule** correspondant à `c`

Remarque

Les fonctions `toupper` et `tolower` sont définies sur les **caractères** et non sur les **chaînes**

La bibliothèque `string.h`

Plusieurs **fonctions** disponibles:

- `strcat`: concatène une chaîne à la suite d'une autre
- `strncat`: concatène une chaîne à une autre en tronquant
- `strcpy`: copie une chaîne dans une autre
- `strncpy`: copie une chaîne dans une autre en tronquant
- `strlen`: longueur d'une chaîne
- `strcmp`: compare deux chaînes
- `strncmp`: compare deux chaînes en tronquant
- `strchr`: cherche un caractère dans une chaîne de gauche à droite
- `strrchr`: cherche un caractère dans une chaîne de droite à gauche
- `strstr`: cherche une chaîne dans une autre de gauche à droite
- `strrstr`: cherche une chaîne dans une autre de droite à gauche
- `strtok`: segmente une chaîne en morceaux (*tokens*)

Les fonctions `strcat` et `strncat` (1/3)

```
// Concatène la chaîne `src` à la suite de la chaîne `dest`
char *strcat(char *dest, const char *src);
```

- Commence à écrire sur le `'\0'` à la fin de `dest`
- Puis insère un caractère `'\0'` à la toute fin
- **Dangereuse**: attaque par dépassement de tampon (*buffer overrun*)

```
// Concatène au plus `n` caractères de la chaine `src` à la suite
    de la chaîne `dest`
char *strncat(char *dest, const char *src, size_t n);
```

- Même idée que `strcat`
- Mais utilise au plus les `n` premiers caractères de `src`
- Puis insère un caractère `'\0'` à la toute fin
- **Sécuritaire**: si `n` est choisie correctement

Les fonctions strcat et strncat (2/3)

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\0')        printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else                     printf(" ? ");
    printf("\n");
}

int main(void) {
    // strcat: résultat pas toujours bien formé
    char s[10] = "Linux "; printf("s[10] avant\n"); print_string(s, 10);
    printf("s[10] après\n"); strcat(s, "Mint"); print_string(s, 10);
    // strncat: sécuritaire
    char t[10] = "Linux "; printf("t[10] avant\n"); print_string(t, 10);
    unsigned int m = 10 - strlen(t) - 1;
    printf("t[10] après\n"); strncat(t, "Mint", m); print_string(t, 10);
    return 0;
}
```

Les fonctions `strcat` et `strncat` (3/3)

Résultat:

```
s[10] avant
 0   1   2   3   4   5   6   7   8   9
 L   i   n   u   x      \0  \0  \0  \0
s[10] après
 0   1   2   3   4   5   6   7   8   9
 L   i   n   u   x       M   i   n   t
t[10] avant
 0   1   2   3   4   5   6   7   8   9
 L   i   n   u   x      \0  \0  \0  \0
t[10] après
 0   1   2   3   4   5   6   7   8   9
 L   i   n   u   x       M   i   n  \0
```

- La chaîne `s` est **mal formée**
- Et on a écrit à un endroit **non réservé**
- La chaîne `t` est **bien formée**
- Car on a recopié seulement les `n - strlen(t) - 1` premiers caractères de `"Mint"`, où `n` est la capacité de `t`

Les fonctions `strcpy` et `strncpy` (1/3)

```c
// Copie la chaîne `src` dans la chaîne `dest`
char *strcpy(char *dest, const char *src);
```

- Puis insère un caractère `'\0'` à la toute fin
- **Dangereuse**: comme `strcat`, dépassement de tampon possible

```c
// Copie au plus `n` caractères de la chaîne `src` dans la chaîne `
    dest`
char *strncpy(char *dest, const char *src, size_t n);
```

- Copie au plus les `n` premiers caractères de `src`
- Puis insère un caractère `'\0'` à la toute fin
- **Sécuritaire**: si la capacité de `dest` est au moins `n + 1`

Les fonctions strcpy et strncpy (2/3)

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\0')        printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else                     printf(" ? ");
    printf("\n");
}

int main(void) {
    // strcpy: résultat pas toujours bien formé
    char s[6]; printf("s[6] avant\n"); print_string(s, 6);
    printf("s[6] après\n"); strcpy(s, "CentOS"); print_string(s, 6);
    // strncpy: plus sécuritaire
    char t[6]; printf("t[6] avant\n"); print_string(t, 6);
    printf("t[6] après\n"); strncpy(t, "CentOS", 5); print_string(t, 6);
}
```

Les fonctions `strcpy` et `strncpy` (3/3)

Résultat:

```
s[6] avant
 0  1  2  3  4  5
 ?  U \0 \0  ?  ?
s[6] après
 0  1  2  3  4  5
 C  e  n  t  O  S
t[6] avant
 0  1  2  3  4  5
\0  ?  ?  ? \0 \0
t[6] après
 0  1  2  3  4  5
 C  e  n  t  O \0
```

- La chaîne `s` est **mal formée**
- Et on a écrit à un endroit **non réservé**
- La chaîne `t` est **bien formée**
- Car on a recopié seulement les `n - 1` premiers caractères de `"Mint"`, où `n` est la capacité de `t`

La fonction strlen

```
// Retourne la longueur de la chaîne `s`
size_t strlen(const char *s);
```

– **Remarque**: parcourt toute la chaîne (complexité linéaire)

```
#include <stdio.h>
#include <string.h>

int main(void) {
    char s[40] = "Cette chaine est bien formee";
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));
    strcpy(s, "Celle-ci aussi");
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));
    strcat(s, ", meme plus longue");
    printf("s = %-40s  strlen(s) = %ld\n", s, strlen(s));
    return 0;
}
```

Résultat:

```
s = Cette chaine est bien formee             strlen(s) = 28
s = Celle-ci aussi                           strlen(s) = 14
s = Celle-ci aussi, meme plus longue         strlen(s) = 32
```

Les fonctions `strcmp` et `strncmp` (1/3)

```c
// Compare les chaînes s1 et s2
int strcmp(const char *s1, const char *s2);
```

- Implémente l'ordre **lexicographique** (dictionnaire)
- Induit par l'**alphabet ASCII**
- C'est une relation d'ordre **total**
- La valeur **retournée** est

$$\begin{cases} 0 & \text{si les chaînes s1 et s2 sont égales} \\ <0 & \text{si s1 précède s2 dans le dictionnaire} \\ >0 & \text{si s1 succède s2 dans le dictionnaire} \end{cases}$$

- Aucun problème si chaînes **bien formées**

```c
// Compare au plus les `n` premiers caractères de s1 et s2
int strncmp(const char *s1, const char *s2, size_t n);
```

Les fonctions strcmp et strncmp (2/3)

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

int main(void) {
    char *words[] = {"Ubuntu", "4+5=0", "uname", "lolcat"};
    for (unsigned int i = 0; i < 4; ++i) {
        for (unsigned int j = i; j < 4; ++j) {
            int c = strcmp(words[i], words[j]);
            printf("strcmp(%-6s, %-6s) = %-3d  =>  ",
                    words[i], words[j], c);
            printf("%-6s %s %-6s\n", words[i],
                    c == 0 ? "==" : (c < 0 ? "<" : ">"),
                    words[j]);
        }
    }
    return 0;
}
```

Résultat:

```
strcmp(Ubuntu, Ubuntu) = 0    =>  Ubuntu == Ubuntu
strcmp(Ubuntu, 4+5=0 ) = 33   =>  Ubuntu > 4+5=0
strcmp(Ubuntu, uname ) = -32  =>  Ubuntu < uname
strcmp(Ubuntu, lolcat) = -23  =>  Ubuntu < lolcat
strcmp(4+5=0 , 4+5=0 ) = 0    =>  4+5=0  == 4+5=0
strcmp(4+5=0 , uname ) = -65  =>  4+5=0  < uname
strcmp(4+5=0 , lolcat) = -56  =>  4+5=0  < lolcat
strcmp(uname , uname ) = 0    =>  uname  == uname
strcmp(uname , lolcat) = 9    =>  uname  > lolcat
strcmp(lolcat, lolcat) = 0    =>  lolcat == lolcat
```

Les fonctions strcmp et strncmp (3/3)

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

bool is_square(const char *s) {
    size_t n = strlen(s);
    if (n % 2 != 0) return false;
    size_t h = n / 2;
    return strncmp(s, s + h, h) == 0;
}

int main(void) {
    char *strings[] = {"abab", "123123", "1122", "aaabaaa", "aaabaaac"};
    for (unsigned int i = 0; i < 5; ++i) {
        unsigned int freq[10];
        printf("Is \"%s\" a square? %s\n",
               strings[i], is_square(strings[i]) ? "yes" : "no");
    }
    return 0;
}
```

Résultat:

```
Is "abab" a square? yes
Is "123123" a square? yes
Is "1122" a square? no
Is "aaabaaa" a square? no
Is "aaabaaac" a square? no
```

La fonction strchr

```c
#include <stdio.h>
#include <stdbool.h>
#include <string.h>

int main(void) {
    const char *p = "abracadabra";
    while (true) {
        p = strchr(p, 'a');
        if (p == NULL) break;
        printf("%s\n", p);
        ++p;
    }
    return 0;
}
```

Résultat:

```
abracadabra
acadabra
adabra
abra
a
```

La fonction strtok (1/2)

```c
#include <stdio.h>
#include <ctype.h>
#include <string.h>

void print_string(const char *s, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        printf("%2d ", i);
    printf("\n");
    for (unsigned int i = 0; i < n; ++i)
        if (s[i] == '\0')        printf("\\0 ");
        else if (isprint(s[i])) printf(" %c ", s[i]);
        else                     printf(" ? ");
    printf("\n");
}

int main(void) {
    char s[] = "-4,3a,2"; char *token;
    printf("Chaîne s = \"%s\" avant segmentation\n", s);
    print_string(s, 8);
    printf("\nSegmentation...\n");
    token = strtok(s, ","); // Premier appel de strtok sur s
    while (token != NULL) {
        printf("token = %s\n", token);
        token = strtok(NULL, ","); // Appels suivants sur NULL
    }
    printf("\n");
    printf("Chaîne s = \"%s\" après segmentation\n", s);
    print_string(s, 8);
}
```

La fonction `strtok` (2/2)

Résultat:

```
Chaîne s = "-4,3a,2" avant segmentation
 0  1  2  3  4  5  6  7
 -  4  ,  3  a  ,  2 \0

Segmentation...
token = -4
token = 3a
token = 2

Chaîne s = "-4" après segmentation
 0  1  2  3  4  5  6  7
 -  4 \0  3  a \0  2 \0
```

- La fonction `strtok` **modifie** la chaîne qu'elle segmente
- En écrasant les délimiteurs avec des caractères `'\0'`
- Pour des raisons d'**efficacité**
- **Copier** la chaîne avant segmentation pour la conserver

Autres fonctions

Standard

- `memcpy`, `memmove`, `memchr`, `memcmp`, `memset`: manipule des octets
- `strcoll`: compare deux chaînes selon la locale
- `strerror`: chaîne de caractère correspondant à une erreur
- `strspn`: longueur d'un préfixe contenant certains caractères
- `strcspn`: longueur d'un préfixe ne contenant pas certains caractères
- `strpbrk`: recherche d'octets
- `strxfrm`: transforme une chaîne

Non standard

`memccpy`, `mempcpy`, `strcat_s`, `strcpy_s`, `strdup`, `strerror_r`, `strerror_r`, `strlcat`, `strlcpy`, `strsignal`, `strsep`, `strtok_r`, etc.

Pointeurs de fonctions

Fonctions comme arguments de fonctions

- Passage de **fonctions** comme **arguments** d'autres fonctions
- Supporté dans plusieurs **langages**: Java, C++, Python, Haskell, ...
- **Exemples:** les fonctions `map` et `filter`

Python:

```
>>> map(len, ["alpha", "beta", "gamma"])
[5, 4, 5]
>>> is_palindrome = lambda s: s == s[::-1]
>>> filter(is_palindrome, ["radar", "allo", "ici", "ressasser"])
['radar', 'ici', 'ressasser']
```

Haskell:

```
Prelude> map (*3) [5,4,1,2,3]
[15,12,3,6,9]
Prelude> filter (<=3) [8,1,4,3,5,6,2,7]
[1,3,2]
```

Pointeur vers une fonction

- Les pointeurs de fonctions sont **typés**
- Ils peuvent être contenus dans des **structures**, des **tableaux**, etc.
- Attention à la **syntaxe**

```
// Pointeur de fonction de type int -> int
int (*f)(int x);
// Pointeur de fonction de type (int, int) -> int
int (*g)(int x, int y);
// Fonction de int -> int*
int *f(int x);
// Fonction de (int,int) -> int*
int *g(int x, int y);
```

La fonction map en C

```c
#include <stdio.h>

// Retourne le carré de `x`
int carre(int x) { return x * x; }

// Applique la fonction `f` sur le tableau `domaine` de taille `n`
// et stocke le résultat dans le tableau `image`
void map(const int *domaine, int *image, int (*f)(int), unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        image[i] = f(domaine[i]);
}

int main(void) {
    int domaine[5] = {2,3,5,7,11}, image[5];
    map(domaine, image, carre, 5);
    for (unsigned int i = 0; i < 5; ++i)
        printf("carre(%d) = %d\n", domaine[i], image[i]);
    return 0;
}
```

Résultat:

```
carre(2) = 4
carre(3) = 9
carre(5) = 25
carre(7) = 49
carre(11) = 121
```

Tableau de fonctions

```c
#include <stdio.h>

int carre(int x) {
    return x * x;
}

int cube(int x) {
    return x * x * x;
}

int main(void) {
    // f est un tableau contenant 2 pointeurs de fonctions
    // On accède aux fonctions avec f[ ]
    int (*f[2])(int) = {carre, cube};
    printf("%d %d %d\n", 4, f[0](4), f[1](4));
    return 0;
}
```

Résultat:

```
4 16 64
```

Tri rapide

- Dans `stdlib.h`: on trouve la fonction `qsort`:

```
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int (*compar)(const void *, const void *));
```

- `base`: pointeur vers le premier élément du tableau
- `nmemb`: nombre d'éléments dans le tableau
- `size`: taille individuelle d'un élément (utiliser `sizeof`)
- `compar` : pointeur de fonction qui compare deux éléments

Remarque

- Le type `void*` est utilisé pour une plus grande **généricité**
- En pratique, on doit faire des **conversions** (*cast*)

Utilisation de qsort

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

/* La fonction de comparaison doit avoir
 * la signature (const void *, const void *)
 */
int compare_ints(const void *a, const void *b) {
    // On doit donc faire des conversions à l'intérieur
    return *(int*)a - *(int*)b;
}

int main(void) {
    int a[] = {8,3,4,2,0,5};
    qsort(a, 6, sizeof(int), compare_ints);
    for (unsigned int i = 0; i < 6; ++i)
        printf ("%d ", a[i]);
    return 0;
}
```

Résultat:

```
0 2 3 4 5 8
```

Fouille binaire

- Toujours dans `stdlib.h`, on trouve la fonction `bsearch`:

```
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

- `key`: pointeur vers la valeur recherchée
- `base`: pointeur vers le premier élément du tableau
- `nmemb`: nombre d'éléments dans le tableau
- `size` : taille individuelle d'un élément
- `compar`: pointeur de fonction qui compare deux éléments

Utilisation de bsearch

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

int compare_ints(const void *a, const void *b) {
    return *(int*)a - *(int*)b;
}

int main(void) {
    int a[] = {2,5,7,8,13,15};
    int *p;
    int i, key;

    for (key = 1; key <= 7; ++key) {
        p = (int*)bsearch(&key, a, 6, sizeof(int), compare_ints);
        printf("%d %sest %sdans le tableau\n",
               key, p == NULL ? "n'" : "", p == NULL ? "pas " : "");
    }
    return 0;
}
```

Résultat:

```
1 n'est pas dans le tableau
2 est dans le tableau
3 n'est pas dans le tableau
4 n'est pas dans le tableau
5 est dans le tableau
6 n'est pas dans le tableau
7 est dans le tableau
```

INF3135
Construction et maintenance de logiciels

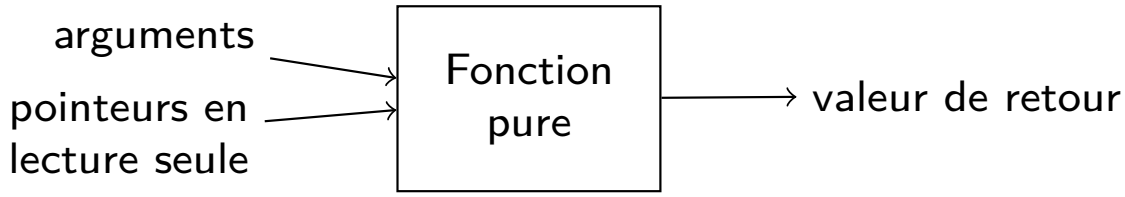
Chapitre 4: Entrées et sorties

Alexandre Blondin Massé

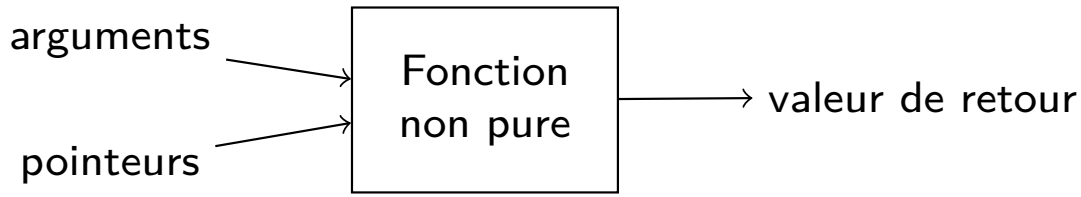
Université du Québec à Montréal
Département d'informatique

Été 2020

Entrées et sorties



Entrées



Sorties

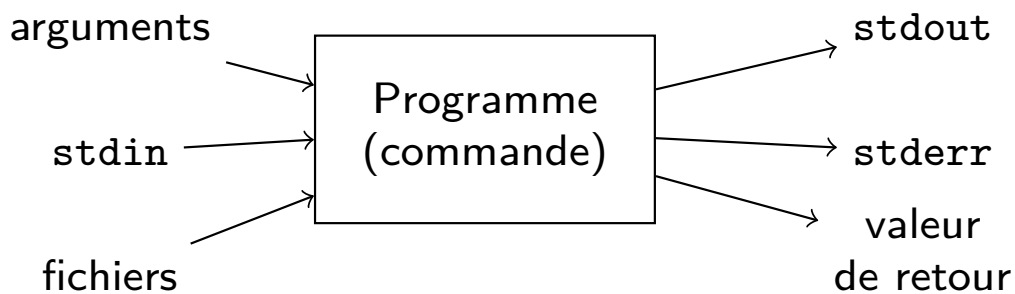


Table des matières

La bibliothèque `stdio.h`

La bibliothèque `stdio.h`

- *stdio* = *standard input output*
- Inclusion avec `#include <stdio.h>`

Macros:

- `EOF`: caractère de fin de fichier
- `stdin`, `stdout`, `stderr`: canaux standards
- `NULL`: pointeur nul, ...

Types:

- `FILE`: flux (*stream*)
- `size_t`: taille en octets
- `fpos_t`: position dans un flux, ...

Variables externes: `optarg`, `opterr`, `optind`, `optopt` (gestion des arguments)

Plusieurs dizaines de **fonctions** (`man stdio`)

Opérations générales (1/2)

Ouverture et fermeture:

```
// Ferme un flux
int       fclose(FILE *);
// Ouvre un flux avec un descripteur de fichier
FILE     *fdopen(int, const char *);
// Ouvre un flux
FILE     *fopen(const char *, const char *);
// Réouvre un flux
FILE     *freopen(const char *, const char *, FILE *);
```

Suppression et renommage:

```
// Supprime un fichier ou un répertoire
int       remove(const char *);
// Renomme un fichier ou un répertoire
int       rename(const char *, const char *);
```

Manipulation de fichier temporaires:

```
// Retourne un nom de fichier temporaire
char     *tmpnam(char *);
// Crée un fichier temporaire
FILE     *tmpfile(void);
```

Opérations générales (2/2)

Informations générales

```
// Vérifie si la fin du flux est atteinte
int       feof(FILE *);
// Récupère le descripteur d'un flux
int       fileno(FILE *);
// Récupère le nom du terminal courant
char     *ctermid(char *);
// Récupère l'utilisateur courant
char     *cuserid(char *);(LEGACY)
```

Gestion des erreurs:

```
// Réinitialise les indicateurs de fin de fichier et d'erreur
void      clearerr(FILE *);
// Vérifie si une erreur est survenue
int       ferror(FILE *);
// Écrit un message d'erreur sur stderr
void      perror(const char *);
```

Manipulation de caractères

```
// Lit le prochain caractère d'un flux
int       fgetc(FILE *);
// Équivalent à fgetc, avec passes multiples du flux
int       getc(FILE *);
// Écrit un caractère sur un flux
int       fputc(int, FILE *);
// Équivalent à fputc, avec passes multiples du flux
int       putc(int, FILE *);
// Remet un caractère sur un flux
int       ungetc(int, FILE *);
```

Canaux standards:

```
// Équivalent à getc(stdin)
int       getchar(void);
// Écrit un caractère sur stdout
int       putchar(int);
```

Manipulation d'octets (*binary stream*)

```
// Lit des octets provenant d'un flux
size_t   fread(void *, size_t, size_t, FILE *);
// Écrit des octets sur un flux
size_t   fwrite(const void *, size_t, size_t, FILE *);
// Lit un mot (int) provenant d'un flux
int      getw(FILE *);
// Écrit un mot (int) sur un flux
int      putw(int, FILE *);
```

Manipulation de chaînes de caractères

```
// Lit une ligne provenant d'un flux
char    *fgets(char *, int, FILE *);
// Écrit une chaîne sur un flux
int      fputs(const char *, FILE *);
```

Canaux standards:

```
// Lit une chaîne sur stdin (obsolète, deprecated)
// Préférer fgets(..., ..., stdin)
char    *gets(char *);
// Écrit une chaîne sur stdout
int      puts(const char *);
```

Lecture et écriture formatées

```c
// Écrit des données formatées sur un flux
int      fprintf(FILE *, const char *, ...);
// Lit des données formatées provenant d'un flux
int      fscanf(FILE *, const char *, ...);
// Équivalent à fprintf avec va_list
int      vfprintf(FILE *, const char *, va_list);
// Équivalent fprintf, mais sur stdout
int      printf(const char *, ...);
// Équivalent à fscanf, mais provenant de stdin
int      scanf(const char *, ...);
// Écrit des données formatées dans une chaîne
int      sprintf(char *, const char *, ...);
// Équivalent à sprintf, mais avec écriture tronquée
int      snprintf(char *, size_t, const char *, ...);
// Lit des données formatées depuis une chaîne
int      sscanf(const char *, const char *, int ...);
// Équivalent à printf, mais avec a va_list
int      vprintf(const char *, va_list);
// Équivalent à snprintf, mais avec va_list
int      vsnprintf(char *, size_t, const char *, va_list);
// Équivalent à sprintf, mais avec va_list
int      vsprintf(char *, const char *, va_list);
```

Navigation

```
// Modifie la position courante dans un flux
int      fseek(FILE *, long int, int);
// Modifie la position courante dans un flux (avec off_t)
int      fseeko(FILE *, off_t, int);
// Récupère la position courante dans un flux
long int ftell(FILE *);
// Récupère la position courante dans un flux (avec off_t)
off_t    ftello(FILE *);
// Modifie la position courante dans un flux
int      fsetpos(FILE *, const fpos_t *);
// Récupère la position courante dans un flux
int      fgetpos(FILE *, fpos_t *);
// Remet la position courante d'un flux au début
void     rewind(FILE *);
```

Autres fonctions (1/2)

Traitement des arguments:

```
// Traite les options d'une commande
int       getopt(int, char * const[], const char); (LEGACY)
```

Tampons (*buffer*):

```
// Vide un tampon
int       fflush(FILE *);
// Modifie la stratégie de tampon
// Possibilités: line buffered, unbuffered, fully buffered
int       setvbuf(FILE *, char *, int, size_t);
// Modifie la stratégie de tampon à unbuffered ou fully buffered
void      setbuf(FILE *, char *);
```

Tubes (communication inter-processus):

```
// Ferme un tube
int       pclose(FILE *);
// Ouvre un tube
FILE     *popen(const char *, const char *);
```

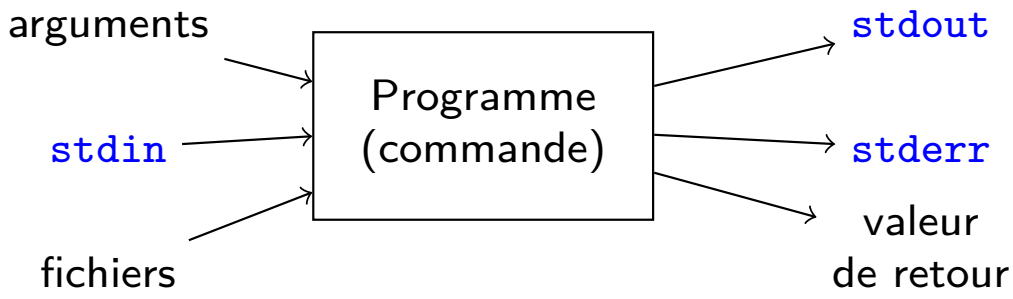
Autres fonctions (2/2)

Verrouillage:

```
// Verrouille un flux (thread-safety)
void      flockfile(FILE *);
// Version non bloquante de verouillage
int       ftrylockfile(FILE *);
// Déverrouille un flux
void      funlockfile(FILE *);
// Lit un caractère provenant d'un flux (non sécuritaire)
int       getc_unlocked(FILE *);
// Écrit un caractère (non sécuritaire)
int       putc_unlocked(int, FILE *);
// Équivalent à getc(stdin) (non sécuritaire)
int       getchar_unlocked(void);
// Écrit un caractère sur stdout (non sécuritaire)
int       putchar_unlocked(int);
```

Canaux standards

Canaux



3 canaux standards

- `stdin`: entrée standard (canal 0)
- `stdout`: sortie standard (canal 1)
- `stderr`: sortie d'erreur standard (canal 2)

Comportement par défaut (peut être redéfini)

- `stdin`: saisie clavier (*line buffered*)
- `stdout`: affichage sur le terminal (*line buffered*)
- `stderr`: affichage sur le terminal (*unbuffered*)

L'entrée standard (stdin)

- Un programme peut lire des données sur l'**entrée standard**
- Par défaut, lit la **saisie clavier**

```c
#include <stdio.h>

#define BUFFER_SIZE 20

int main(void) {
    char c1 = getchar(), c2 = getchar(), c3 = getchar();
    char line[BUFFER_SIZE];
    fgets(line, BUFFER_SIZE, stdin);
    printf("3 premiers caractères: %c %c %c\n", c1, c2, c3);
    printf("Reste de ligne: %s\n", line);
    return 0;
}
```

Résultat:

```
$ gcc stdin.c -o stdin && ./stdin
Git est un super logiciel!
3 premiers caractères: G i t
Reste de ligne:  est un super logic
```

La sortie standard (stdout)

- Un programme peut écrire des données sur la **sortie standard**
- Par défaut, l'affichage se fait sur le **terminal**

```c
#include <stdio.h>

int main(void) {
    char c = 'A';
    putchar(c);
    putchar('\n');
    puts("C est un langage particulier");
    fwrite("Incroyable!", sizeof(char), 4, stdout);
    printf("\n%s %c %p %lf\n",
           "Fantastique", '!', &c, 1.23456789);
    return 0;
}
```

Résultat:

```
$ gcc stdout.c -o stdout && ./stdout
A
C est un langage particulier
Incr
Fantastique ! 0x7ffe6e743c27 1.234568
```

La sortie d'erreur standard (stderr)

- On peut aussi écrire sur la **sortie d'erreur standard**
- Par défaut, l'affichage se fait aussi sur le **terminal**

```c
#include <stdio.h>

int main(void) {
    char c = 'A';
    fputc(c, stderr);
    fputc('\n', stderr);
    fputs("C est un langage particulier", stderr);
    fwrite("Incroyable!", sizeof(char), 4, stderr);
    fprintf(stderr, "\n%s %c %p %lf\n",
            "Fantastique", '!', &c, 1.23456789);
    return 0;
}
```

Résultat:

```
$ gcc stderr.c -o stderr && ./stderr
A
C est un langage particulierIncr
Fantastique ! 0x7ffd48babbd7 1.234568
```

Redirections (1/2)

- Par défaut, `stdin` lit la saisie clavier
- Et `stdout`/`stderr` écrivent sur le terminal
- Ces comportements peuvent être modifiés avec des **redirections**
- Les redirections sont gérées par le **shell**
- Elles ne sont donc **pas gérées** par `argc` et `argv`

Syntaxe

- `commande < fichier`: redirige `fichier` sur `stdin`
- `commande > fichier`: redirige `stdout` dans `fichier`
- `commande 2> fichier`: redirige `stderr` dans `fichier`

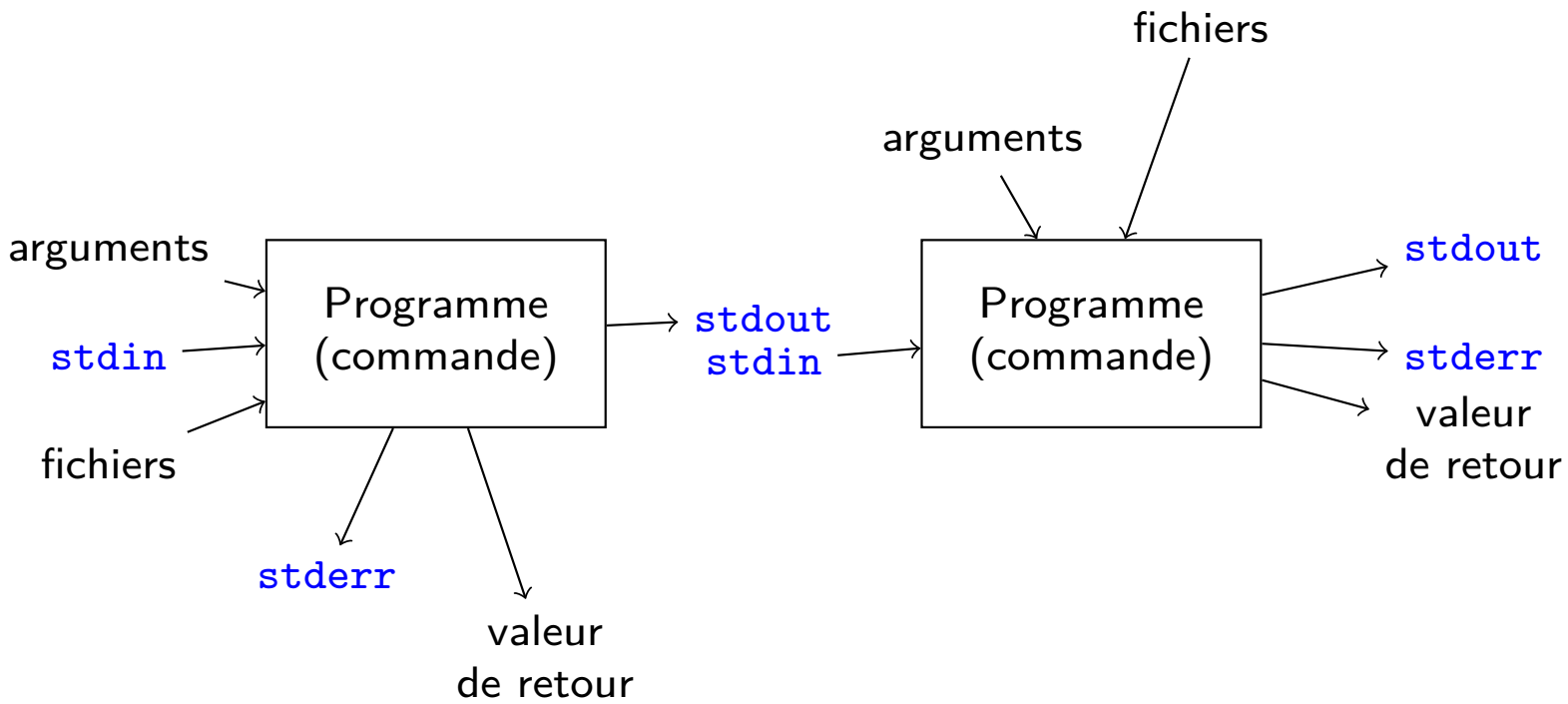
Redirections (2/2)

```c
#include <stdio.h>

int main(void) {
    char c = getchar();
    if (c == 'y')
        printf("Yes!\n");
    else
        fprintf(stderr, "No!\n");
    return 0;
}
```

```
$ cat redirections.y
yoyo
yaourt
$ cat redirections.z
zèbre
zoo
$ gcc redirections.c -o redirections
$ ./redirections < redirections.y
Yes!
$ ./redirections < redirections.z
No!
$ ./redirections < redirections.z 2> /dev/null
```

Tubes

- Permet d'**enchaîner** des programmes
- Le contenu écrit sur `stdout` par la première commande
- Est lu sur `stdin` par la deuxième commande
- **Syntaxe**: `commande1 | commande2 | ... | commandeN`



Filtres utiles

Filtre: programme souvent utilisé dans un tube

- `sort`: trie les lignes d'un flux
- `uniq`: supprime les doublons consécutifs
- `grep`: filtre selon une expression régulière
- `fmt`: formatte des données
- `pr`: formatte du texte pour impression
- `head`: affiche les premières lignes d'un flux
- `tail`: affiche les dernières lignes d'un flux
- `tr`: traduit caractère par caractère
- `sed`: transforme du texte
- `awk`: transforme du texte

Plus de détails dans **INF1070**

Consulter le manuel (`man`)

Exemple de filtres

Fichier `maj.c`:

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

int main(void) {
    char c;
    while ((c = getchar()) != EOF) {
        putchar(toupper(c));
    }
    return 0;
}
```

```
$ gcc maj.c -o maj
$ head -n 2 maj.c | ./maj
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
$ head -n 2 maj.c | ./maj | tail -n 1
#INCLUDE <CTYPE.H>
$ grep 'char' maj.c | ./maj
    CHAR C;
    WHILE ((C = GETCHAR()) != EOF) {
        PUTCHAR(TOUPPER(C));
```

Valeur de retour

Valeur de retour

- En Unix, tout programme retourne une **valeur entière**
- Lorsque son exécution est **terminée**

Sémantique

- 0: le programme s'est terminé **« normalement »**
- $\neq$ 0: le programme s'est terminé **« anormalement »**

Récupérer la valeur de retour

- Contenue dans la **variable spéciale** $?
- Valeur de retour de la **dernière commande**

Valeur de retour en C

Fonction `main`

- Valeur retournée à l'aide de `return`
- Doit être **entière**
- Peut être **négative**
- Par défaut, retourne 0
- **Bonne pratique**: toujours spécifier la valeur de retour

La fonction `exit`

```c
void exit(int status);
```

- Permet de **terminer** l'exécution du programme proprement
- Vide et ferme les **flux** encore ouverts
- Supprime les **fichiers temporaires**

Combinaisons de commandes

- On peut **combiner** des commandes avec ; && et ||
- Le comportement dépend de la **valeur de retour**
- ;: deux commandes consécutives indépendantes
- &&: 2e commande exécutée seulement si la 1re réussit
- ||: 2e commande exécutée seulement si la 1re échoue

```
$ echo "commande" && echo $?
commande
0
$ echo "commande" || echo $?
commande
$ cat fichier.inexistant && echo $?
cat: fichier.inexistant: No such file or directory
$ cat fichier.inexistant ; echo $?
cat: fichier.inexistant: No such file or directory
1
$ cat fichier.inexistant || echo $?
cat: fichier.inexistant: No such file or directory
1
```

Quelques commandes utiles

La suite Graphviz

- **Bibliothèques** et **programmes**
- Permettant de générer des **graphes**
- **Site officiel**: `https://graphviz.org/`
- **Dépôt Git**: sur GitLab
- **Licence**: Common Public License

Le format DOT

- Permet de décrire un graphe
- **Attributs**: noeuds, arcs, graphe, sous-graphe
- **Type d'attributs**: forme, couleur, police, espacement, ...

Utilisation

- En ligne de commande: `dot`, `neato`, `circo`
- Sous forme de bibliothèque C: `#include <gvc.h>`

Le format DOT

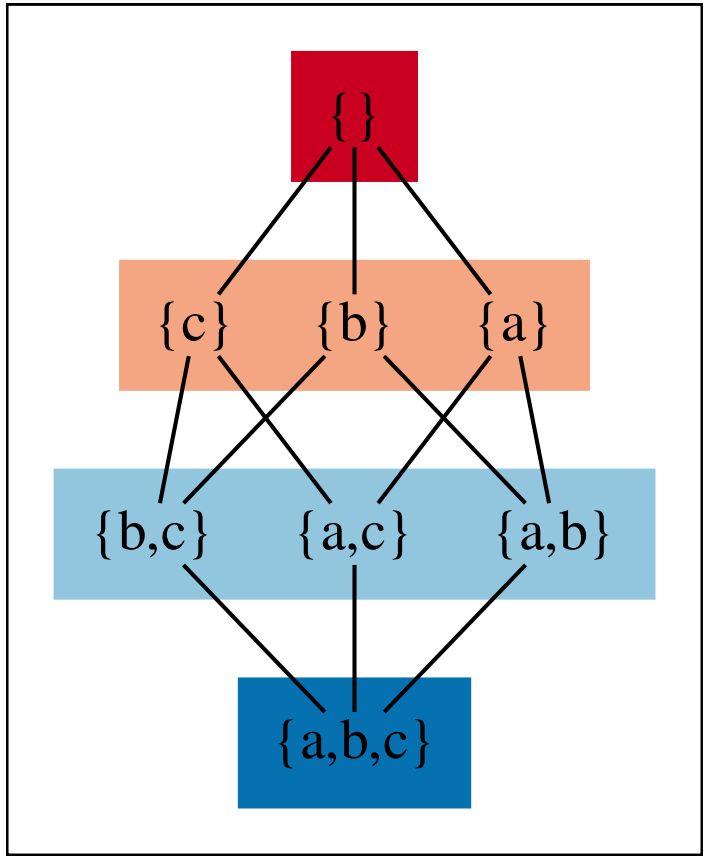
```
// Fichier graphviz.dot
digraph G {
    // Style des sous-graphes
    style=filled; labeljust=l; colorscheme=rdbu4;
    // Style des noeuds et des flèches
    node [shape=plain]; edge [arrowhead=none];
    // Sous-graphes
    subgraph cluster_0 {
        e [label="{}"];
        color=1;
    }
    subgraph cluster_1 {
        a [label="{a}"]; b [label="{b}"]; c [label="{c}"];
        color=2;
    }
    subgraph cluster_2 {
        ab [label="{a,b}"]; ac [label="{a,c}"]; bc [label="{b,c}"];
        color=3;
    }
    subgraph cluster_3 {
        abc [label="{a,b,c}"];
        color=4;
    }
    // Autres relations
    e -> {a b c};
    a -> ab; a -> ac;
    b -> ab; b -> bc;
    c -> ac; c -> bc;
    {ab ac bc} -> abc;
}
```

Invocation et résultat

Invocation:

```
$ dot -Tpdf -o graphviz.pdf < graphviz.dot
```

Résultat:



Le programme QPDF

- Programme en **ligne de commande**
- Permettant de manipuler des **documents pdf**: concaténation, extraction de pages, affichage d'information, rotation, chiffrement, ...
- **Site officiel**: http://qpdf.sourceforge.net/
- **Dépôt Git**: sur Github
- **Licence**: Apache

```
# Nombre de pages
$ qpdf --show-npages a.pdf
4
$ qpdf --show-npages b.pdf
7
# Concatène a.pdf et b.pdf et place le résultat dans o.pdf
$ qpdf --empty --pages a.pdf b.pdf -- o.pdf
$ ls
a.pdf  b.pdf  o.pdf
$ qpdf --show-npages o.pdf
11
# Extrait les pages 1, 3 et 4 de a.pdf
$ qpdf --empty --pages a.pdf 1,3-4 -- n.pdf
$ qpdf --show-npages n.pdf
3
```

La suite ImageMagick

- Permet de manipuler des **images matricielles** (*bitmap*)
- Peut aussi convertir en **format vectoriel** (PDF, SVG, ...)
- **Site officiel**: `https://imagemagick.org/index.php`
- **Dépôt Git**: sur Github
- Licence: personnalisée, de type *copyleft*

Plusieurs opérations

- Extraction d'**informations**: dimensions, format, ...
- **Transformations**: rognage, rotations, changements d'échelle, ...
- Application de **filtres**: flou, colorisation, convolutions, ...
- **Combinaison** d'images: concaténation, superposition, ...

Utilisation

- En ligne de commande: `magick`, `convert`, `montage`, `mogrify`, ...
- **API** pour plusieurs langages de programmation

Exemples

```
# Redimensionne une image
$ mogrify -resize 50% photo.jpg

# Transforme une image couleur en niveau de gris
$ convert image.png -set colorspace Gray -separate \
>         -average result.png

# Produit une animation GIF (0.2 seconde par image)
$ convert -delay 20 -loop 0 image*.gif animation.gif

# Produit une image avec le mot ImageMagick
$ convert -size 300x60 xc:skyblue -fill white -stroke black \
>         -pointsize 40 -gravity center \
>         -draw "text 0,0 'ImageMagick'" text.png

# Combine plusieurs images dans une matrice (spritesheet)
$ montage walking*.png -tile 4x2 -geometry 128x128+0+0 \
>         -background transparent walking-spritesheet.png
```

La suite `ffmpeg`

- Permet de manipuler des fichiers **audio** et **vidéo**
- **Site officiel**: https://ffmpeg.org/
- **Dépôt Git**: plusieurs dépôts
- Licence: de type GPL

```
# Convertit un fichier MKV au format MP4
$ ffmpeg -i input.mkv -codec copy output.mp4

# Affiche la durée d'un vidéo au format H:M:S:MS
$ ffprobe -i input.mp4 -sexagesimal -show_entries format=duration \
>         -v quiet -of csv="p=0"
0:15:14.400000

# Supprimer les 2 premières secondes d'un vidéo
$ ffmpeg -i input.mkv -ss 2 copy output.mkv

# Concatène 3 vidéos MP4 en un seul
$ ffmpeg -i "concat:input1.mp4|input2.mp4|input3.mp4" \
>        -c copy output.mp4
```

Le programme Gnuplot

- Permet de générer des **graphiques** statistiques
- Plusieurs **formats** de sortie supportés: PNG, PDF, SVG, ...
- **Site officiel**: `http://www.gnuplot.info/`
- **Dépôt Git**: sur Github
- Licence: personnalisée, permissive, mais limites sur la distribution

Plusieurs types de graphiques

- Histogrammes, lignes brisées, diagrammes à secteurs, ...
- Tracé de fonctions, de fonction paramétrées, fonction implicites
- Courbes dans l'espace, surfaces, ...

Utilisation

- De façon interactive en lançant `gnuplot`
- À l'aide d'un **script**

Exemple

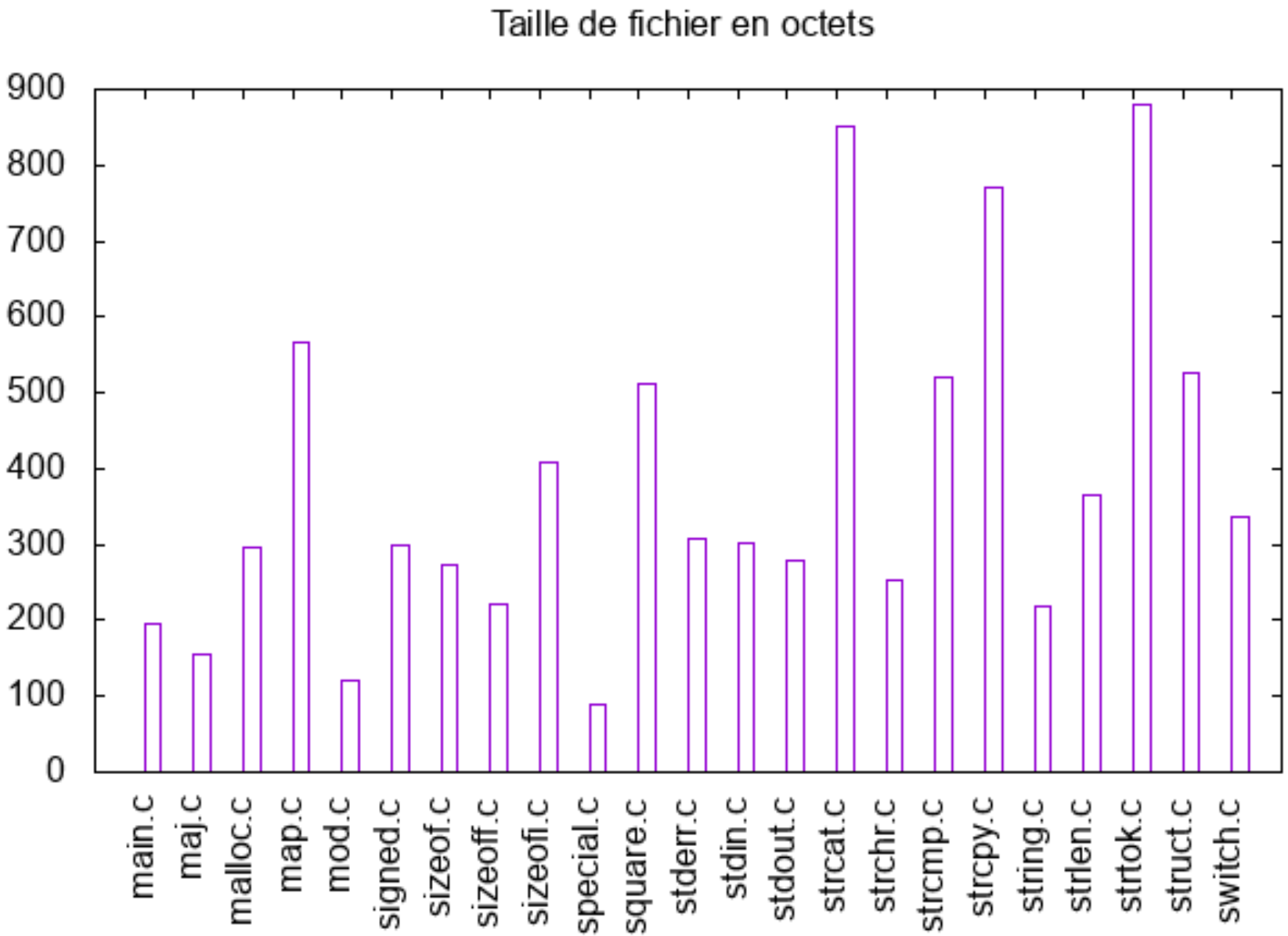
Fichier `gnuplot.gp`:

```
# Sélectionne le format de sortie
set terminal png
# Supprime la légende
set nokey
# Spécifie le titre
set title "Taille de fichier en octets"
# Rotation de 90 degrés des noms de fichier
set xtics rotate
# Spécifie le style de graphique (histogramme)
set style data histograms
# Lit les données sur stdin
# La colonne 1 en abscisse et la colonne 2 en ordonnée
plot '/dev/stdin' using 2:xtic(1)
```

Invocation:

```
# La commande stat affiche des statistiques sur des fichiers
#     %n: nom du fichier, %s: taille du fichier en octets
#     [ms]*.c: tous les fichiers commençant par m ou s
#              et finissant par .c
$ stat -c '%n %s' [ms]*.c | gnuplot gnuplot.gp > histogram.png
```

Résultat


Taille de fichier en octets
900
800
700
600
500
400
300
200
100
0
main.c
maj.c
malloc.c
map.c
mod.c
signed.c
sizeof.c
sizeoff.c
sizeofi.c
special.c
square.c
stderr.c
stdin.c
stdout.c
strcat.c
strchr.c
strcmp.c
strcpy.c
string.c
strlen.c
strtok.c
struct.c
switch.c

INF3135
Construction et maintenance de logiciels

Chapitre 5: Structures de données

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Généralités

Structure de données

Organisation **logique** d'un ensemble de données

Plusieurs objectifs

- **Simplifier** le traitement
- Offrir des opérations **efficaces**
- Économiser de l'espace **mémoire**

Interface et implémentation

- **Interface**: opérations supportées (type abstrait)
- **Implémentation**: organisation des données en mémoire, actions effectuées pour réaliser les opérations

Type abstrait: exemples

- **Pile** (*stack*): principe *last in first out (LIFO)*
- **File** (*queue*): principe *first in first out (FIFO)*
- **File à priorité** (*priority queue*): la priorité des éléments peut être augmentée ou diminuée
- **Liste**: les éléments sont ordonnés, on peut avoir des opérations d'accès, d'insertion, de suppression, ...
- **Ensemble** (*set*): aucune donnée répétée (doublon), vérification d'appartenance d'éléments, données ordonnées ou non, etc.
- **Tableau associatif** (*map*): un ensemble de paires clé-valeur, les clés doivent être uniques, les valeurs peuvent être répétées
- **Partition**: division d'un ensemble en parties, fusion entre parties, vérifier si deux éléments sont dans la même partie, ...
- **Graphe**: relations symétriques (graphes non orientés) ou non symétrique (graphes orientés) entre entités
- etc.

Implémentation: exemples

- **Tableau statique**: mémoire allouée et fixe, capacité maximale permise
- **Tableau dynamique**: tableau compressé ou allongé selon les besoins
- **Liste simplement chaînée**: chaque élément a une référence au suivant
- **Liste doublement chaînée**: chaque élément a une référence à l'élément précédent et à l'élément suivant
- **Structure arborescente**: arbres binaires, arbres préfixes, arbres suffixes, arbres d'arité quelconque, arbres coloriés, arbres-kd, etc.
- **Tableau multidimensionnel**: statiques ou dynamiques
- **Liste d'adjacence**: matrice creuse, graphes
- etc.

Invariants et opérations

Invariant

- **Propriété** qui doit être satisfaite en tout temps
- Généralement vérifiable à l'aide d'une **fonction** booléenne

Opération

- Toute fonction qui **modifie** la structure de données
- Doit toujours préserver les **invariants**

Exemples

- **Chaîne de caractères**: termine par `'\0'`
- **Liste simplement chaînée**: le dernier noeud pointe vers `NULL`
- **Arbre binaire de recherche**: les clés respectent l'ordre, ...

Table des matières

Allocation dynamique

Allocation dynamique

- Jusqu'à maintenant: mémoire réservée de façon **statique**
- Or, cette information n'est **pas toujours connu** à l'avance
- **Solution**: allouer l'espace mémoire de façon **dynamique**
- Dans la bibliothèque `stdlib.h`:

```
// Réserve un bloc de taille `size`
void *malloc(size_t size);
// Libère l'espace mémoire pointé par `ptr`
void free(void *ptr);
// Réserve un bloc de taille `nmemb * size` initialisé à 0
void *calloc(size_t nmemb, size_t size);
// Redimensionne un bloc de taille `size` déjà alloué dynamiquement
void *realloc(void *ptr, size_t size);
// Redimensionne un bloc de taille `nmemb * size`
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Les fonctions `malloc` et `calloc`

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

- Réserve sur le **tas** (*heap*) un bloc de mémoire
- De **taille** `size`
- **Retourne** un pointeur vers ce bloc
- **Retourne** `NULL` s'il n'y a plus d'espace mémoire

```
void *calloc(size_t nmemb, size_t size);
```

- Réserve `nmemb` blocs de mémoire **consécutifs**
- De taille **individuelle** `size`
- **Initialise** toutes les valeurs à 0
- **Retourne** un pointeur vers ce bloc
- **Retourne** `NULL` s'il n'y a plus d'espace mémoire

La fonction `free`

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

- **Libère** l'espace mémoire pointé par `ptr`
- Réservé lors d'un appel **précédent** à `malloc` ou `calloc`
- La taille libérée est **égale** à celle réservée
- Si `ptr == NULL`, alors rien ne se passe

Attention

- Si `free` a déjà été appelé sur `ptr`
- Ou si la mémoire pointée par `ptr` n'a pas été allouée précédemment

Alors le comportement est **indéfini**.

Les fonctions `realloc` et `reallocarray`

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

- **Redimensionne** un bloc de mémoire à la taille `size`
- **Préalablement** réservé avec `malloc` ou `calloc`
- **Retourne** un pointeur vers le bloc redimensionné
- **Retourne** `NULL` s'il n'y a plus d'espace mémoire

```
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

- **Équivalent** à `realloc(ptr, nmemb * size)`

Attention

- La valeur des **octets** présents avant et après est **préservée**
- Si **agrandissement**, les « nouveaux » octets sont **indéterminés**
- Pointeur retourné peut être **différent** du pointeur en 1er argument

Gestion de la mémoire

Fuite de mémoire (*memory leak*)

- Mémoire **réservée** mais non **référencée**
- Provoquée lorsqu'on appelle `malloc` ou `calloc`
- Et qu'on oublie de libérer avec `free`
- **Attention**: souvent « caché » derrière une autre fonction

Exemples

- Initialisation d'une **structure de données** dynamique
- Utilisation de la fonction `strdup` (duplication de chaîne)
- Bibliothèque SDL: `SDL_Init`

Comment les éviter?

- S'assurer que tout **appel** à `malloc` ou `calloc`
- Est **couplé** à un appel de la fonction `free`
- Souvent à l'aide de **fonctions**

Responsabilité de mémoire

- Si une fonction utilise `malloc` sans `free` associé
- Le comportement doit être **documenté** (*docstring*)
- **Exemple**: la fonction `strdup`

  The strdup() function returns a pointer to a new string which is a duplicate of the string s. Memory for the new string is obtained with malloc(3), and can be freed with free(3).

- Fournir une fonction **complémentaire** qui libère l'espace alloué
- **Exemple**: `SDL_Quit` est l'« inverse » de `SDL_Init`

Attention

- Habitude des langages avec **ramasse-miettes** (*garbage collector*)
- Dans lequel on utilise `new` sans ménagement
- Préférer un passage par **adresse**
- Et utiliser `malloc/calloc/free` seulement lorsqu'inévitable

L'outil Valgrind

- Cadriciel permettant de concevoir des outils d'**analyse dynamique**
- Permet de détecter des erreurs de **gestion de mémoire**
- Et de **profiler** un programme en détail
- **Lien officiel**: http://valgrind.org/
- **Invocation**:

```
$ valgrind [options valgrind] [programme] [options programme]
```

Plusieurs dizaines d'**options**:

- `--tool=<toolname>`: outil (par défaut, `memcheck`)
- `--leak-check=<no|summary|yes|full>`: vérifier fuites mémoires
- `--time-stamp=<yes|no>`: afficher chronologie
- `--track-origins=<yes|no>`: origine des valeurs non initialisées
- etc.

Exemple (1/5)

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

double *sum(const double *v1, const double *v2, unsigned int n) {
    double *v = malloc(n * sizeof(double));
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
    return v;
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5,  3.4};
    double v2[] = {-1.0,  2.1, -0.8};
    print_vector(sum(v1, v2, 3), 3);
    return 0;
}
```

Résultat:

```
[ 1.000000 0.600000 2.600000 ]
```

Exemple (2/5)

```
$ gcc sum_malloc.c -o sum_malloc
$ valgrind ./sum_malloc
$ valgrind --leak-check=yes ./sum_malloc
==6724== Memcheck, a memory error detector
==6724== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==6724== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==6724== Command: ./sum_malloc
==6724==
[ 1.000000 0.600000 2.600000 ]==6724==
==6724== HEAP SUMMARY:
==6724==     in use at exit: 24 bytes in 1 blocks
==6724==   total heap usage: 2 allocs, 1 frees, 1,048 bytes allocated
==6724==
==6724== 24 bytes in 1 blocks are definitely lost in loss record 1 of 1
==6724==    at 0x4C2FB0F: malloc (in /usr/lib/valgrind/[...]
==6724==    by 0x10876B: sum (in [...]/code/sum_malloc)
==6724==    by 0x1088BE: main (in [...]/code/sum_malloc)
==6724==
==6724== LEAK SUMMARY:
==6724==    definitely lost: 24 bytes in 1 blocks
==6724==    indirectly lost: 0 bytes in 0 blocks
==6724==      possibly lost: 0 bytes in 0 blocks
==6724==    still reachable: 0 bytes in 0 blocks
==6724==         suppressed: 0 bytes in 0 blocks
==6724==
==6724== For counts of detected and suppressed errors, rerun with: -v
==6724== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

Exemple (3/5)

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

double *sum(const double *v1, const double *v2, unsigned int n) {
    double *v = malloc(n * sizeof(double));
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
    return v;
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5,  3.4};
    double v2[] = {-1.0,  2.1, -0.8};
    double *v = sum(v1, v2, 3);
    print_vector(v, 3);
    free(v);
    return 0;
}
```

Résultat:

```
[ 1.000000 0.600000 2.600000 ]
```

Exemple (4/5)

```
$ gcc sum_malloc_free.c -o sum_malloc_free
$ valgrind --leak-check=yes ./sum_malloc_free
==10376== Memcheck, a memory error detector
==10376== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==10376== Using Valgrind-3.13.0 and LibVEX; rerun with -h for copyright info
==10376== Command: ./sum_malloc_free
==10376==
[ 1.000000 0.600000 2.600000 ]==10376==
==10376== HEAP SUMMARY:
==10376==     in use at exit: 0 bytes in 0 blocks
==10376==   total heap usage: 2 allocs, 2 frees, 1,048 bytes allocated
==10376==
==10376== All heap blocks were freed -- no leaks are possible
==10376==
==10376== For counts of detected and suppressed errors, rerun with: -v
==10376== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

Allocation dynamique

- Est-ce que `malloc` est vraiment nécessaire ici?
- **Réponse**: non!

Exemple (5/5)

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

void compute_sum(const double *v1, const double *v2,
                 double *v, unsigned int n) {
    for (unsigned int i = 0; i < n; ++i)
        v[i] = v1[i] + v2[i];
}

void print_vector(const double *v, unsigned int n) {
    printf("[ ");
    for (unsigned int i = 0; i < n; ++i)
        printf("%lf ", v[i]);
    printf("]");
}

int main(void) {
    double v1[] = { 2.0, -1.5,  3.4};
    double v2[] = {-1.0,  2.1, -0.8};
    double v[3];
    compute_sum(v1, v2, v, 3);
    print_vector(v, 3);
    return 0;
}
```

Résultat:

```
[ 1.000000 0.600000 2.600000 ]
```

Piles

Pile

- Structure de donnée **fondamentale**
- Stratégie *LIFO* = *last in first out*

Interface minimale (pile de caractères)

```
// Initialize the stack
void stack_initialize(stack *s);
// Is stack empty?
bool stack_is_empty(const stack *s);
// Push a value on top
void stack_push(stack *s, char value);
// Pop the value from the top
char stack_pop(stack *s);
// Delete a stack
void stack_delete(stack *s);
```

Implémentation

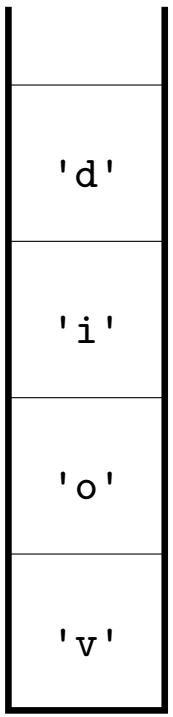
- **Tableau** statique ou dynamique
- Liste **simplement chaînée**

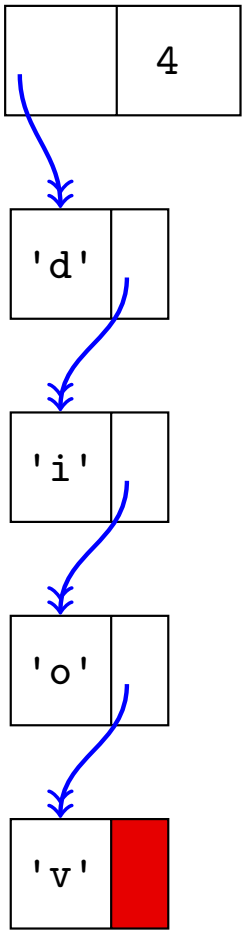
Représentation

Déclaration de types en C:

```
// Node
struct stack_node {
    char value;
    struct stack_node *next;
};

// Stack
typedef struct {
    struct stack_node *first;
    unsigned int size;
} stack;
```

Représentation schématique:





Invariants

Trois invariants

Pour toute **pile** `s` et pour tout **noeud** `node` de `s`,

- `node.next == NULL` **ssi** `node` est le dernier noeud de `s`
- `s.first == NULL` **ssi** `stack_is_empty(s)` est *vrai*
- Le nombre de noeuds dans `s` est donné par `s.size`

Utilité?

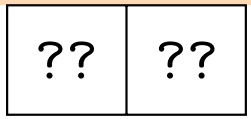
- Quand on implémente `stack_push` et `stack_pop`
- **Supposer** les invariants satisfaits en **début de fonction**
- Et s'assurer qu'ils sont encore satisfaits à la **fin**

Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

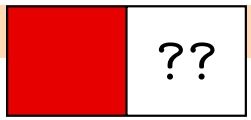
Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



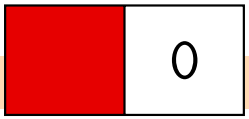
Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



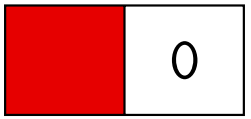
Implémentation (1/2)

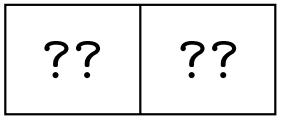
```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



Implémentation (1/2)

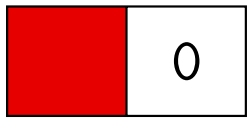
```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

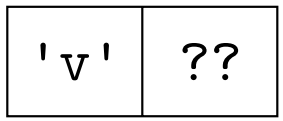




Implémentation (1/2)

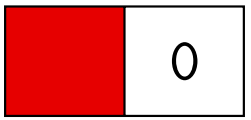
```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```

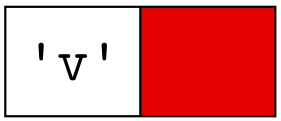




Implémentation (1/2)

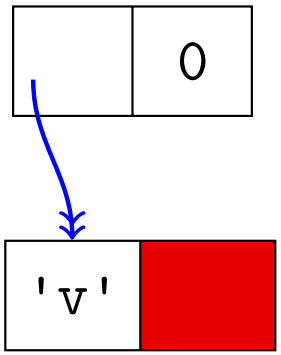
```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```





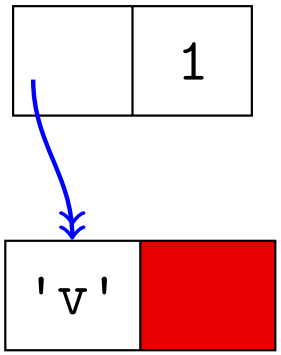
Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



Implémentation (1/2)

```
// Initialize the stack
void stack_initialize(stack *s) {
    s->first = NULL;
    s->size = 0;
}

// Is stack empty?
bool stack_is_empty(const stack *s) {
    return s->size == 0;
}

// Push a value on top
void stack_push(stack *s, char value) {
    struct stack_node *node = malloc(sizeof(struct stack_node));
    node->value = value;
    node->next = s->first;
    s->first = node;
    ++s->size;
}
```



Implémentation (2/2)

```
// Pop the value from the top
char stack_pop(stack *s) {
    if (!stack_is_empty(s)) {
        char value = s->first->value;
        struct stack_node *node = s->first;
        s->first = node->next;
        free(node);
        --s->size;
        return value;
    } else {
        fprintf(stderr, "Cannot pop from empty stack\n");
        exit(1);
        return '?';
    }
}

// Delete a stack
void stack_delete(stack *s) {
    while (!stack_is_empty(s)) stack_pop(s);
}
```

Exemple d'utilisation: parenthésage équilibré

```
/**
 * Returns true if and only if an expression is balanced
 *
 * @param expr  The expression to check
 * @returns     True if and only if the expression is balanced
 */
bool is_balanced(char *expr) {
    bool balanced = true;
    stack s;
    stack_initialize(&s);
    for (unsigned int i = 0; balanced && expr[i] != '\0'; ++i) {
        if (expr[i] == '(') {
            stack_push(&s, ')');
        } else if (expr[i] == '[') {
            stack_push(&s, ']');
        } else if (expr[i] == '{') {
            stack_push(&s, '}');
        } else if (expr[i] == ')' || expr[i] == ']' || expr[i] == '}') {
            if (stack_is_empty(&s))
                balanced = false;
            else
                balanced = expr[i] == stack_pop(&s);
        }
    }
    balanced = balanced && stack_is_empty(&s);
    stack_delete(&s);
    return balanced;
}
```

Tableaux dynamiques

Tableau dynamique

- Tableau dont la taille **varie** selon l'usage
- Supporté **par défaut** dans plusieurs langages
- **Exemples**: C++ (`vector`) Java (`ArrayList`), Python (listes)

Implémentation

- **Capacité** (*capacity*): taille « réelle » du tableau en mémoire
- **Taille** (*size*): nombre d'éléments « pertinents » dans le tableau

Redimensionnement

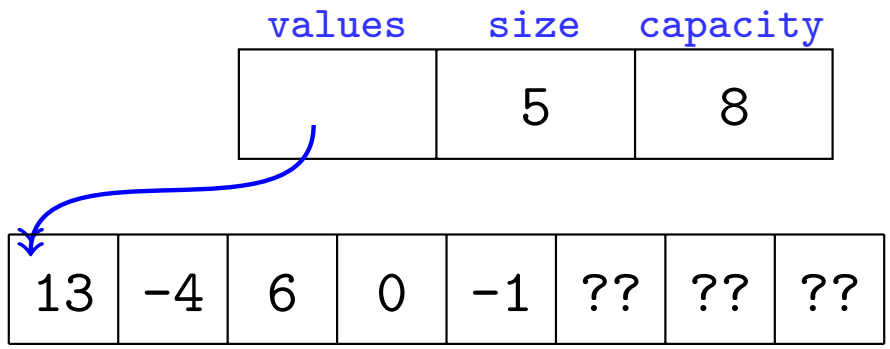
- **Automatiquement**, en doublant la taille
- Ou **manuellement**, par un appel de fonction
- Parfois, **contracté** automatiquement quand trop d'espace inoccupé

Interface

```
// Initialize an empty array
void array_initialize(array *a);

// Append an element to the end of an array
void array_append(array *a, int e);

// Insert an element in an array
void array_insert(array *a, unsigned int i, int element);

// Remove an element from an array at a given index
void array_remove(array *a, unsigned int i);

// Check if an array contains a given element
bool array_has_element(const array *a, int e);

// Return the element at a given index in an array
int array_get(const array *a, unsigned int i);

// Delete an array
void array_delete(array *a);
```

Représentation

```c
// A resizable array
typedef struct {
    int *values;            // The values of the array
    unsigned int size;      // The current size
    unsigned int capacity; // The capacity
} array;
```



Invariants:

- La taille réservée par `values` est `capacity * sizeof(int)`
- `size <= capacity`
- Seules les `size` premières valeurs sont « pertinentes »

Implémentation (1/2)

```
// Initialize an empty array
void array_initialize(array *a) {
    a->values = malloc(sizeof(int));
    a->size = 0;
    a->capacity = 1;
}

// Append an element to the end of an array
void array_append(array *a, int e) {
    array_resize_if_needed(a);
    a->values[a->size] = e;
    ++a->size;
}

// Insert an element in an array
void array_insert(array *a, unsigned int i, int e) {
    array_check_index(a, i);
    array_resize_if_needed(a);
    for (unsigned int j = a->size - 1; j > i; --j)
        a->values[j] = a->values[j - 1];
    a->values[i] = e;
    ++a->size;
}
```

Implémentation (2/2)

```
// Remove an element from an array at a given index
void array_remove(array *a, unsigned int i) {
    array_check_index(a, i);
    ++i;
    while (i < a->size) {
        a->values[i - 1] = a->values[i];
        ++i;
    }
    --a->size;
}

// Check if an array contains a given element
bool array_has_element(const array *a, int e) {
    unsigned int i = 0;
    while (i < a->size && array_unsafe_get(a, i) != e)
        ++i;
    return i < a->size;
}

// Return the element at a given index in an array
int array_get(const array *a, unsigned int i) {
    array_check_index(a, i);
    return array_unsafe_get(a, i);
}

// Delete an array
void array_delete(array *a) {
    free(a->values);
}
```

Fonctions supplémentaires

```c
// Resize an array if the capacity is reached
void array_resize_if_needed(array *a) {
    if (a->size == a->capacity) {
        a->capacity *= 2;
        a->values = realloc(a->values, a->capacity * sizeof(int));
    }
}

// Return the element at index i in an array (no check)
int array_unsafe_get(const array *a, unsigned int i) {
    return a->values[i];
}

// Print an array to stdout
void array_print(const array *a) {
    unsigned int i;
    printf("[");
    for (i = 0; i < a->size; ++i) {
        printf(" %d", a->values[i]);
    }
    printf(" ]");
}

// Check if an index is out of bound
void array_check_index(const array *a, unsigned int i) {
    if (i >= a->size) {
        fprintf(stderr, "Invalid index %d (size = %d)\n", i, a->size);
        exit(1);
    }
}
```

Utilisation

```
int main() {
    array a; array_initialize(&a);
    printf("Appending 3, 2, 5, 7, 8, 7: ");
    array_append(&a, 3); array_append(&a, 2); array_append(&a, 5);
    array_append(&a, 7); array_append(&a, 8); array_append(&a, 7);
    array_print(&a);
    printf("\nRemoving at position 2: "); array_remove(&a, 2); array_print(&a);
    printf("\nRemoving at position 4: "); array_remove(&a, 4); array_print(&a);
    printf("\nRemoving at position 2: "); array_remove(&a, 2); array_print(&a);
    printf("\nInserting 7 at position 1: ");
    array_insert(&a, 1, 7); array_print(&a); printf("\n");
    for (int e = 0; e <= 9; e += 2)
        printf("Has element %d ? %s\n", e,
               array_has_element(&a, e) ? "yes" : "no");
    array_delete(&a);
}
```

Résultat:

```
Inserting 3, 2, 5, 7, 8, 7: [ 3 2 5 7 8 7 ]
Removing at position 2: [ 3 2 7 8 7 ]
Removing at position 4: [ 3 2 7 8 ]
Removing at position 2: [ 3 2 8 ]
Inserting 7 at position 1: [ 3 7 2 8 ]
Has element 0 ? no
Has element 2 ? yes
Has element 4 ? no
Has element 6 ? no
Has element 8 ? yes
```

Tableaux multidimensionnels

Tableau multidimensionnel

- Généralisation d'un tableau à **plusieurs dimensions**
- Aussi appelé **matrice**
- Simplifie la manipulation des **données** homogènes
- En les organisant selon leurs **dimensions**

Version aplatie

- Tableau **unidimensionnel**
- **Plus** compact
- Mais on doit gérer l'**accès** (indexation)

Par indirection

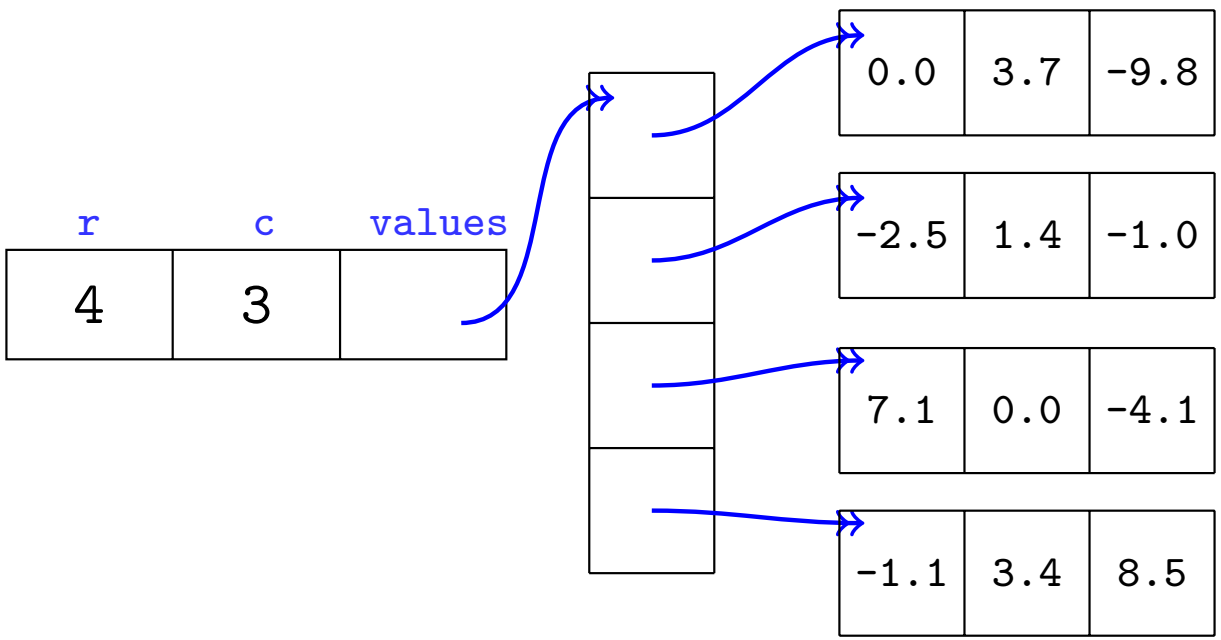
- Tableau de **pointeurs**
- **Moins** compact
- Mais plus facile de gérer l'indexation

Interface (partielle)

```c
// Initialize a matrix
void matrix_initialize(struct matrix *m,
                       unsigned int r,
                       unsigned int c,
                       bool random);

// Print a matrix to stdout
void matrix_print(const struct matrix *m);

// Add the matrix `m2` to the matrix `m1`
void matrix_add(struct matrix *m1, const struct matrix *m2);

// Delete the given matrix
void matrix_delete(struct matrix *m);
```

Représentation

```
struct matrix {
    unsigned int r;  // Number of rows
    unsigned int c;  // Number of columns
    double **values; // Values in matrix
};
```



Implémentation (1/2)

```
// Initialize a matrix with 0 or random values
void matrix_initialize(struct matrix *m,
                       unsigned int r,
                       unsigned int c,
                       bool random) {
    m->r = r;
    m->c = c;
    m->values = malloc(r * sizeof(double*));
    for (unsigned int i = 0; i < r; ++i) {
        m->values[i] = malloc(c * sizeof(double));
        for (unsigned int j = 0; j < c; ++j) {
            if (random) {
                m->values[i][j] = (float)rand() /
                                  (float)(RAND_MAX / 20.0) - 10.0;
            } else {
                m->values[i][j] = 0.0;
            }
        }
    }
}
```

Implémentation (2/2)

```c
// Print a matrix to stdout
void matrix_print(const struct matrix *m) {
    for (unsigned int i = 0; i < m->r; ++i) {
        printf("[ ");
        for (unsigned int j = 0; j < m->c; ++j) {
            printf("%6.2lf ", m->values[i][j]);
        }
        printf("]\n");
    }
}

// Add the second matrix to the first one
void matrix_add(struct matrix *m1, const struct matrix *m2) {
    if (m1->r != m2->r || m1->c != m2->c) {
        fprintf(stderr, "Error: matrices have different dimensions\n");
        exit(1);
    }
    for (unsigned int i = 0; i < m1->r; ++i)
        for (unsigned int j = 0; j < m1->c; ++j)
            m1->values[i][j] += m2->values[i][j];

}

// Delete a matrix
void matrix_delete(struct matrix *m) {
    for (unsigned int i = 0; i < m->r; ++i)
        free(m->values[i]);
    free(m->values);
}
```

Utilisation

```c
int main(void) {
    srand(time(NULL));
    struct matrix m1, m2;
    printf("Initializing m1:\n");
    matrix_initialize(&m1, 3, 5, true); matrix_print(&m1);
    printf("Initializing m2:\n");
    matrix_initialize(&m2, 3, 5, true); matrix_print(&m2);
    printf("Adding m2 to m1:\n");
    matrix_add(&m1, &m2); matrix_print(&m1);
    matrix_delete(&m1); matrix_delete(&m2);
    return 0;
}
```

Résultat (les valeurs peuvent varier):

```
Initializing m1:
[   0.01    8.16   -7.81   -5.44   -3.65 ]
[  -2.18   -1.61   -5.82   -7.56   -1.02 ]
[   4.41   -3.73   -7.61    0.10    8.28 ]
Initializing m2:
[  -3.21   -1.22   -4.39    8.25    2.22 ]
[  -1.98   -3.68    4.90   -6.46    6.50 ]
[   6.21   -3.26   -8.34   -1.79   -3.21 ]
Adding m2 to m1:
[  -3.20    6.94  -12.20    2.81   -1.43 ]
[  -4.16   -5.29   -0.93  -14.02    5.48 ]
[  10.62   -6.99  -15.95   -1.69    5.07 ]
```

Arbres binaires de recherche

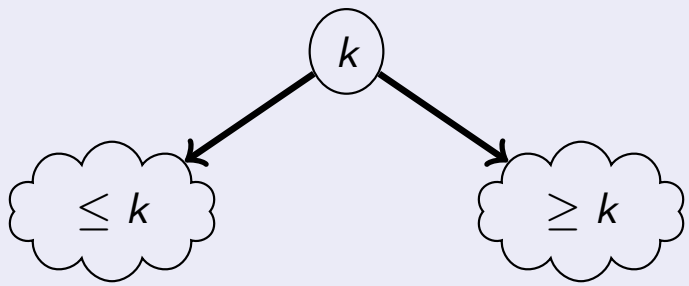
Arbres binaires de recherche

Arbre binaire

- Ensemble de noeuds organisés de façon **hiérarchique**
- Chaque noeud référence **deux** enfants
- Possiblement **vides**

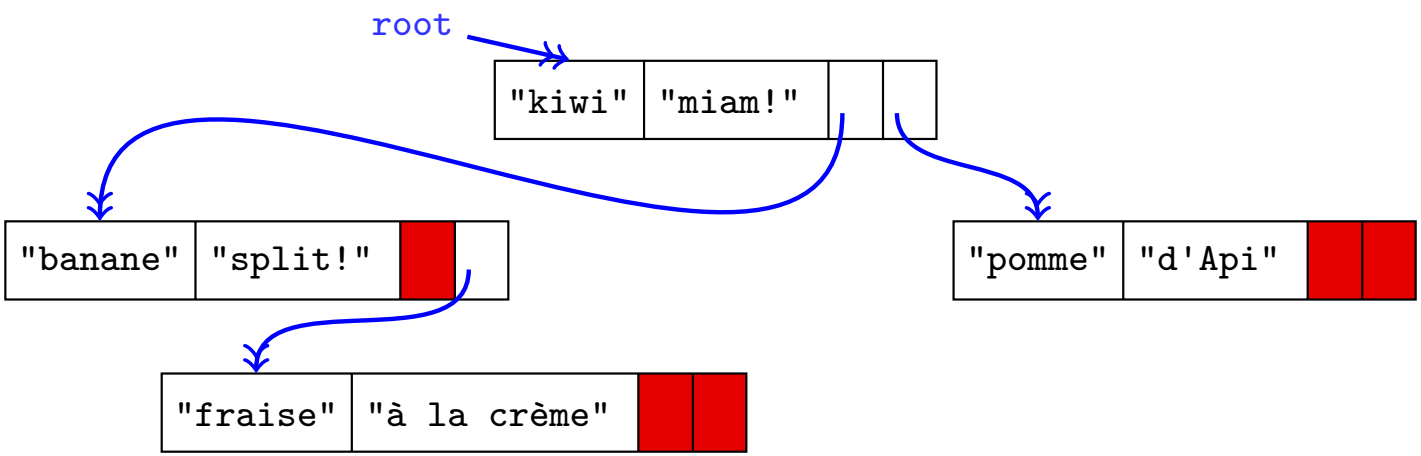
Arbre binaire de recherche (ABR)

- Chaque noeud est identifié par une **clé**
- Choisie dans un ensemble de clés **totalement ordonné**
- Et contient une **valeur**
- **Invariant**:



Représentation

```
// A tree node
struct tree_node {
    char *key;                  // Key
    char *value;                // Value
    struct tree_node *left;     // Left subtree
    struct tree_node *right; // Right subtree
};

// A tree map
typedef struct {
    struct tree_node *root; // Root of tree
} treemap;
```



Interface

```
// Initialize an empty tree map
void treemap_initialize(treemap *t);

// Return the value associated with the given key in a tree map
char *treemap_get(const treemap *t, const char *key);

// Set the value for the given key in a tree map
void treemap_set(treemap *t, const char *key, const char *value);

// Indicate if a key exists in a tree map
bool treemap_has_key(const treemap *t, const char *key);

// Print a tree map to stdout
void treemap_print(const treemap *t);

// Delete a tree map
void treemap_delete(treemap *t);
```

Récupérer un noeud

```
struct tree_node *treemap_get_node(const struct tree_node *node,
                                   const char *key) {
    if (node == NULL) {
        return NULL;
    } else {
        int cmp = strcmp(key, node->key);
        if (cmp == 0)
            return (struct tree_node*)node;
        else if (cmp < 0)
            return treemap_get_node(node->left, key);
        else
            return treemap_get_node(node->right, key);
    }
}
```

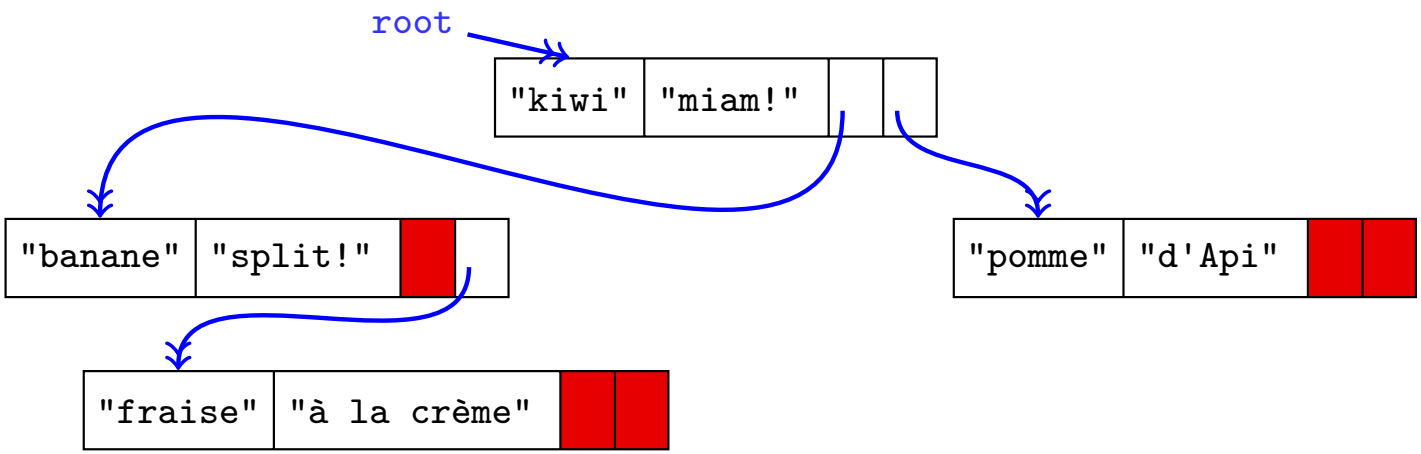
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```

Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, v
    } else {
        treemap_insert_node(&(*node)->right, key, valu);
    }
}
```

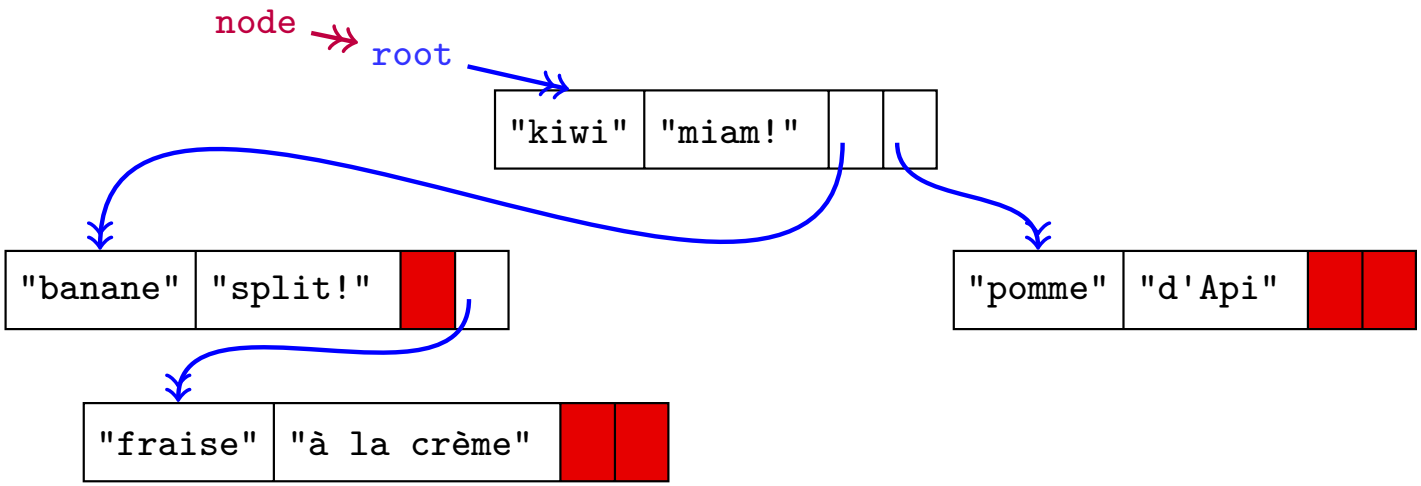
On souhaite insérer la clé melon



Insérer un noeud

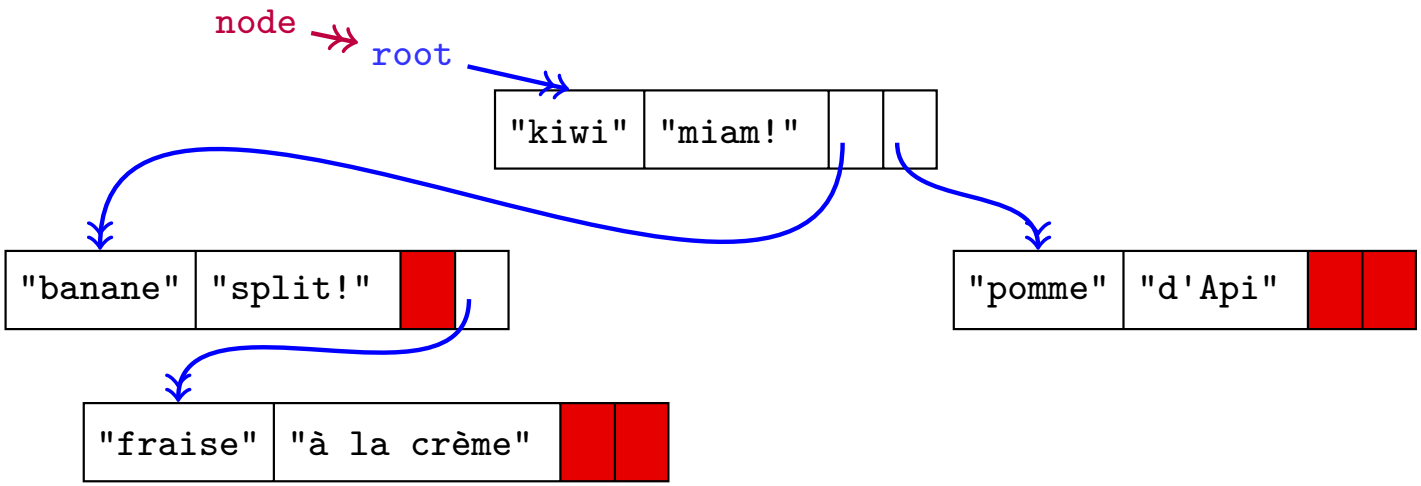
```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, v
    } else {
        treemap_insert_node(&(*node)->right, key, valu);
    }
}
```

On souhaite insérer la clé melon



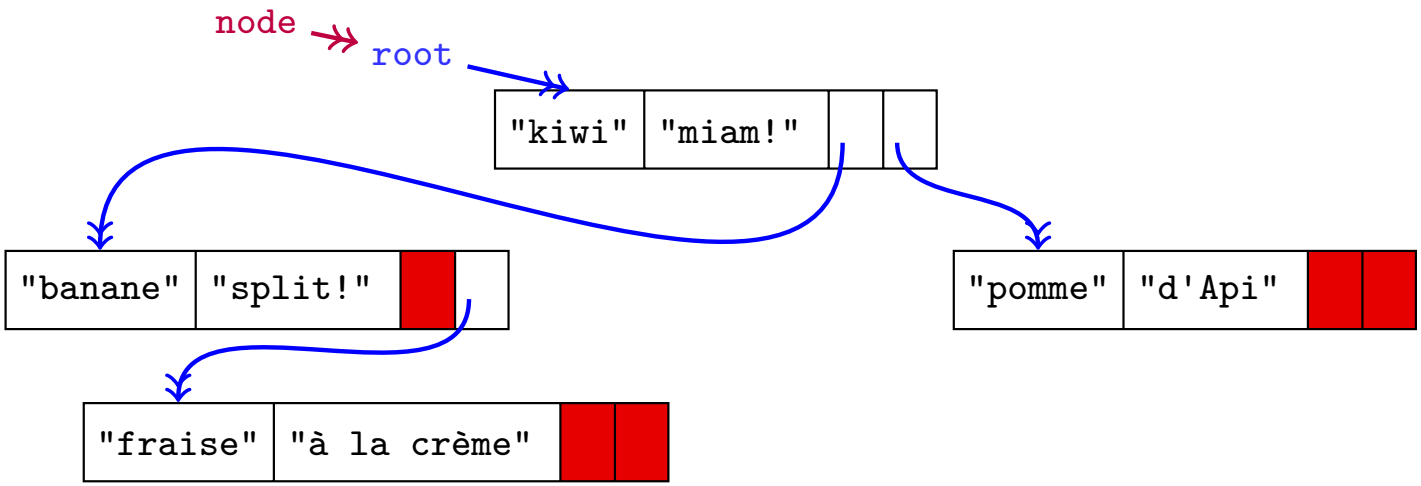
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



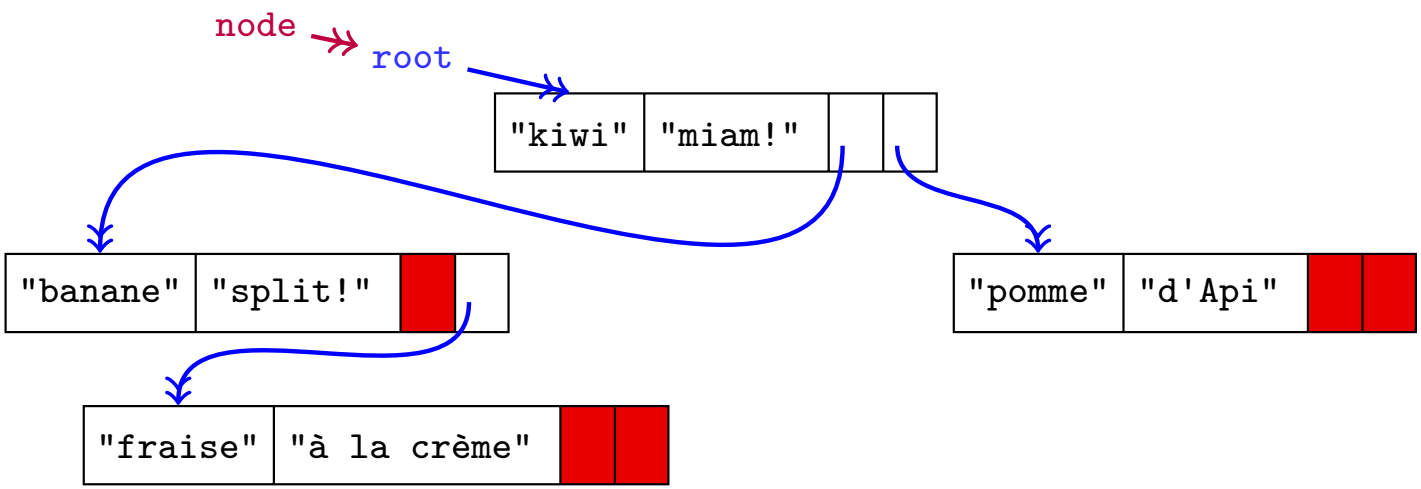
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



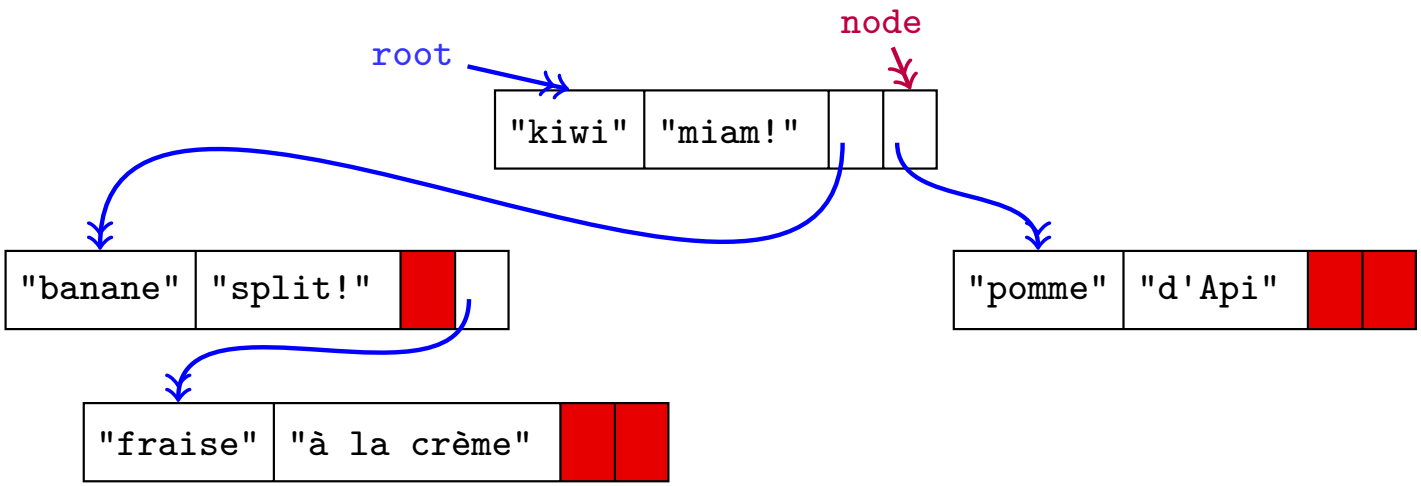
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



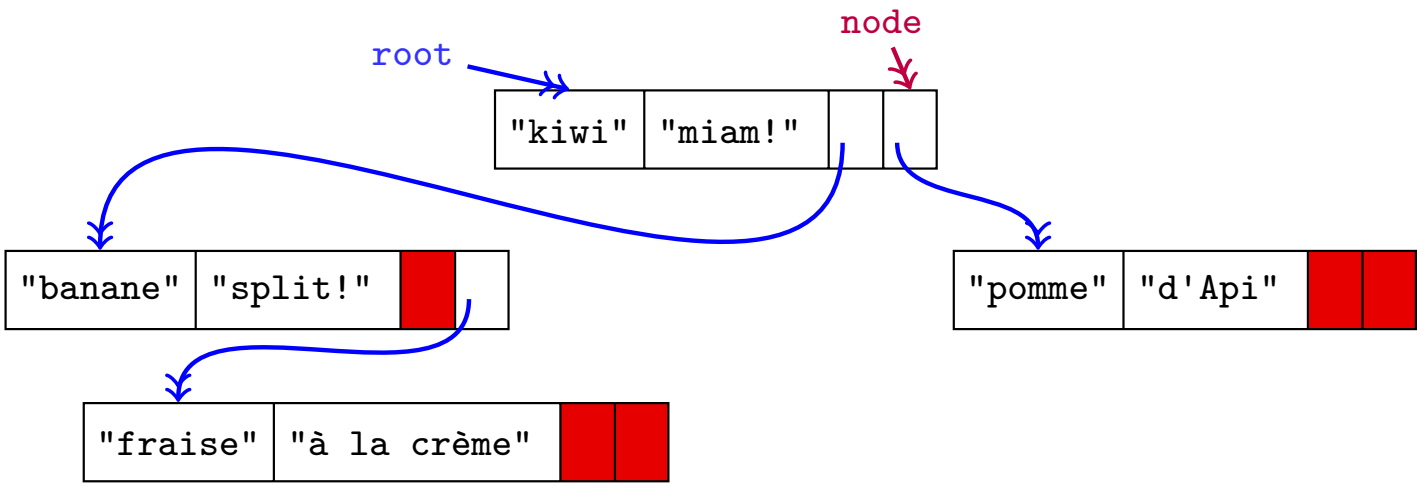
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



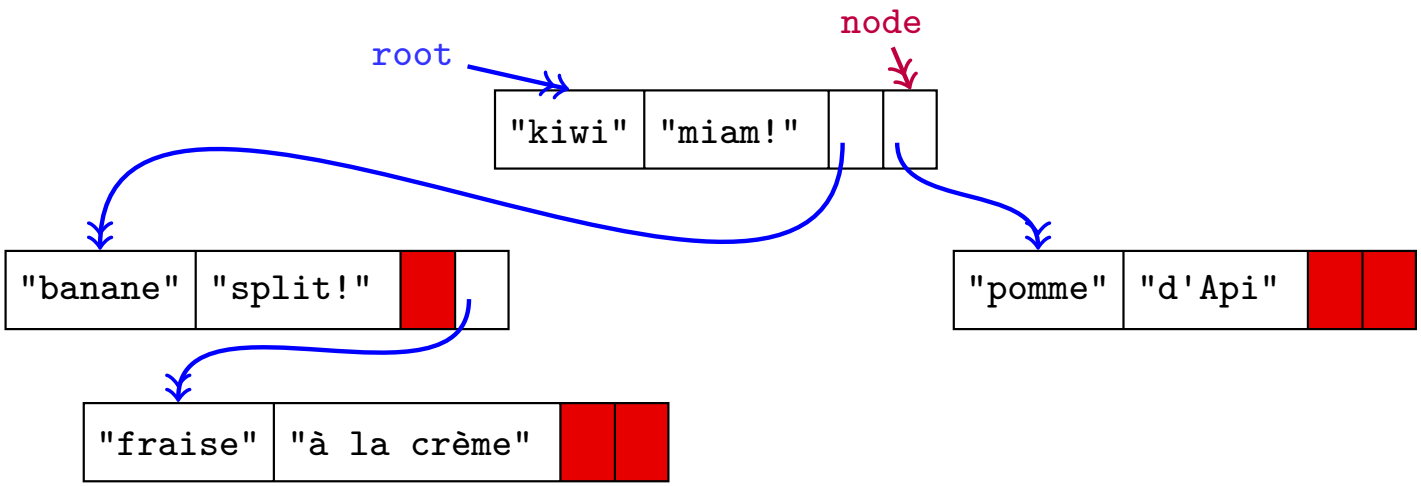
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



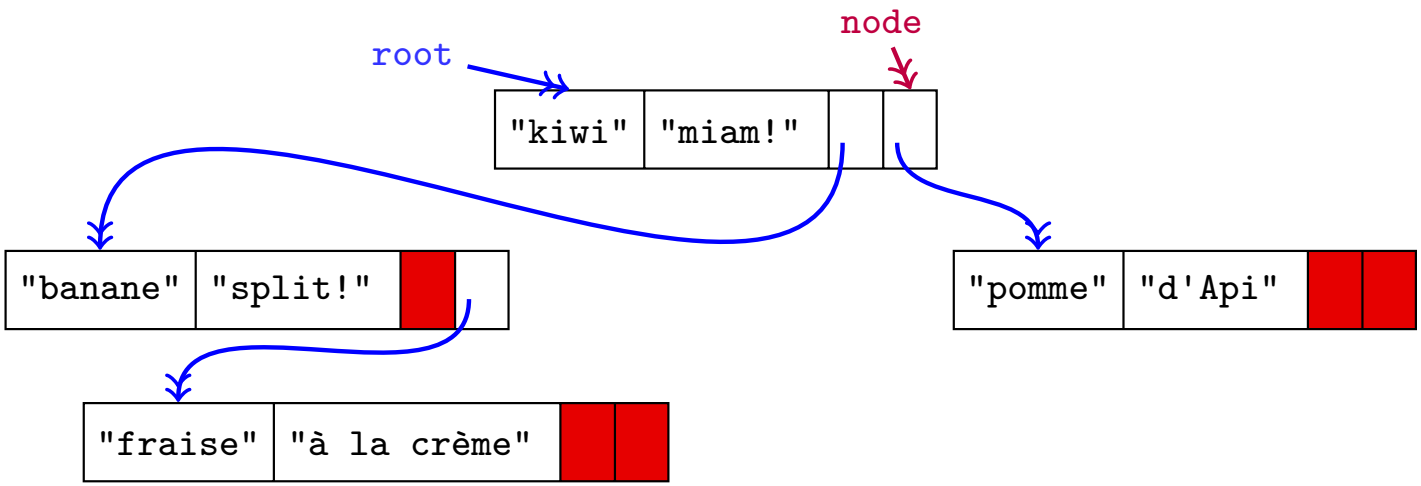
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



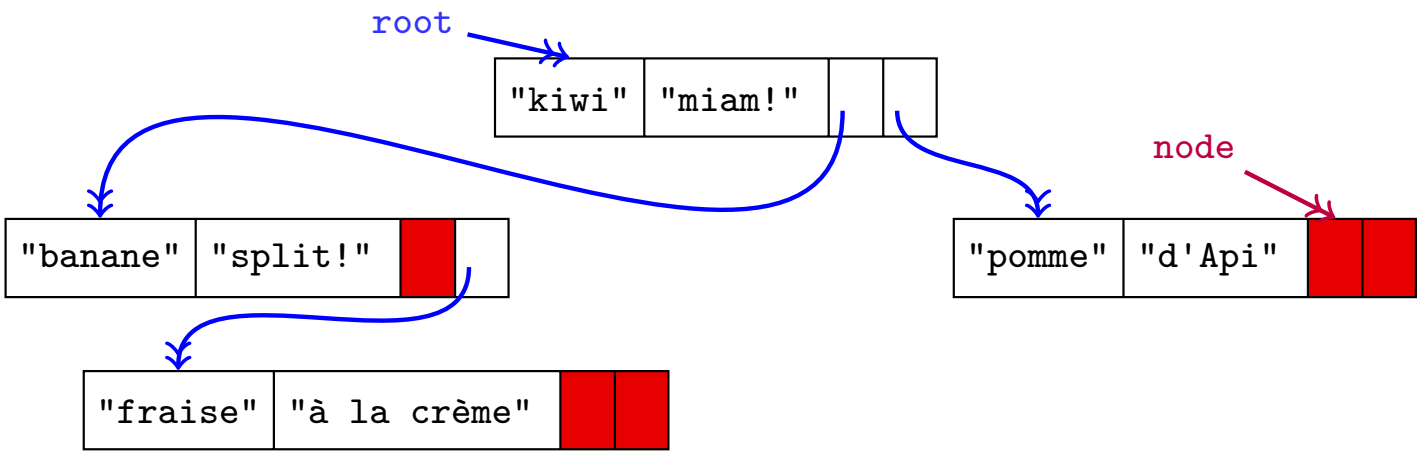
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



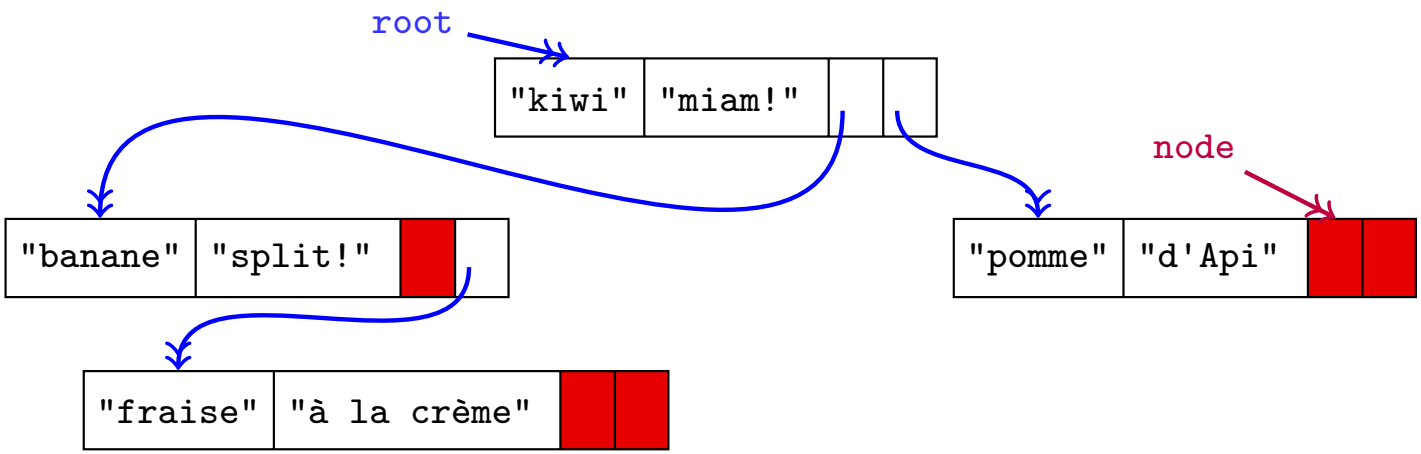
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



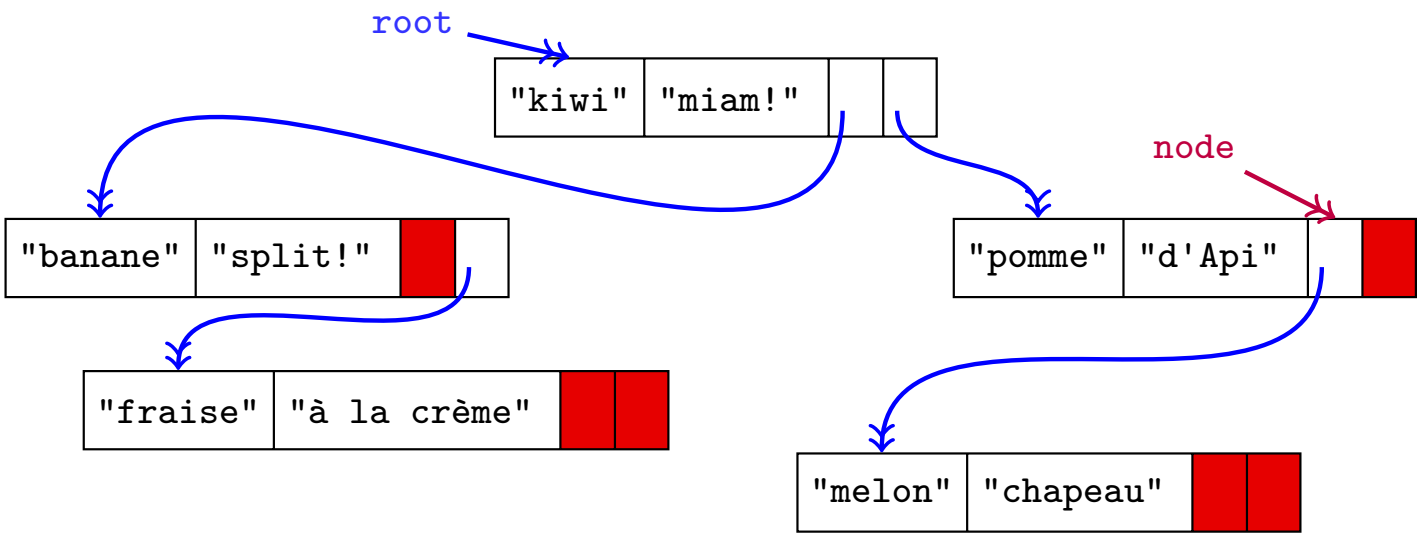
Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



Insérer un noeud

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```



Implémentation (1/2)

```
void treemap_initialize(treemap *t) {
    t->root = NULL;
}

char *treemap_get(const treemap *t, const char *key) {
    struct tree_node *node = treemap_get_node(t->root, key);
    return node == NULL ? NULL : node->value;
}

bool treemap_has_key(const treemap *t, const char *key) {
    return treemap_get_node(t->root, key) != NULL;
}

void treemap_set(treemap *t, const char *key, const char *value) {
    struct tree_node *node = treemap_get_node(t->root, key);
    if (node != NULL) {
        free(node->value);
        node->value = strdup(value);
    } else {
        treemap_insert_node(&(t->root), key, value);
    }
}
```

Implémentation (2/2)

```
void treemap_print(const treemap *t) {
    printf("TreeMap {\n");
    treemap_print_node(t->root);
    printf("}\n");
}

void treemap_print_node(const struct tree_node *node) {
    if (node != NULL) {
        treemap_print_node(node->left);
        printf("  %s: %s\n", node->key, node->value);
        treemap_print_node(node->right);
    }
}

void treemap_delete(treemap *t) {
    treemap_delete_node(t->root);
}

void treemap_delete_node(struct tree_node *node) {
    if (node != NULL) {
        treemap_delete_node(node->left);
        treemap_delete_node(node->right);
        free(node->key);
        free(node->value);
        free(node);
    }
}
```

Utilisation

```c
int main() {
    treemap t;
    treemap_initialize(&t);
    treemap_set(&t, "firstname", "Doina"); treemap_set(&t, "lastname", "Precup");
    treemap_set(&t, "city", "Montreal");   treemap_set(&t, "province", "Quebec");
    treemap_set(&t, "country", "Canada");  treemap_set(&t, "position", "DeepMind");
    printf("Printing the tree map\n"); treemap_print(&t);
    printf("Get \"firstname\": %s\n", treemap_get(&t, "firstname"));
    printf("Get \"province\": %s\n", treemap_get(&t, "province"));
    printf("Get \"position\": %s\n", treemap_get(&t, "position"));
    printf("Changing country to Romania\n"); treemap_set(&t, "country", "Romania");
    printf("Get \"country\": %s\n", treemap_get(&t, "country"));
    printf("Printing the tree map\n"); treemap_print(&t);
    treemap_delete(&t);
}
```

Résultat:

```
Printing the tree map
TreeMap {
  city: Montreal
  country: Canada
  firstname: Doina
  lastname: Precup
  position: DeepMind
  province: Quebec
}
Get "firstname": Doina
Get "province": Quebec
Get "position": DeepMind
```

```
Changing country to Romania
Get "country": Romania
Printing the tree map
TreeMap {
  city: Montreal
  country: Romania
  firstname: Doina
  lastname: Precup
  position: DeepMind
  province: Quebec
}
```

INF3135
Construction et maintenance de logiciels

Chapitre 6: Modularité

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Généralités

Modularité

Définition (extraite de Wikipedia)

« Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. »

Caractéristiques

- **Séparation**: les préoccupations sont divisées en composantes
- **Indépendance**: les dépendances entre modules sont minimales
- **Interchangeabilité**: facilité de remplacer une composante
- **Spécifique**: un module règle une préoccupation précise
- **Réutilisation**: un module est souvent réutilisable

Terminologie

Varie selon le langage

- **Montage** (*assembly*): spécifique à Microsoft
- **Module**: un seul fichier ou un ensemble de fichiers
- **Paquet** (*package*): ensemble de modules
- **Composante**: une partie d'un système complexe

Exemples

- **Java**: un paquet (*package*)
- **C**: une paire de fichiers `.h`/`.c` (ou juste `.c`)
- **C++**: une paire de fichiers `.hpp`/`.cpp` (ou juste `.cpp`)
- **Python**: module = fichier, paquet = ensemble de modules
- **Haskell**: un fichier

Contenu d'un module

Interface

- Ce qui est **fourni** et **requis**
- Généralement visible de façon **« publique »**
- Souvent présenté sous forme **déclarative**
- Documentation décrivant l'**utilisation**

Implémentation

- **Mise en oeuvre** de ce qui est déclaré dans l'interface
- Généralement **« privé »**
- Souvent à l'aide de programmation **structurée**
- Documentation décrivant le **développement**

Couplage et cohésion

Objectifs

- **Minimiser** le couplage
- **Maximiser** la cohésion

Couplage (inter-modules)

« In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. »

Cohésion (intra-module)

« In computer programming, cohesion refers to the degree to which the elements inside a module belong together. »

Macros

Rappel

Précompilation

- **Interprétées** par le préprocesseur **avant** la compilation
- Symbole # au **début** de la ligne
- On peut insérer des **espaces** entre # et la directive

Directives permises

- `#include`: **inclusion** d'un fichier externe
- `#define`: définition d'un **symbole** ou d'une **macro**
- `#undef`: **annulation** d'un symbole ou d'une macro
- `#ifdef`/`#ifndef`: **vérifie** si une macro est définie ou non définie
- `#if`/`#else`/`#elif`/`#endif`: structure **conditionnelle**
- `#error`: indique une **erreur fatale**
- `#pragma`: pour des traitements plus **spécifiques**

Les macros

Macro

- **Fragment** de code
- Auquel on donne un **nom**
- Définie à l'aide de la directive `#define`

Deux types

- **Macro-objet**: ressemble à un objet ou une donnée
- **Macro-fonction**: ressemble à une fonction

Utilité

- Déclaration de **constantes** numériques ou textuelles
- Favorise la **réutilisation**
- Facilite la **méta-programmation**

Exemple

```c
#include <stdio.h>

// Une macro définie par rapport à d'autres
#define AREA (WIDTH * HEIGHT)
// Même si elles sont définies après
#define WIDTH 100
#define HEIGHT 200
// On peut utiliser n'importe quelle expression
#define values {8, 3, 1, 4, 5};
// Une suite de plusieurs instructions
#define MULTIPLE_PRINTS printf("1-"); \
                        printf("2-"); \
                        printf("3")

int main(void) {
    printf("%d x %d = %d\n", WIDTH, HEIGHT, AREA);
    int t[] = values
    for (unsigned int i = 0; i < 5; ++i) printf("%d ", t[i]);
    MULTIPLE_PRINTS; printf("\n"); return 0;
}
```

Résultat:

```
100 x 200 = 20000
8 3 1 4 5 1-2-3
```

Mécanisme

- L'expression `#define symbole valeur`
- Remplace toutes les **occurrences** de `symbole` par `valeur`
- **Valeur**: reste de la ligne
- Valeur **multi-ligne**: à l'aide de \
- **Portée**: jusqu'à la fin du fichier
- Sauf si on trouve une commande `#undef`

```c
#include <stdio.h>

int main(void) {
#   define X 10
    printf("%d ", X);
#   undef X
#   define X 20
    printf("%d ", X);
    return 0;
}
```

Résultat:

```
10 20
```

Visualiser l'expansion des macros

- Possible de visualiser l'**expansion** des macros
- En passant l'**option** `-E` à GCC:

```
$ gcc -E macro.c | tail -n 6
int main(void) {
    printf("%d x %d = %d\n", 100, 200, (100 * 200));
    int t[] = {8, 3, 1, 4, 5};
    for (unsigned int i = 0; i < 5; ++i) printf("%d ", t[i]);
    printf("1-"); printf("2-"); printf("3"); printf("\n"); return 0;
}
```

Options utiles:

- `-E`: n'effectue que la précompilation
- `-C`: conserver les commentaires
- `-P`: supprimer les marqueurs de ligne (de la forme #)

Symboles prédéfinis

Fichier `predefined.c`:

```c
#include <stdio.h>

int main() {
    printf("Nom du fichier source courant: %s\n", __FILE__);
    printf("Numéro de la ligne courante: %d\n", __LINE__);
    printf("Date de compilation: %s\n", __DATE__);
    printf("Heure de compilation: %s\n", __TIME__);
    printf("Compilateur conforme à la norme ISO? %s\n",
           __STDC__ == 1 ? "oui" : "non");
    return 0;
}
```

Affiche:

```
Nom du fichier source courant: predefined.c
Numéro de la ligne courante: 5
Date de compilation: Jul 14 2020
Heure de compilation: 11:34:11
Compilateur conforme à la norme ISO? oui
```

Définition de symboles à la compilation

- Il est possible de définir des symboles à la **compilation**
- Grâce à l'option `-D` de GCC
- **Exemples**:

```
# Préciser la langue
gcc -DLANG=FR fichier.c

# Récupérer une information dépendante du système
# Rappel: pwd retourne le chemin absolu du répertoire courant
root_dir="$(pwd)"
gcc -DROOT_DIR="\"$root_dir\"" fichier.c

# Récupérer le noyau du système
# Rappel: uname retourne le nom du noyau
gcc -DKERNEL="\"$(uname)\"" fichier.c

# Désactiver les assertions
# Remplacer assert(...) par ((void)0)
# On va y revenir
gcc -DNDEBUG fichier.c
```

Directives conditionnelles

```
// Si un symbole existe
#ifdef symbol
[...]
#else
[...]
#endif

// Si un symbole n'existe pas
#ifndef symbol
[...]
#else
[...]
#endif

// Structure conditionnelle générale
// condition doit être vérifiable à la précompilation
// On peut utiliser `defined` pour vérifier si un symbole est
    défini
#if condition
[...]
#elif condition
[...]
#else
[...]
#endif
```

Exemples

- Empêcher les inclusions **multiples**:

```
#ifndef MAP_H
#define MAP_H
[...]
#endif
```

- Corriger l'inclusion selon le système

```
#if defined(LINUX)
#  include <SDL2/SDL.h>
#  include <SDL2/SDL_image.h>
#elif defined(OSX) || defined(WINDOWS)
#  include <SDL.h>
#  include <SDL_image.h>
#endif
```

- Fournir une **valeur** par défaut à un symbole

```
#ifndef ROOT_DIR
#define ROOT_DIR "."
#endif
```

Macro-fonctions

Macro-fonctions

- Ce sont des macros avec **paramètres**
- Le symbole doit être suivi immédiatement d'une parenthèse **ouvrante**
- Les paramètres sont séparés par des **virgules**
- Tout ce qui suit la parenthèse **fermante** fait partie de la définition
- **Exemple**:

```c
#include <stdio.h>

#define abs(X) ((X) >= 0 ? (X) : -(X))
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

int main(void) {
    int a = 2, b = -5;
    printf("abs(%d) = %d          ", b, abs(b));
    printf("abs(%d) = %d\n", a - 9, abs(a - 9));
    printf("min(%d, %d) = %d      ", a, b, min(a, b));
    printf("min(%d, %d) = %d\n", -8, 5 * b, min(-8, 5 * b));
    return 0;
}
```

Résultat:

```
abs(-5) = 5          abs(-7) = 7
min(2, -5) = -5      min(-8, -25) = -25
```

Exemple: trace d'un programme

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

#ifdef NOTRACE
#  define trace(...) /**/
#else
#  define trace printf
#endif

int main(void) {
    trace("Program starts\n");
    for (unsigned int i = 0; i < 5; ++i) {
        trace("i = %d ", i);
        printf("sqrt(%d) = %lf\n", i, sqrt(i));
    }
    trace("Program ends\n");
}
```

Résultat:

```
$ gcc trace.c -lm -o trace
$ ./trace
Program starts
i = 0 sqrt(0) = 0.000000
i = 1 sqrt(1) = 1.000000
i = 2 sqrt(2) = 1.414214
i = 3 sqrt(3) = 1.732051
i = 4 sqrt(4) = 2.000000
Program ends
```

```
$ gcc -DNOTRACE trace.c -lm -o trace
$ ./trace
sqrt(0) = 0.000000
sqrt(1) = 1.000000
sqrt(2) = 1.414214
sqrt(3) = 1.732051
sqrt(4) = 2.000000
```

Substitution de chaîne (*stringizing*)

- Un **argument** `ARG` peut être utilisé comme chaîne littérale
- À l'aide de l'expression `#ARG`
- Noter que des chaînes **littérales** peuvent être **concaténées**
- Simplement en les séparant par des **espaces**

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

#define warn_if(EXP) \
do { if (EXP) \
        printf("** " #EXP " ** "); } \
while (0)

bool divides_3(unsigned int i) { return i % 3 == 0; }

int main(void) {
    for (unsigned int i = 0; i < 8; ++i) {
        printf("%d ", i);
        warn_if(divides_3(i));
    }
    return 0;
}
```

Résultat:

```
0 ** divides_3(i) ** 1 2 3 ** divides_3(i) ** 4 5 6 ** divides_3(i) ** 7
```

Concaténation

- Un argument `ARG` peut être concaténé à un identifiant
- À l'aide de la notation `##`
- **Exemple**:

```
#define function(NAME)  draw_ ## NAME
```

- Alors `function(rectangle)` devient `draw_rectangle`
- Et `function(circle)` devient `draw_circle`

Fonction variadique

- Fonction ayant un nombre **arbitraire** d'arguments
- Commence avec k arguments **fixes**
- On doit en avoir **au moins un** $(k \geq 1)$
- Suivi de ...
- **Exemples**

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
```

Combien d'arguments?

- Utiliser un des arguments **fixes**
- Utiliser un **format** ou un **masque**
- Utiliser une valeur spéciale qui **marque** la fin

Implémentation

- À l'aide de la bibliothèque standard `stdarg.h`
- Fournit un **type** `va_list`
- Et un ensemble de **fonctions** pour le manipuler:

```
// Initialise ap sur le premier argument
void va_start(va_list ap, last);

// Récupère le prochain argument de ap
type va_arg(va_list ap, type);

// Clôt la lecture d'arguments de ap
void va_end(va_list ap);
```

- Les fonctions `va_start` et `va_end` doivent apparaître par **paires**

Exemple: `max` à plusieurs arguments

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

int max(int n, ...) {
    va_list list;
    va_start(list, n);
    if (n <= 0) return 0;
    int m = va_arg(list, int);
    for (unsigned int i = 1; i < n; ++i) {
        int m2 = va_arg(list, int);
        m = m2 > m ? m2 : m;
    }
    va_end(list);
    return m;
}

int main (void) {
    printf("%d ", max(3, 8, 2, -3));
    printf("%d\n", max(5, -4, -1, -8, 7, 6));
    return 0;
}
```

Résultat:

```
8 7
```

Exemple: types variables

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

void print_n_char(char c, int n) {
    for (unsigned int i = 0; i < n; ++i) printf("%c", c);
}

void print_n_chars(char c, int n, ...) {
    va_list list; va_start(list, n);
    print_n_char(c, n);
    while (true) {
        c = va_arg(list, int);
        if (c == '\0') break;
        int k = va_arg(list, int);
        print_n_char(c, k);
    }
}

int main (void) {
    print_n_chars('+', 1, '-', 10, '+', 1, '-', 10, '+', 1, '\0');
    printf("\n"); print_n_chars('a', 5, 'b', 8, 'c', 13, '\0');
    return 0;
}
```

Résultat:

```
+----------+----------+
aaaaabbbbbbbbccccccccccccc
```

Macros variadiques

- Macro-fonction ayant un nombre **arbitraire** d'arguments
- Commence avec k arguments **fixes**
- On peut avoir $k = 0$
- Suivi de ...
- **Exemples**

```
// Affichage formaté sur stderr
#define eprintf(...) fprintf(stderr, __VA_ARGS__)

// Débogage plus sophistiqué
#ifdef NDEBUG
#  define debug(format, ...) ((void)0)
#else
#  define debug(format, ...) \
      fprintf(stderr, "%s:%d:%s(): " FORMAT "\n", \
              __FILE__, __LINE__, __func__, __VA_ARGS__)
#endif
```

Exemple: Libtap

Extrait du fichier `tap.h`:

```
#define NO_PLAN          -1
#define SKIP_ALL         -2
#define ok(...)          ok_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define is(...)          is_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define isnt(...)        isnt_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define cmp_ok(...)      cmp_ok_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define cmp_mem(...)     cmp_mem_at_loc(__FILE__, __LINE__, __VA_ARGS__, NULL)
#define plan(...)        tap_plan(__VA_ARGS__, NULL)
#define done_testing()   return exit_status()
#define BAIL_OUT(...)    bail_out(0, "" __VA_ARGS__, NULL)
#define pass(...)        ok(1, "" __VA_ARGS__)
#define fail(...)        ok(0, "" __VA_ARGS__)

#define skip(test, ...) do {if (test) {tap_skip(__VA_ARGS__, NULL); break;}
#define end_skip         } while (0)

#define todo(...)        tap_todo(0, "" __VA_ARGS__, NULL)
#define end_todo         tap_end_todo()

#define dies_ok(...)     dies_ok_common(1, __VA_ARGS__)
#define lives_ok(...)    dies_ok_common(0, __VA_ARGS__)
```

Dangers associés aux macro-fonctions

```c
#include <stdio.h>

#define abs(X) ((X) >= 0 ? (X) : -(X))
#define min(X, Y) ((X) < (Y) ? (X) : (Y))

int main(void) {
    int a = 2, b = -5;
    printf("abs(%d) = %d          ", b, abs(b));
    printf("abs(%d) = %d\n", a - 9, abs(a - 9));
    printf("min(%d, %d) = %d      ", a, b, min(a, b));
    printf("min(%d, %d) = %d\n", -8, 5 * b, min(-8, 5 * b));
    return 0;
}
```

- **Priorité** des opérateurs
- $\rightarrow$ bien parenthéser
- Inefficacité lors d'évaluations **multiples**
- $\rightarrow$ éviter les expressions non évaluées
- **Duplication** des effets de bord
- $\rightarrow$ éviter les expressions avec effets de bord
- **Instruction multiples** mal formatées
- $\rightarrow$ protéger avec des accolades

Modules en C

Modules en C

Deux types de fichiers

- `fichier.h`: contient l'**interface** ou l'en-tête (*header*)
- `fichier.c`: contient l'implémentation de cette interface

Trois types de modules

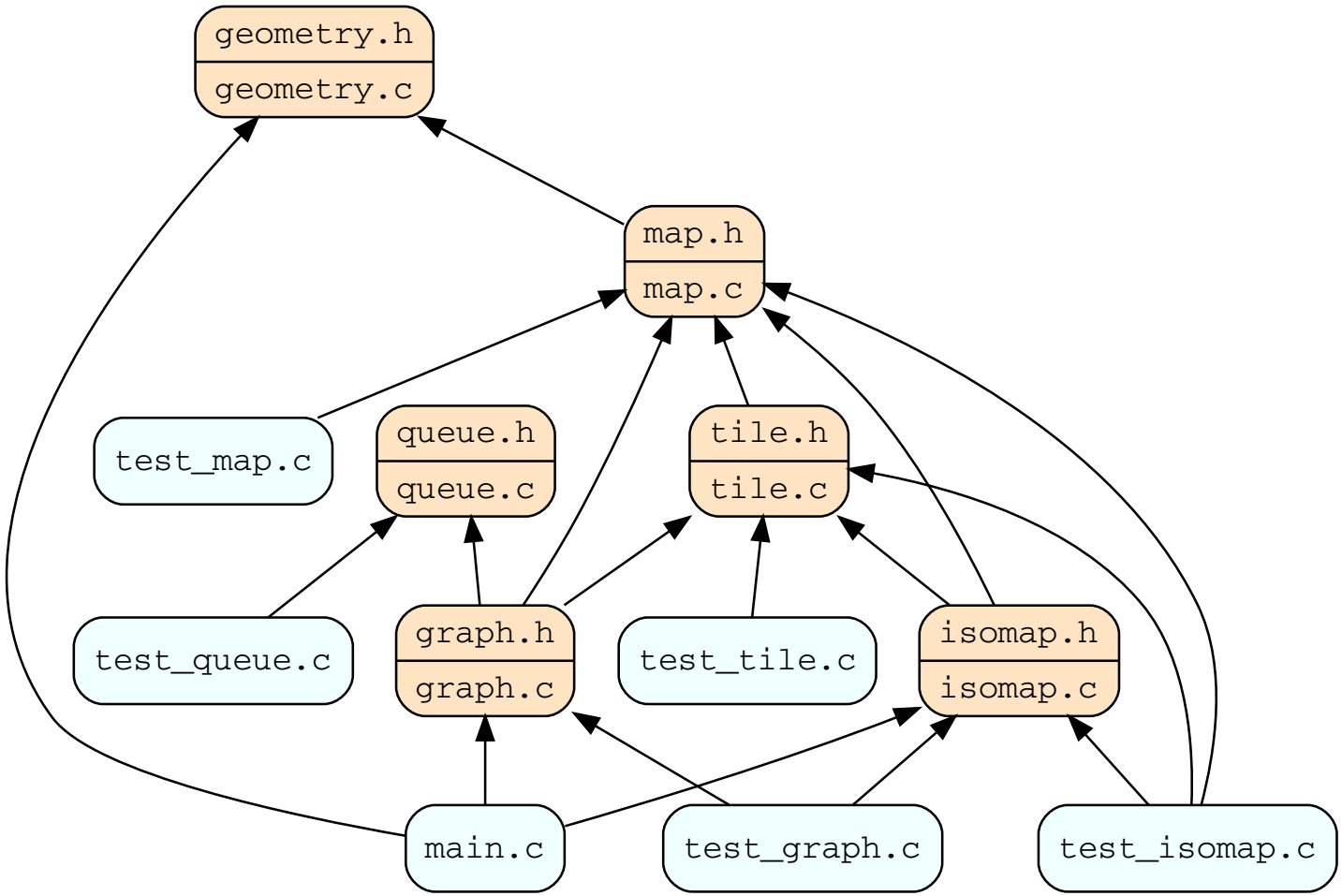
- **Régulier**: paire de fichier `.h` et `.c`
- **Déclaratifs**: seulement `.h`, avec déclarations et fonctions courtes
- **Point d'entrée**: seulement `.c` avec fonction `main`

Exemple

- `treemap.h`: interface d'un tableau associatif
- `treemap.c`: implémentation du tableau associatif
- `test_treemap.c`: teste le module `treemap`

Graphe de dépendances entre modules

- On trace une **flèche** du module `m1` vers le module `m2`
- si `m1` **inclut** `m2` (« inclure » = utiliser la directive `#include`)
- Idéalement, le graphe est **acyclique**



Contenu d'une interface (fichier `.h`) (1/2)

- Types: `enum`, `struct`, `union`, `typedef`, ...
- Inclusions **minimales**
- Constantes, variables externes
- Macros, macros-fonctions
- Fonctions usuelles

Qualité de l'interface

- Documenter l'**utilisation**
- Protéger contre les **inclusions multiples** (`#ifndef MODULE_H`)
- **Uniformiser** les signatures des fonctions
- En particulier l'**ordre** des arguments
- Placer le **type principal** comme premier argument
- **Nommer** les paramètres dans les signatures
- **Préfixer** (ou **suffixer**) les noms avec celui du **module**
- Uniformiser l'**indirection** (pointeur ou pas pointeur)

Contenu d'une interface (fichier `.h`) (2/2)

Organisation suggérée:

```
/**
 * Docstring d'en-tête
 */
#ifndef MODULE_H
#define MODULE_H

// Inclusions avec " "
// Inclusions avec < >

// Déclaration des macros et des macros-fonctions

// Déclaration des types

// Déclaration des fonctions « publiques »
// Avec leur docstring respective

#endif
```

- Remplacer `MODULE` par le nom du module
- On peut inverser les inclusions `" "` et `< >`

Contenu de l'implémentation (fichier `.c`) (1/2)

Déclarations

- Inclure le fichier `.h` associé à l'implémentation
- Toute autre constante, macro, macro-fonction auxiliaire
- Fonctions auxiliaires

Qualité de l'implémentation

- Documenter le **développement**
- Mêmes remarques générales que pour l'**interface**
- Ne pas répéter les **inclusions** déjà dans `.h`
- Ne pas répéter les *docstrings* des fonctions « publiques »
- Idéalement, **documenter** les fonctions « privées »
- **Encapsulation**: cacher le plus possible l'implémentation

Contenu de l'implémentation (fichier .c) (2/2)

Organisation suggérée:

```
#include "MODULE.h"
// Autres inclusions avec " "
// Inclusions avec < >

// Déclaration de macros et de macros-fonctions auxiliaires
// Déclaration de types auxiliaires

// Déclaration et implémentation des fonctions auxiliaires
// Idéalement avec leur docstring respective

// Implémentation des fonctions « publiques »
// Sans leur docstring
```

– Remplacer `MODULE` par le nom du module

Compilation séparée

Étape 1: compilation des fichiers sources

- Avec `-Wall`, `-Wextra`, `-g`, ...
- Lien vers fichiers d'en-tête: `-I` (on va y revenir)

```
$ gcc -c [...] geometry.c
$ gcc -c [...] map.c
$ gcc -c [...] queue.c
[...]
```

- **Étape 2**: édition des liens
- Avec **options** pertinentes: `-lm`, `-ltap`, ...
- Lien vers bibliothèques: `-L` (on va y revenir)

```
$ gcc -o main main.o geometry.o map.o queue.o [...]
$ gcc -o test_queue queue.o test_queue.o [...]
$ gcc -o test_map map.o test_map.o [...]
[...]
```

Bibliothèques

Bibliothèque

Définition (extraite de Wikipedia)

*« En informatique, une bibliothèque logicielle est une collection de **routines**, qui peuvent être déjà compilées et prêtes à être utilisées par des programmes. Les bibliothèques sont enregistrées dans des fichiers semblables, voire identiques aux fichiers de programmes, sous la forme d'une collection de fichiers de code objet rassemblés accompagnée d'un index permettant de **retrouver facilement** chaque routine. Le mot librairie est souvent utilisé à tort pour désigner une bibliothèque logicielle. Il s'agit d'un anglicisme fautif dû à un **faux-ami** (library). »*

En C

- Il suffit d'ajouter `#include <bibliotheque.h>`
- Puis, lors de la **compilation** et de l'**édition des liens**,
- on indique à GCC où trouver les **fichiers** nécessaires

Deux types de bibliothèques

Statique

- Extension: `.a` en Unix, `.lib` sous Windows
- La bibliothèque est **incluse** dans l'exécutable
- **Avantage:** réduit les dépendances
- **Inconvénient:** exécutables plus volumineux

Dynamique

- Extension: `.so` en Unix, `.dll` sous Windows
- La bibliothèque est **liée dynamiquement**
- **Avantage:** évite les **redondances**, exécutables moins volumineux
- **Inconvénient:** nécessite une installation, problèmes de version

Compilation et édition des liens

Rappel sur les **étapes** de compilation:

- **Compilation**: `.c` → `.o`
- **Édition des liens**: `.o` → exécutable

Comment GCC gère-t-il ces deux étapes?

- **Compilation**: trouver les **en-tête** (fichiers `.h`)
- **Édition des liens**: trouver les fichiers **binaires**
- **Plusieurs formats** possibles: `.o`, `.a`, `.so`, `.dll`, ...

Deux syntaxes possibles

- À quel **endroit** GCC cherche ces fichiers?
- `#include "module.h"`: cherche dans le répertoire courant
- `#include <module.h>`: cherche dans le système

Emplacement des fichiers d'en-tête

- À la **compilation**
- GCC cherche seulement les fichiers d'**en-tête** (`.h`)
- Sur une installation **typique Unix**:

```
/usr/local/include
libdir/gcc/target/version/include
/usr/target/include
/usr/include
```

- Si le fichier `.h` est **ailleurs**, il faut le **spécifier**
- À l'aide de l'option `-I` (pour *include*):

```
$ gcc -I<chemin> ...
```

Attention

- Éviter les chemins **absolus** en **dur**
- Sinon le code n'est pas **portable**

Emplacement des binaires

- À l'**édition des liens**
- GCC cherche les implémentations **binaires** (`.o`, `.so`, `.dll`, …)
- Il inspecte **plusieurs répertoires**
- Pour les **connaître** (sur Linux), utiliser `ldconfig -v`:

```
$ ldconfig -v 2>/dev/null | grep -v ^$'\t'
/usr/lib/x86_64-linux-gnu/libfakeroot:
    libfakeroot-0.so -> libfakeroot-tcp.so
/lib/i386-linux-gnu:
    libwrap.so.0 -> libwrap.so.0.7.6
    libnss_dns.so.2 -> libnss_dns-2.27.so
[...]
```

- Si la bibliothèque se trouve ailleurs, il faut le **spécifier**:

```
$ gcc -L<chemin> ...
```

Attention

Ici aussi, éviter les chemins **absolus** en **dur**

L'utilitaire `pkg-config`

- L'utilisation de **chemins absolus en dur** n'est pas acceptable
- Si on souhaite qu'une application soit **portable**
- **Solution**: utiliser le programme PKG-config
- Pour les **inclusions** (`-I`):

```
$ pkg-config --cflags bibliotheque
```

- Pour les **binaires** (`-L` et `-l`):

```
$ pkg-config --libs bibliotheque
```

- Remplacer `bibliotheque` par la bibliothèque correspondante: `cairo`, `tap`, `jansson`, etc.

Exemple: Vec3D (1/2)

- Concevons notre propre bibliothèque: `vec3d`
- Voir les fichiers `vec3d.h` et `vec3d.c`
- Tout d'abord, on **compile** le fichier `vec3d.c` en objet `vec3d.o`:

```
$ gcc -o vec3d.o -c vec3d.c
```

- Ensuite, on crée la bibliothèque **statique** avec `ar`:

```
$ ar -cvq libvec3d.a vec3d.o
```

- On peut ensuite l'inclure via l'instruction en autant que l'**en-tête** et l'**implémentation** soient disponibles

```
#include <vec3d.h>
```

Exemple: Vec3D (2/2)

- Par exemple, supposons que les fichiers `vec3d.h` et `libvec3d.a` se trouvent respectivement dans les répertoires

```
/home/blondin_al/clib/include
/home/blondin_al/clib/lib
```

- Alors il suffit de compiler avec la commande

```
$ gcc -I/home/blondin_al/clib/include \
>     -c test_vec3d.c
```

- Puis de compléter l'**édition des liens** avec

```
$ gcc -L/home/blondin_al/clib/lib -o \
>     test_vec3d test_vec3d.o -lvec3d
```

Makefiles

Compilation de modules

- Souvent, un projet est divisé en plusieurs **modules**
- Il devient pénible de tout compiler **manuellement**
- En particulier en raison des **options** multiples
- **Compilation**: `-Wall`, `-Wextra`, `-g`, `-I`, ...
- **Édition des liens**: `-l`, `-L`, ...
- **Solution**: enrichir le contenu du `Makefile`

GNU Make

- Offre plusieurs **fonctions** (`wildcard`, `shell`, ...)
- Et **variables** spéciales (`MAKE`, `MAKEFILE_LIST`, ...)

Quelques mécanismes et fonctions

- Les règles à motifs (*pattern rule*)
- `wildcard`: lister des fichiers (*glob*)
- `patsubst`: substituer des motifs
- `shell`: invoquer une commande shell
- `filter-out`: retire un motif d'une liste de mots
- `realpath`: résoudre un chemin (simplifier et liens symboliques)
- `dir`: semblable à la commande `dirname`
- `abspath`: chemin absolu d'un fichier
- `lastword`: récupère le dernier mot d'une liste de mots
- ...

Voir la documentation officielle pour plus d'informations

Règles à motifs

- Permet de déclarer des règles **générales**
- Le caractère % est utilisé pour indiquer la **substitution**
- $<: première dépendance
- $@: cible

```
%.o: %.c
    gcc -c $(CFLAGS) $< -o $@
```

- On peut **restreindre** les cibles visées
- À l'aide d'une règle **statique**:

```
objects = geometry.o graph.o isomap.o map.o queue.o tile.o

$(objects): %.o: %.c %.h
    gcc -c $(CFLAGS) $< -o $@
```

Fonctions utiles (1/2)

- La fonction `wildcard`:

```
# Tous les fichiers avec extension .c
$(wildcard *.c)
# Tous les fichiers commençant avec test et finissant par .c
$(wildcard test*.c)
```

- La fonction `patsubst`:

```
# Fichiers objets souhaités
$(patsubst %.o, %.c, $(wildcard *.c))
# Exécutables des tests souhaités
$(patsubst %, %.c, $(wildcard test*.c))
```

- La fonction `shell`:

```
# Options de GCC pour la compilation
CFLAGS = "-std=c11 -Wall -Wextra $(shell pkg-config --cflags cairo)
    "
# Options de GCC pour l'édition des liens
LFLAGS = "$(shell pkg-config --libs cairo)"
```

Fonctions utiles (2/2)

- La fonction `filter-out`:

```
# Pour retirer les tests de la liste
test_c_files = $(wildcard test*.c)
c_files = $(filter-out $(test_c_files), $(wildcard *.c))
objects=$(patsubst %.o, %.c, $(c_files))
```

- Les fonctions `realpath`, `dir` et `abspath`:

```
# Récupérer le chemin absolu du répertoire parent d'un Makefile
# $(lastword $(MAKEFILE_LIST)) récupère le nom du Makefile courant
# Ensuite on calcule avec $(abspath ...) le chemin absolu
# Puis $(dir ...) permet de récupérer le répertoire parent
# Et finalement on prend le chemin simplifié avec $(realpath ...)
current_make = $(lastword $(MAKEFILE_LIST))
root_dir := $(realpath $(dir $(abspath $(current_make)))/..)
```

Quelques exemples

Les bibliothèques `unistd.h` et `getopts.h`

- Facilite le **traitement** des arguments récupérés par la fonction `main`
- Autrement dit, simplifie le traitement de `argc` et `argv`
- Deux types d'options: **courtes** et **longues**
- **Courtes**: un tiret, suivi d'une lettre:

```
$ ls -als
$ gcc -o tp1 tp1.c
```

- **Longues:** deux tirets, suivis d'un mot pouvant contenir des tirets:

```
$ valgrind --leak-check=yes ./prog
$ ./isomap --help
```

- Lorsque possible, préférer `getopts.h` à `unistd.h`
- Car gère **simultanément** les options courtes et longues

La bibliothèque Jansson (1/2)

- Permet de manipuler le format JSON
- Site officiel: https://digip.org/jansson/
- Dépôt Github: https://github.com/akheron/jansson
- Licence: MIT

```
{
    "firstname": "Petri",
    "lastname": "Lehtinen",
    "num-followers": 295,
    "projects": [
        "jansson",
        "optics-ts",
        "sala"
    ]
}
```

La bibliothèque Jansson (2/2)

```c
#include <stdio.h>
#include <string.h>
#include <jansson.h>

int main(void) {
    json_t *root = json_object();
    json_t *json_arr = json_array();
    json_object_set_new(root, "firstname", json_string("Petri"));
    json_object_set_new(root, "lastname", json_string("Lehtinen"));
    json_object_set_new(root, "num-followers", json_integer(295));
    json_object_set_new(root, "projects", json_arr);
    json_array_append(json_arr, json_string("jansson"));
    json_array_append(json_arr, json_string("optics-ts"));
    json_array_append(json_arr, json_string("sala"));
    char *s = json_dumps(root, 0);
    puts(s);
    json_decref(root);
    return 0;
}
```

Résultat:

```
$ gcc jansson.c -o jansson -ljansson && ./jansson
{"firstname": "Petri", "lastname": "Lehtinen", "num-followers": 295, "
     projects": ["jansson", "optics-ts", "sala"]}
```

La bibliothèque Cairo (1/2)

- Permet de dessiner des images **vectorielles**
- Supporte différents **formats**: PNG, PDF, SVG, etc.
- Site officiel: https://www.cairographics.org/
- Compilation/édition des liens: plus facile avec PKG-config

```
exec = hello
CFLAGS = $(shell pkg-config --cflags cairo)
LFLAGS = $(shell pkg-config --libs cairo)

$(exec): $(exec).o
	gcc $< -o $@ $(LFLAGS)

$(exec).o: $(exec).c
	gcc -o $@ $(CFLAGS) -c $<

.PHONY: clean

clean:
	rm -f *.o $(exec)
```

La bibliothèque Cairo (2/2)

```c
#include <cairo.h>

int main (int argc, char *argv[]) {
    cairo_surface_t *surface
        = cairo_image_surface_create(CAIRO_FORMAT_ARGB32,
                                     240, 80);
    cairo_t *cr = cairo_create (surface);

    cairo_select_font_face(cr, "serif",
        CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size (cr, 32.0);
    cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
    cairo_move_to (cr, 10.0, 50.0);
    cairo_show_text (cr, "Hello, world");

    cairo_destroy (cr);
    cairo_surface_write_to_png (surface, "hello.png");
    cairo_surface_destroy (surface);
    return 0;
}
```

La bibliothèque SDL

- Site officiel: `https://www.libsdl.org/`
- Permet de concevoir des applications **graphiques**
- À la **base** de plusieurs autres bibliothèques: Pygame, Kivy, ...
- Versions majeures: SDL**1.2** et SDL**2.0**
- Interaction de bas niveau avec les périphériques **graphiques** et **audio**
- Interface entre autres avec OpenGL
- Supporte seulement les formats BMP et WAV par défaut
- Voir l'application Maze

Bibliothèques compagnonnes

- `SDL_Image`: supporte le format PNG
- `SDL_mixer`: supporte le format MP3
- `SDL_ttf`: supporte le rendu de fontes (polices de caractères)

INF3135
Construction et maintenance de logiciels

Chapitre 8: Tests

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Tests

Pourquoi tester?

- Détecter des **bogues**
- Et éventuellement les **corriger**
- Avoir une plus grande **confiance** en notre programme
- S'assurer de ne pas introduire de **régression**

Idéal

- Prouver que notre programme est **sans bogue**
- **Impossible** dans la majorité des cas
- Sauf pour des **petits programmes**
- Ou en utilisant un outil de **vérification formelle**
- **Champ d'étude**: analyse de programme

Problème de l'arrêt (*halting problem*)

Problème

- Peut-on écrire un programme `G` qui
- Étant donné un **programme** `P`
- Décide si `P` **termine toujours**
- Peu importe les valeurs en **entrée**?

Réponse

- **1936**: Turing a prouvé qu'un tel programme `G` ne peut pas exister
- Plus formellement, le problème est **indécidable** (Gödel)

Conséquence

Impossible de **prouver** qu'un programme arbitraire est **sans bogue**

Exemple: collatz

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

void print_collatz_sequence(unsigned int n) {
    while (true) {
        printf("%d ", n);
        if (n == 1)
            break;
        else if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
    }
}

int main(int argc, char *argv[]) {
    print_collatz_sequence(atoi(argv[1]));
    return 0;
}
```

Résultat:

```
$ gcc collatz.c -o collatz
$ ./collatz 85
85 256 128 64 32 16 8 4 2 1
```

Différents types de tests (1/3)

Compilation (*build*)

- **Compilation** correcte
- Édition des **liens**
- Avec **bibliothèques** tierces
- Ou autres **dépendances**

Intégration

- Interaction correcte **entre** les modules
- Autant **compilation**
- Que tests **unitaires**

Différents types de tests (2/3)

Unitaires

- Teste un aspect **spécifique**
- De façon **individuelle**

Fonctionnels

- **Comportement** du programme
- Est en adéquation avec ce qui était **demandé**
- Respecte le **cahier des charges**

Non-régression

- Pas de perte de **fonctionnement**
- Ou de perte importante de **performance**

Différents types de tests (3/3)

Configuration

- Fonctionne dans des **environnements** variés
- **Appareils** (ordinateur, mobile, console, ...)
- **Distribution** (Linux, MacOS, Windows, ...)
- **Architecture** (32 bits, 64 bits, autre processeur, ...)

Performance

- **Rapidité** du programme
- Utilisation de **mémoire**

Installation

L'application s'installe correctement

Propriétés d'un bon test

- **Juste**: il teste bien ce qu'il faut
- **Robuste**: il ne plante pas
- **Pur**: il est sans effet de bord
- **Reproductible**: il a le même comportement peu importe l'environnement
- **Pertinent**: il augmente notre confiance
- **Non redondant**: il teste quelque chose de distinct d'un autre test
- **Efficace**: il prend un temps raisonnable
- **Automatisable**: il ne nécessite pas d'intervention humaine

Automatisation

Tests automatisés

- Vérification **textuelle**, notamment à l'aide de *regex*
- Utilisation des **canaux standards** (`stdin`, `stdout`, `stderr`)
- Ou accès direct au **contenu** (tests internes)
- Construction de **scénarios**

Tests manuels

- Quand automatisation **pas possible**
- Vérification **humaine**
- Évaluation **visuelle**, **sonore**, ...
- Suite d'**événements**, parfois **asynchrones**
- Prennent du **temps** et souvent **coûteux**
- Typique des applications **graphiques**
- Tests d'**interface d'utilisation**

Tests externes

Tests externes

- Aussi appelés tests en **boîte noire**
- Basés sur les **spécifications fonctionnelles**
- Autrement dit, on se concentre sur l'**interface**
- Et sur la **documentation** du module ou du programme
- En faisant abstraction de l'**implémentation**

Deux types de tests externes

- Tester l'**interface** d'un module (bibliothèque)
- Tester le **programme** lui-même (shell)

Quoi et comment tester?

- Tester les cas **typique**
- Puis des cas limites du **domaine**
- Regrouper les tests **équivalents**
- Pour éviter la **redondance**

Valeurs limites

Valeurs numériques

Valeurs négatives, 0, valeurs positives, débordements

Types énumératifs

Première et dernière valeurs

Caractère

Caractères spéciaux, affichables, encodage, ...

Collections (tableaux, listes, chaînes de caractères, ...)

Vide, singleton, tailles maximales, ...

Exemple: `int factorielle(int n)`

Un bon **cadre** de tests: $-5, 0, 1, 8, 12, 13, 28$

Exemple: le module set

Interface (fichier `set.h`):

```
struct set {                    // A set of integers
    int *elements;              // Its elements
    unsigned int size;          // Its cardinality
    unsigned int capacity; // Its capacity
};

// Create an empty set
struct set *set_create(void);
// Delete a set
void set_delete(struct set *set);
// Check if a set is empty
bool set_is_empty(const struct set *set);
// Check if a set contains an element
bool set_contains(const struct set *set, int element);
// Add an element to a set
void set_add(struct set *set, int element);
// Print a set to stdout
void set_print(const struct set *set);
```

Question

Quoi tester?

Tests possibles (1/2)

`set_create`:

- Vérifier que le résultat est bien vide
- Ou que sa cardinalité est `0`

`set_delete`:

- Pas vraiment testable

`set_is_empty`:

- Cas d'ensemble vide
- Cas d'ensemble non vide

`set_contains`:

- Cas où l'élément appartient à l'ensemble
- Cas où l'élément n'appartient pas à l'ensemble

Tests possibles (2/2)

`set_add`:

- Ajout d'un élément dans un ensemble vide
- Ajout d'un élément absent de l'ensemble
- Ajout d'un élément déjà dans l'ensemble

`set_print`:

- Pas vraiment testable à l'interne
- Mais pourrait être testé à l'externe (test shell)

Tests shell

- De nombreuses **commandes shell** permettent de tester
- Avec la **sémantique** habituelle (0: succès, $\neq$0: erreur)

Exemples:

```
# Vérifie si README.md contient un code permanent
#    -q: mode silencieux
#    -E: expression étendue
$ grep -qE "[A-Z]{4}[0-9]{8}" README.md

# Vérifie si bidon.c existe dans un sous-dossier de /tmp
$ find /tmp -name bidon.c | grep -q .

# Vérifie si fichier1 et fichier 2 sont presque identiques
#    -i: ignorer la casse
#    -w: ignorer les espaces
$ diff -iw fichier1 fichier2

# Vérifie si une commande engendre une fuite mémoire
# Valgrind retourne 0 si aucune fuite, 1 sinon
$ valgrind --leak-check=yes --error-exitcode=1 ./prog; echo $?
```

La commande `test`

Vérifier le type des fichiers et compare des valeurs:

```
test EXPRESSION [OPTION]
```

- Si l'expression est **vraie** alors la commande retourne 0
- Sinon elle retourne 1

Exemples:

```
# Vérifie si 1 < 2 (comparaison numérique)
$ test 1 -lt 2; echo $?
0
# Vérifie si linux = linux (en tant que chaînes)
$ test $(echo "linux") = "linux"; echo $?
0
# Vérifie s'il y a un Makefile dans le répertoire courant
$ test -f Makefile; echo $?
0
# Vérifie s'il y a un répertoire code dans le répertoire courant
$ test -d code; echo $?
0
```

Tests sur chaînes de caractères

```
test CHAINE1 OPERATEUR CHAINE2
```

Exemples:

```
# Vérifie si deux chaînes sont égales
$ test "linux" = "Linux"; echo $?
1
# Vérifie si deux chaînes sont différentes
$ test "linux" != "Linux"; echo $?
0
# Vérifie si une chaîne est vide
$ test -z ""; echo $?
0
$ test -z "linux"; echo $?
1
# Vérifie si une chaîne est non vide
$ test -n ""; echo $?
1
$ test -n "linux"; echo $?
0
```

Tests sur les valeurs numériques

```
test VALEUR1 OPERATEUR VALEUR2
```

Exemples:

```
# Vérifie si deux valeurs sont égales
$ test 1 -eq 1
# Vérifie si deux valeurs sont différentes
$ test 1 -ne 2
# Vérifie une inégalité
$ test 1 -lt 2
$ test 2 -le 2
$ test 2 -gt 1
$ test 1 -ge 1
# Attention à différencier `=` de `-eq`
$ test "01" = 1; echo $?
1
$ test "01" -eq 1; echo $?
0
```

Tests sur les fichiers

```
test OPTION CHEMIN
```

Exemples:

```
# Est-ce que le chemin existe?
$ test -e /usr/local/bin; echo $?
0
# Est-ce que le chemin est un fichier?
$ test -f /usr/local/bin/bats; echo $?
0
# Est-ce que le chemin est un répertoire?
$ test -f /usr/local/bin; echo $?
0
# Est-ce que le chemin est un fichier non vide?
$ touch nouveau
$ test -s nouveau; echo $?
1
# Est-ce que le chemin est accessible en lecture?
$ test -r chemin
# Est-ce que le chemin est accessible en écriture?
$ test -w chemin
# Est-ce que le chemin est exécutable?
$ test -x chemin
```

Opérateurs logiques

```
test EXPRESSION1 OPERATEUR EXPRESSION2
```

Exemples:

```
# ET logique
$ test expr1 -a expr2
# OU logique
$ test expr1 -o expr2
# NON logique
$ test ! expr
# Retourne vrai si chemin est un fichier vide
test -f chemin -a ! -s chemin
```

Syntaxe allégée

La syntaxe

```
test EXPRESSION
```

est équivalente à

```
[ EXPRESSION ]
```

Exemples:

```
$ [ -f Makefile ]; echo $?
1
$ [ -d bin ]; echo $?
0
```

- Les **espaces** après [et avant] sont importants
- Le caractère] est optionnel (mais plus joli)

Tests Bash

Syntaxe avec **doubles crochets**:

```
[[ EXPRESSION ]]
```

Opérateurs possibles:

- && et ||: connecteur logiques ET et OU
- (et): pour parenthéser
- < et >: comparaison lexicographique de chaînes
- ==: la 2e opérande est un motif de type *glob*
- =~: la 2e opérande est une expression régulière étendue

```
# La première expression correspond avec b = ? et * = njour
# La deuxième aussi avec onjou
#    [un]       alternative entre u et n
#    j?         j optionnel
#    (...){2}   motif répété exactement deux fois
$ [[ bonjour == ?o* && bonjour =~ (o[un]j?){2} ]]
$ echo $?
0
```

Bats

- *Bats = Bash Automated Testing System*
- **Lien**: `https://github.com/bats-core/bats-core`
- **Image**: DockerHub

  *« Bats is a TAP-compliant testing framework for **Bash**. It provides a simple way to verify that the UNIX programs you write behave as expected.*
  *A Bats test file is a Bash script with special syntax for defining test cases. Under the hood, each test case is **just a function with a description**. »*

- Autrement dit, il suffit d'utiliser des **commandes** de test
- Et on peut profiter des **expressions régulières**

Exemples de tests Bats

```
valgrind_options="--error-exitcode=1 --leak-check=full"

# On vérifie s'il y a une fuite mémoire
@test "No leak with default program" {
    run valgrind $valgrind_options ./program
    [ "$status" -eq 0 ]
}

# On vérifie que le fichier diagram.dot est valide
@test "File diagram.dot is valid" {
    run neato diagram.dot
    [ "$status" -eq 0 ]
}

# On vérifie les premières lignes
@test "Show help with -h" {
    run ./program -h
    [ "$status" -eq 0 ]
    [ "${lines[0]}" = "Usage: ./program [-h|--help]" ]
    [[ "${lines[0]}" == "[Uu]sage" ]]
    [[ "${lines[0]}" =~ "h(elp)?" ]]
}
```

Exemple plus complexe (1/2)

- Programme `tournament.c` qui **génère** une grille de tournoi
- En lisant sur l'**entrée standard**
- Chaque ligne correspond à un **joueur** ou une **équipe**
- On veut **limiter à 20** pour des raisons d'affichage

```
$ cat examples/tennis.in
Djokovic
Nadal
Federer
Murray
$ ./tournament -s table < examples/tennis.in
ID  Player    Day 1   Day 2   Day 3
--  --------  ------  ------  ------
 1  Djokovic  4       2       3
 2  Nadal     3       1       4
 3  Federer   2       4       1
 4  Murray    1       3       2
```

Exemple plus complexe (2/2)

```
# Vérifier l'affichage à espace près
#     -Z   ignorer les espaces en fin de ligne
#     -B   ignorer les lignes vides
@test "Tennis example with default options" {
    run diff -ZB examples/tennis-default.out \
                 <(./tournament < examples/tennis.in)
    [ "$status" -eq 0 ]
}

# Détecter le cas où on a plus de 20 équipes
@test "Too many players" {
    run ./tournament < examples/soccer-long.in
    [[ "$output" =~ "Error.*too many players" ]]
    [ "$status" -eq 1 ]
}

# Permettre le cas où on a exactement 20 équipes
# On ne peut pas utiliser | en combinaison avec run
@test "Twenty players is ok" {
    head -n 20 examples/soccer-long.in | ./tournament
    [ "$?" -eq 0 ]
}
```

Tests internes

Tests internes

- Aussi appelés tests en **boîte blanche**
- Basés sur l'**implémentation**
- Autrement dit, on se concentre sur la **structure** du programme
- À l'aide d'une **bibliothèque** spécifique au langage
- En **C**: Libtap, CUnit, Cmockery...

Comment tester?

- Cas **limites**
- Cas **particuliers**
- Couverture des **branchements**

Outils

- Assertions
- Graphes de flux
- Graphes d'appels de fonctions, ...

Programmation par contrat

- Déroulement du traitement régi par des **règles**
- Mises en place à l'aide d'**assertions**
- **Précondition**: hypothèse sur les entrées
- **Postcondition**: garantie sur les sorties
- **Invariant**: propriété en précondition et en postcondition

Mise en place

- En C, à l'aide de `#include <assert.h>`
- Fournit une **macro** `assert(expr)`
- Si `expr` est fausse, le programme **arrête**
- Peut être désactivée avec `-DNDEBUG`

Exemple (1/4)

- Reprenons le **module** `treemap.c`
- Et proposons des **préconditions** et **postconditions**

```
// Help functions
struct tree_node *treemap_get_node(struct tree_node *node,
                                   const char *key);
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value);
void treemap_print_node(const struct tree_node *node);
void treemap_delete_node(struct tree_node *node);

// Functions
void treemap_initialize(treemap *t);
char *treemap_get(const treemap *t, const char *key);
void treemap_set(treemap *t, const char *key, const char *value);
bool treemap_has_key(const treemap *t, const char *key);
void treemap_print(const treemap *t);
void treemap_delete(treemap *t);
```

Exemple (2/4)

```
struct tree_node *treemap_get_node(struct tree_node *node,
                                   const char *key) {
    if (node == NULL) {
        return NULL;
    } else {
        int cmp = strcmp(key, node->key);
        if (cmp == 0)
            return (struct tree_node*)node;
        else if (cmp < 0)
            return treemap_get_node(node->left, key);
        else
            return treemap_get_node(node->right, key);
    }
}
```

- **Précondition**: `assert(key != NULL)` à cause de `strcmp`
- **Postcondition**: aucune

Exemple (3/4)

```
void treemap_insert_node(struct tree_node **node,
                         const char *key,
                         const char *value) {
    if (*node == NULL) {
        *node = malloc(sizeof(struct tree_node));
        (*node)->key = strdup(key);
        (*node)->value = strdup(value);
        (*node)->left = NULL;
        (*node)->right = NULL;
    } else if (strcmp(key, (*node)->key) < 0) {
        treemap_insert_node(&(*node)->left, key, value);
    } else {
        treemap_insert_node(&(*node)->right, key, value);
    }
}
```

Préconditions:

- `assert(node != NULL)` à cause de `*node`
- `assert(key != NULL)` à cause de `strdup`
- `assert(value != NULL)` à cause de `strdup`

Postcondition:

- `assert(treemap_get_node(*node, key) != NULL)`

Exemple (4/4)

```
void treemap_set(treemap *t, const char *key, const char *value) {
    struct tree_node *node = treemap_get_node(t->root, key);
    if (node != NULL) {
        free(node->value);
        node->value = strdup(value);
    } else {
        treemap_insert_node(&(t->root), key, value);
    }
}
```

Préconditions:

- `assert(t != NULL)` à cause de `t->root`
- `assert(key != NULL)` à cause de l'appel à `treemap_get_node`
- `assert(value != NULL)` à cause de `strdup`

Postcondition:

- `assert(treemap_get(t, key) != NULL)`

Graphes de flux

Définition

- **Sommet**: une suite d'instructions sans branchement
- ou un expression booléenne
- **Arc**: représente un lien de contrôle entre deux sommets
- **Source**: sommet n'ayant aucun **prédécesseur**
- **Puits**: sommet n'ayant aucun **successeur**

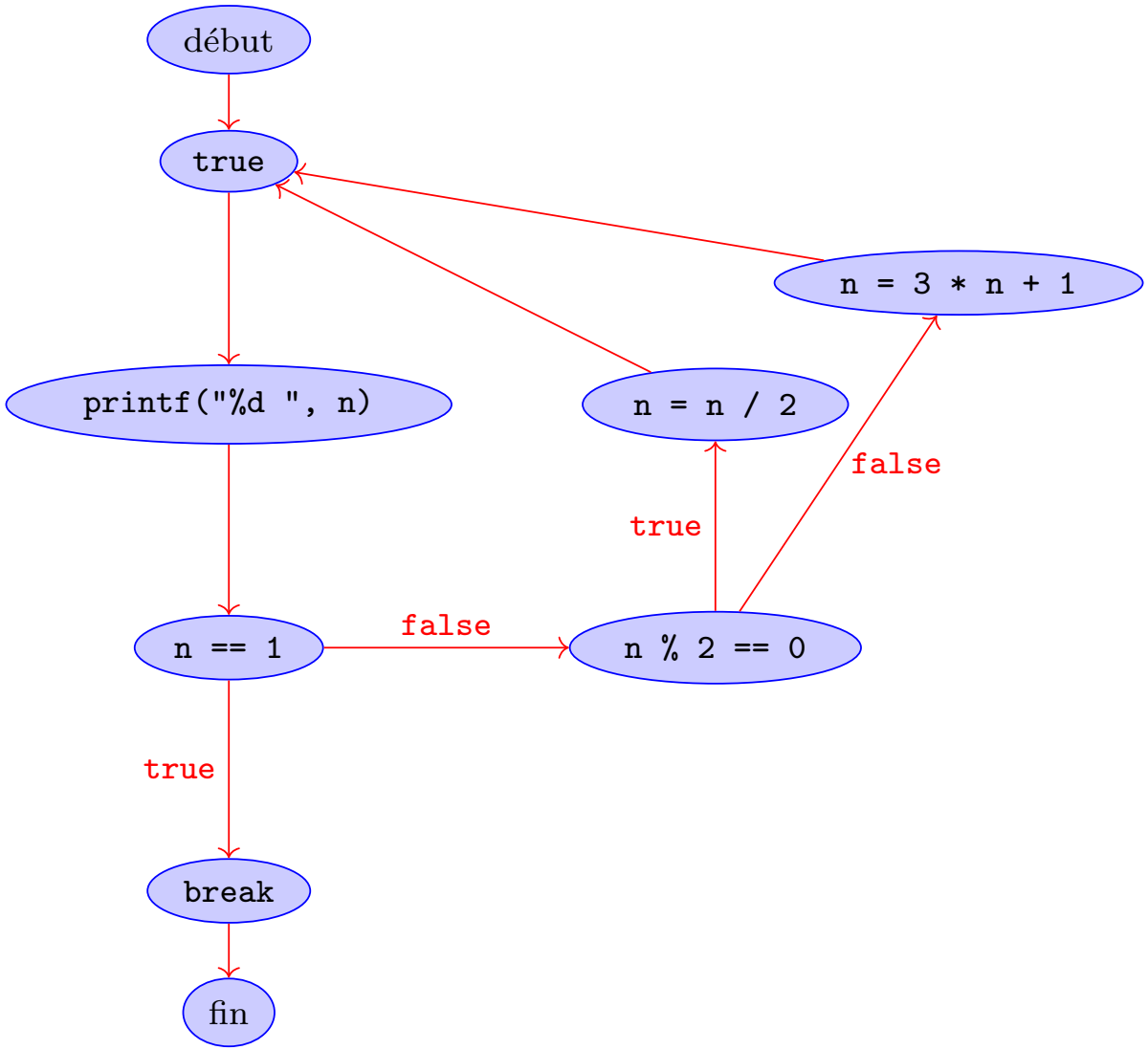
Graphe de flux

- Graphe représentant un bout de programme
- Ayant une **source unique** et
- Un **puits unique**

La fonction print_collatz_sequence (1/2)

```
1  void print_collatz_sequence(unsigned int n) {
2      while (true) {
3          printf("%d ", n);
4          if (n == 1)
5              break;
6          else if (n % 2 == 0)
7              n = n / 2;
8          else
9              n = 3 * n + 1;
10     }
11 }
```

La fonction `print_collatz_sequence` (2/2)



Couverture

Chemins

- On considère tous les chemins possibles
- **Commençant** à la source et
- **Terminant** au puits

Couverture

Un ensemble de chemin $\mathcal{C}$ forme une **couverture** si

- $\mathcal{C}$ couvre tous les **sommets** et tous les **arcs**
- Les chemins sont **linéairement indépendants**

Complexité cyclomatique

- Mesure de la **complexité** d'un bout de programme
- **Complexité cyclomatique** = cardinalité minimale d'une couverture
- Égale au nombre de **faces** du graphe de flux
- Incluant la face **extérieure**

Complexité cyclomatique et test

- Un **ensemble** de tests
- Qui couvrent tous les **branchements** (couverture)
- De façon **non redondante** (indépendance linéaire)
- Est considéré comme une **couverture** de qualité

La bibliothèque Libtap (1/4)

- **Dépôt**: `https://github.com/zorgnax/libtap`
- Assez **élémentaire**
- Respecte le protocole TAP

Utilisation

- Installation **manuelle**:

```
$ make
$ sudo make install
$ sudo ldconfig
```

- Utilisation à l'aide de `#include <tap.h>`
- Fournit plusieurs **macros-fonctions** pratiques

La bibliothèque Libtap (2/4)

- Déclarer une **série** de tests:

```
plan(num_tests)
plan(NO_PLAN)
plan(SKIP_ALL)
plan(SKIP_ALL, format, ...)
done_testing()
```

- Vérifier une **assertion**:

```
ok(test)
ok(test, format, ...)
```

- Comparer des **chaînes**:

```
is(got, expected)
is(got, expected, format, ...)
isnt(got, unexpected)
isnt(got, unexpected, format, ...)
```

La bibliothèque Libtap (3/4)

- Comparer des valeurs **numériques**:

```
cmp_ok(a, op, b)
cmp_ok(a, op, b, format, ...)
```

- Comparer des **octets**:

```
cmp_mem(got, expected, n)
cmp_mem(got, expected, n, format, ...)
```

- Chercher des **expressions régulières**:

```
like(got, expected)
like(got, expected, format, ...)
unlike(got, unexpected)
unlike(got, unexpected, format, ...)
```

- Écrire des messages de **diagnostic**:

```
diag(fmt, ...)
```

La bibliothèque Libtap (4/4)

- Vérifier si un programme **plante**:

```
dies_ok(code)
dies_ok(code, format, ...)
lives_ok(code)
lives_ok(code, format, ...)
```

- **Sauter** une série de tests:

```
skip(test, n)
skip(test, n, format, ...)
end_skip
```

- Identifier des fonctionnalités **non implémentées**:

```
todo()
todo(fmt, ...)
end_todo
```

Développement guidé par les tests

3 lois

Extraites de Wikipedia:

1. Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
2. Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Proposée par **Robert C. Martin** (2014)

Cycle de développement en 5 étapes

1. Ajouter un test ou plusieurs tests

Qui mettent en évidence le comportement souhaité

2. Lancer tous les tests

Les nouveaux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les nouveaux tests devraient réussir, les anciens aussi

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

Exemple: le module `set` (1/4)

Interface (fichier `set.h`):

```
struct set {                    // A set of integers
    int *elements;              // Its elements
    unsigned int size;          // Its cardinality
    unsigned int capacity; // Its capacity
};

// Create an empty set
struct set *set_create(void);
// Delete a set
void set_delete(struct set *set);
// Check if a set is empty
bool set_is_empty(const struct set *set);
// Check if a set contains an element
bool set_contains(const struct set *set, int element);
// Add an element to a set
void set_add(struct set *set, int element);
// Print a set to stdout
void set_print(const struct set *set);
```

Exemple: le module set (2/4)

Implémentation (fichier set.c):

```
struct set *set_create(void) {
    struct set *set = malloc(sizeof(struct set));
    set->elements = malloc(sizeof(int));
    set->capacity = 1;
    set->size = 0;
    return set;
}

void set_delete(struct set *set) {
    free(set->elements);
    free(set);
}

bool set_is_empty(const struct set *set) {
    return set->size == 0;
}

void set_print(const struct set *set) {
    printf("{");
    for (unsigned int i = 0; i < set->size; ++i) {
        if (i > 0) printf(", ");
        printf("%d", set->elements[i]);
    }
    printf("}");
}
```

Exemple: le module set (3/4)

Implémentation (fichier set.c):

```
int compare_ints(const void *i1, const void *i2) {
    return *(int*)i1 - *(int*)i2;
}

bool set_contains(const struct set *set,
                  int element) {
    return bsearch(&element, set->elements, set->size,
                   sizeof(int), compare_ints) != NULL;
}

void set_add(struct set *set,
             int element) {
    unsigned int i = 0;
    while (i < set->size && set->elements[i] < element)
        ++i;
    if (set->elements[i] == element) return;
    if (set->size == set->capacity) {
        set->capacity *= 2;
        set->elements = realloc(set->elements, set->capacity * sizeof(int));
    }
    for (unsigned int j = set->size; j > i; --j)
        set->elements[j] = set->elements[j - 1];
    set->elements[i] = element;
    ++set->size;
}
```

Exemple: le module set (4/4)

Tests (fichier `test_set.c`):

```c
#include "set.h"
#include <tap.h>

int main(void) {
    struct set *set = set_create();
    ok(set_is_empty(set), "created set is empty");
    ok(set->size == 0, "size of set is 0");
    set_add(set, 3);
    set_add(set, 5);
    set_add(set, 2);
    diag("Adding 3, 5, 2");
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is 3");
    ok(set_contains(set, 3), "set contains 3");
    ok(!set_contains(set, 1), "set does not contains 1");
    diag("Adding 5 again");
    set_add(set, 5);
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is still 3");
    set_delete(set);
    return 0;
}
```

Ajout de la fonction de suppression d'un élément

1. Ajouter un test ou plusieurs tests

Deux cas: élément **présent** ou **absent**

2. Lancer tous les tests

Les deux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les deux nouveaux tests et les anciens devraient réussir

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

INF3135
Construction et maintenance de logiciels

Cours 1: Présentation du cours

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Présentation du cours

Informations générales

- **Trimestre**: Été 2020
- **Titre du cours**: Construction et maintenance de logiciels
- **Sigle**: INF3135
- **Département**: Informatique
- **Enseignant**: Alexandre Blondin Massé
- **Courriel**: blondin_masse.alexandre@uqam.ca
- **Site personnel**: http://lacim.uqam.ca/~blondin
- **Bureau**: PK-4525
- **Coordonnateur**: Alexandre Blondin Massé
- **Site du cours**: http://lacim.uqam.ca/~blondin/fr/inf3135
- **Plan de cours**: cliquer ici

Description officielle (site de l'UQAM):

« Notions de base de la programmation procédurale et impérative en langage C sous environnement Unix/Linux (définition et déclaration, portée et durée de vie, fichier d'interface, structures de contrôle, unités de programme et passage des paramètres, macros, compilation conditionnelle). Décomposition en modules et caractéristiques facilitant les modifications (cohésion et couplage, encapsulation et dissimulation de l'information, décomposition fonctionnelle). Style de programmation (conventions, documentation interne, gabarits). Déboggage de programmes (erreurs typiques, traces, outils). Assertions et conception par contrats. Tests (unitaires, intégration, d'acceptation, boîte noire vs. boîte blanche, mesures de couverture, outils d'exécution automatique des tests). Évaluation et amélioration des performances (profils d'exécution, améliorations asymptotiques vs. optimisations, outils). Techniques et outils de base pour la gestion de la configuration. Système de contrôle de version. »

Objectifs du cours

- Maîtriser le développement d'un programme dans un **environnement Unix**
- Apprendre les bases du **langage C**
- **Documenter** convenablement un projet écrit en C
- Gérer la **construction** d'un programme C
- Gérer adéquatement **l'historique** d'un programme
- Apporter des **modifications** à un système développé par d'autres personnes
- Apprendre à utiliser des **bibliothèques** C
- Fournir une **couverture de tests** d'un programme
- Mettre en place des mécanismes d'**intégration continue** et de **déploiement continu**
- Travailler en **équipe** sur la construction d'un programme
- Bien **communiquer** par écrit et à l'oral ses idées à propos du développement d'un programme

Références

Contenu du cours

- *The C Programming Language*, de Kernighan et Ritchie
- *Manuel d'utilisation de Git*: disponible en ligne
- *Documentation de Make*: disponible en ligne
- *Documentation de Markdown*: documentation officielle et spécialisation pour GitLab

Politiques de l'UQAM

- Règlement 18 sur la tricherie et l'intégrité académique (plagiat): `http://r18.uqam.ca/`.
- Politique 16 contre le harcèlement sexuel: document pdf.

Ressources

Laboratoires

- **Description des labos**: dépôt GitLab
- **Démonstrateurs**: Jocelyn Bédard et Elyes Bejaoui
- **24 juin** et **1er juillet**: pas de labo $\rightarrow$ cours plus long

Liens importants

- **Théorie**: capsules vidéos à visionner **avant** les cours
- **Matériel**: groupe GitLab du cours
- **Support en ligne**: équipe Mattermost plutôt que canal Slack
- **Quiz**: sur Moodle

Modalités d'évaluation

Travaux pratiques (60%)

- **TP1 (seul)**: initiation au langage C (20%)
- **TP2 (seul)**: maintenance d'un programme (20%)
- **TP3 (seul ou équipe)**: construction d'un programme (20%)

Quiz (10% chacun, 40% au total)

- **Quiz 1**: 26 mai
- **Quiz 2**: 23 juin
- **Quiz 3**: 14 juillet
- **Quiz 4**: 11 août

Double seuil

Exceptionnellement, pas de double seuil

Environnement Unix

Terminal et shell

- **Terminal** = interface
- **Shell** = interpréteur de commandes

Interaction

Intermédiaire entre

- l'**utilisateur**,
- le système de **fichiers**
- les **applications**

Particulièrement utile pour des connexions à **distance**

INF1070 Utilisation et admininistration des systèmes informatiques

- Introduit comme cours **obligatoire** à l'automne 2018
- Répond à un **besoin** des enseignants, des étudiants et de l'industrie
- Permet d'**apprivoiser** la ligne de commande

Commandes de base

- `echo` — afficher un message
- `ls` — lister le contenu du répertoire
- `cat` — afficher un fichier
- `cd` — changer de répertoire
- `exit` — fermer le shell

Aller chercher de l'aide

```
$ man ls
$ man cat
$ man man
```

Afficher le contenu de fichiers

- `head` — Affiche les première lignes
- `tail` — Affiche les dernières lignes
- `less` (et `more`) — Affiche le fichier page par page
- `tac` — Affiche un fichier en commençant par la dernière ligne
- `rev` — Inverse chacune des lignes affichée
- `wc` — Affiche le nombre de lignes, mots et octets

Exemples

```
$ wc -l /usr/share/common-licenses/GPL-3
$ head -n 2 /usr/share/common-licenses/GPL-3
$ less /usr/share/common-licenses/GPL-3
```

Répertoires et chemins

- `tree` — affiche l'arborescence d'un répertoire
- `pwd` — affiche le chemin du répertoire courant
- `mv` — déplace/renomme des fichiers (ou répertoires)
- `cp` — copie (ou écrase) des fichiers
- `rm` — supprime des fichiers
- `mkdir` — crée des répertoires (vides)
- `rmdir` — supprime des répertoires vides
- `grep` — cherche les lignes correspondant à un motif dans un fichier
- `find` — rechercher des fichiers dans une hiérarchie de répertoires

Exemple

Suppression d'un répertoire **non vide**:

```
$ rm -rf nom-du-repertoire
```

Droits et utilisateurs

- `chmod` — modifie les droits d'un fichier (ou répertoire)
- `chown` — modifie l'utilisateur propriétaire d'un fichier
- `chgrp` — modifie le groupe propriétaire d'un fichier
- sudo — exécute une commande sous un autre utilisateur
- `su` — change d'utilisateur

Exemples

```
$ chmod +x nom-du-script
$ ls -l
$ sudo apt install pandoc
```

Autres commandes utiles

- `file` — affiche le type de fichier
- `date` — affiche et configure la date
- `touch` — modifie l'horodatage d'un fichier ou crée un fichier vide
- `stat` — affiche différentes informations sur un fichier

Exemples

```
$ date
$ file 01-intro.pdf hello.c
```

Gestionnaire de paquets

- Tâche courante: **installer** des logiciels/bibliothèques
- Gestion des **dépendances** entre paquets est complexe
- **Solution**: utiliser un **gestionnaire de paquets**
- En anglais, *package manager*

Exemples

- Aptitude (`apt`): Debian et ses dérivés
- Pacman (ArchLinux)
- RPM (Red Hat, Fedora, CentOS)
- MacPorts (MacOS)
- Homebrew (MacOS), etc.

Système Unix

Linux

Idéal

MacOS

Ça va pour le cours, mais attention aux **différences**

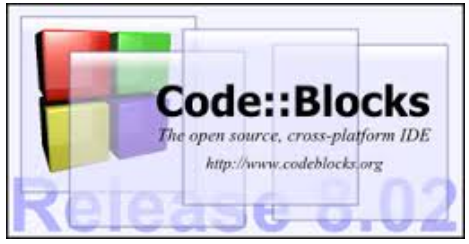
Windows

- **[Recommandé]** Installer une distribution **Linux** en partition double ou comme unique système (Ubuntu, Linux Mint)
- **[Alternative]** Installer une **machine virtuelle** (VirtualBox, VMWare)
- **Linux Subsystem** et autres variantes suffit pour le TP1, mais pas pour les deux autres TP
- Mieux vaut en profiter **dès le début** pour s'habituer à Linux

Environnement de développement

Environnement de développement

- Outil de **base** d'un programmeur
- En anglais, *IDE = integrated development environment*
- Quelques exemples:



- Pourtant, de nombreux programmeurs avancés préfèrent un simple éditeur de texte
- Certains plaident même que Unix est un EDD

Éditeur de texte

Offre très variée

- Notepad/Notepad++ (Windows)
- TextEdit (MacOS)
- Gedit (GNOME), souvent installé par défaut
- SublimeText (multiplateforme)
- Visual Studio Code
- Emacs
- Vi/Vim et ses dérivés (multiplateforme) $\rightarrow$ **préféré**

Éditeur de texte en ligne de commande

- **Nécessaire** pour certaines manipulations
- Édition de **dialogues** Git (lors d'un rebasement **interactif**)
- Modifications de fichiers **distants**

Vim

- Un des plus **anciens** éditeurs de texte
- Un des éditeurs de texte les **plus utilisés**
- Son ancêtre, `vi`, a été créé par Bill Joy en **1976**
- Le nom *Vim* vient de *Vi iMproved*
- Multiplateforme (Linux, MacOS, Windows)

Caractéristiques

- Très **mature**
- Interaction **directe** avec le terminal
- Installé par défaut avec plusieurs distributions Unix
- **Rapide**, en particulier pour la programmation à distance
- Hautement **configurable**
- Orienté **clavier** (GVim permet une utilisation limitée de la souris)
- Courbe d'apprentissage **difficile** pour les débutants
- Une bonne **configuration** est importante!

Le langage C

Bref historique

- **Années 70**: Naissance du langage, créé par **Ritchie** et **Kernighan**
- Origine du langage fortement liée à celle d'**Unix** $\rightarrow$ 90% du système Unix écrit en C
- **1978**: Publication du livre « The C Programming Language », par Kernighan et Ritchie
- **1983**: ANSI forme un comité pour **normaliser** le langage
- **1989** Apparition de la norme **ANSI-C**
- **1999**: Révision du standard (ISO C99)
- **2011**: Révision du standard (ISO C11)

Caractéristiques du langage

- **Bas niveau**: près du langage machine, contrôle fin de la mémoire, très efficace
- **Déclaratif**: le type des variables doit être déclaré
- **Flexible**: espacement, indentation
- **Structuré**: organisé en blocs (accolades)
- **Modulaire**: division en fichiers, compilation séparée
- **Flexible**: peu de vérification, pointeurs typés mais non contrôlés
- **Spécifique**: pour bien faire une tâche, adapté aux petits programmes et aux bibliothèques
- **Portable**: mais avec parfois un certain effort
- **Simple**: spécification assez courte
- **Verbeux**: il faut écrire beaucoup de code

Exemple

Fichier `maj.c`:

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

int main(void) {
    char c;
    while ((c = getchar()) != EOF) {
        putchar(toupper(c));
    }
    return 0;
}
```

Question

Que fait ce programme?

Cycle de compilation

Édition du programme source (`.c`)

À l'aide d'un éditeur de texte ou d'un EDD

Compilation (`.c` → `.o`)

- Indique les erreurs de **syntaxe**
- **Ignore** les fonctions et les bibliothèques invoquées

Édition de liens (*linking*)

- Fichiers `.o` assemblés pour former un binaire exécutable
- **Unix**: extension `.out` (par défaut), **Windows**: extension `.exe`

Exécution du programme

Invocation: `./<nom exécutable>`

Compilation « directe »

- Combinaison de la **compilation** et l'**édition des liens**
- Produit par **défaut** un exécutable nommé `a.out`:

```
$ gcc maj.c
$ ls -l a.out
```

- **Invocation** de l'exécutable:

```
$ ./a.out
```

- On peut **nommer** l'exécutable avec l'option `-o`:

```
$ gcc -o maj maj.c
$ ./maj
```

Compilation et édition des liens

- Parfois préférable de **séparer** compilation et édition des liens
- Projets de plus **grande envergure**
- On compile **chaque fichier** séparément
- Par **défaut** l'extension `.c` devient `.o`

```
$ gcc -c maj.c
# Équivalent
$ gcc -o maj.o -c maj.c
```

- Puis on produit l'**exécutable**:

```
$ gcc -o maj maj.o
```

- Et on **invoque** l'exécutable:

```
$ ./maj
```

Logiciel de contrôle de versions

Logiciel de contrôle de versions

- Permet de stocker un ensemble de **fichiers**
- Conserve en mémoire la **chronologie** de toutes les modifications effectuées
- Permet de **partager** des fichiers entre plusieurs personnes
- Permet de conserver différentes **versions** du code source d'un projet
- Permet également de gérer des versions **parallèles** (branches)
- Garantit l'**intégrité** des fichiers

Liste de logiciels connus

Nom	Type	Accès
Bazaar	distribué	libre
BitKeeper	distribué	propriétaire
CVS	centralisé	libre
Darcs	distribué	libre
Git	distribué	libre
Mercurial	distribué	libre
Subversion	centralisé	libre

- Dans ce cours, nous utiliserons **Git**
- **Obligatoire** pour la remise des travaux

Naissance de Git

Dates

- **2002**: Linus Torvalds utilise BitKeeper pour versionner Linux
- **6 avril 2005**: La version **gratuite** de BitKeeper est supprimée
- Torvalds décide de créer son **propre** logiciel: **Git**
- **18 avril 2005**: Git supporte l'opération de fusion de fichiers
- **16 juin 2005**: Utilisé pour conserver l'historique de Linux
- **Fin juillet 2005**: Junio Hamano devient le développeur principal

Nom - Extrait de Wikipedia

« Git » is British English slang for an unpleasant person.

« I'm an egotistical bastard, and I name all my projects after myself. First Linux, now git.»

— Linus Torvalds

Commandes les plus courantes

- **Créer** un nouveau projet: `git init`
- **Cloner** un projet existant: `git clone`
- **Sauvegarder** l'état courant du projet: `git commit`
- **Versionner** un nouveau fichier: `git add`
- **Ajouter** un fichier pour le prochain commit: `git add`
- **Consulter** l'historique: `git log`
- **Récupérer** des changements à distance: `git pull`
- **Téléverser** des changements à distance: `git push`

Apprendre les commandes

- Plusieurs commandes vues en **laboratoire**
- Vous devrez aussi en apprendre par **vous-même**
- Beaucoup d'**options** sont disponibles:

```
$ git remote --help
```

Hébergement de dépôts Git

La plupart des commandes Git se font de façon **locale**

- Chaque clone d'un dépôt est **autonome**
- L'historique est **purement local**

Cependant, il est pratique de pouvoir **partager** nos modifications

Pour cela, il existe des sites dédiés à l'**hébergement** de tels projets:

- Github
- Bitbucket
- GitLab

Dans ce cours, vous devrez utiliser le GitLab du département d'informatique

INF3135
Construction et maintenance de logiciels

Cours 2: Bases du C, partie 1

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Organisation du cours

Informations générales

- **Enseignant**: Alexandre Blondin Massé
- **Courriel**: blondin_masse.alexandre@uqam.ca
- **Site personnel**: `http://lacim.uqam.ca/~blondin`
- **Site du cours**: `http://lacim.uqam.ca/~blondin/fr/inf3135`
- **Plan de cours**: cliquer ici
- **Description des labos**: dépôt GitLab
- **Théorie**: capsules vidéos à visionner **avant** les cours
- **Matériel**: groupe GitLab du cours
- **Support en ligne**: équipe Mattermost plutôt que canal Slack

Modalités d'évaluation

Travaux pratiques (60%)

- **TP1 (seul)**: initiation au langage C (20%)
- **TP2 (seul)**: maintenance d'un programme (20%)
- **TP3 (seul ou équipe)**: construction d'un programme (20%)

Quiz (10% chacun, 40% au total)

- **Quiz 1**: 26 mai
- **Quiz 2**: 23 juin
- **Quiz 3**: 14 juillet
- **Quiz 4**: 11 août

Double seuil

Pas de double seuil

Résumé des capsules

Types

Numériques

- **Entiers**: `bool`, `char`, `short`, `int`, `long`, `long long`
- **Variante**: `signed` ou `unsigned`
- **Valeurs flottantes**: `float`, `double`, `long double`
- **Énumératifs**: `enum` (représentés par des entiers)

Types complexes

- Tableaux
- **Structures** (`struct`)
- **Unions** (`union`)
- **Vide**: `void`
- **Pointeurs**: plus tard
- **Synonymes**: avec `typedef`

Variables et constantes

Variables

- **Déclaration**: mémoire réservée
- **Initialisation** optionnelle
- Automatique (`auto`)
- Statique (`static`)
- Externe (`extern`)

Constantes

- Directive `#define`
- Mot réservé `const`
- Avec `enum`
- **Littéraux**: `u` (non signée), `l` (long), `0` (octal), `0x` (hexadécimal), `'c'` (caractère), `"..."` (chaînes)
- **Caractères spéciaux**: `\n`, `\t` `\\`, `\'`, `\"`

Structures de contrôle

Répétitives

- `for`
- `while`
- `do/while`
- `break/continue`

Conditionnelles

- `if/else if/else`
- `switch/case/break/default`

Opérateurs (1/2)

Arithmétiques

`+`, `-`, `*`, `/`, `%`

Comparaison

`==`, `!=`, `>`, `>=`, `<`, `<=`

Logiques

`!`, `&&`, `||`

Affectation

`=`, `+=`, `-=`, `*=`, `/=`, `%=`, `++`, `--`

Opérateurs (2/2)

Séquençage

`,` (rarement utilisé)

Ternaire

`? :`

Bit à bit

`&`, `|`, `^`

Taille en octets

`sizeof`

Conversions

Implicites

- Entre valeurs **entières**:

$$\texttt{bool} \rightarrow \texttt{char} \rightarrow \texttt{short} \rightarrow \texttt{int} \rightarrow \texttt{long} \rightarrow \texttt{long long}$$

- Entre valeurs **flottantes**:

$$\texttt{float} \rightarrow \texttt{double} \rightarrow \texttt{long double}$$

- Promotion automatique **entier** $\rightarrow$ **flottant**

Explicites

`(type)variable`

Priorité des opérateurs

Arité	Associativité	Par priorité décroissante
2	$\rightarrow$	`( )`, `[ ]`
2	$\rightarrow$	`->`, `.`
1	$\leftarrow$	`!`, `++`, `--`, `+`, `-`, `(int)`, `*`, `&`, `sizeof`
2	$\rightarrow$	`*`, `/`, `%`
2	$\rightarrow$	`+`, `-`
2	$\rightarrow$	`<`, `<=`, `>`, `>=`
2	$\rightarrow$	`==`, `!=`
2	$\rightarrow$	`&&`
2	$\rightarrow$	`||`
3	$\rightarrow$	`?  :`
1	$\leftarrow$	`=`, `+=`, `-=`, `*=`, `/=`, `%=`
2	$\rightarrow$	`,`

Exercices

Exercices

- Écrire une fonction

```
void print_n(char c, unsigned int n);
```

qui affiche `n` fois le caractère `c` sur `stdout`

- Écrire une fonction

```
void print_triangle(char c, unsigned int n);
```

qui affiche `n` fois le caractère `c` sur une première ligne, suivi de `n - 1` fois le même caractère sur une deuxième ligne, ainsi de suite, jusqu'à la dernière ligne qui contient `1` fois le caractère `c` sur `stdout`

- Écrire une fonction

```
void afficher_damier(char c, unsigned int m, unsigned int n);
```

qui affiche un damier (`c` alterné avec espace) de `m` lignes et `n` colonnes sur `stdout`

INF3135
Construction et maintenance de logiciels

Cours 3: Bases du C, partie 2

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Résumé des capsules

Tableaux

- Collection de données **homogènes** (de même type)
- Stockées de façon **contiguë** en mémoire
- Aucune **vérification** s'il y a dépassement

```
int t1[8];              // Réserve 8 * sizeof(int) sur le tas
double t2[n];           // Réserve n * sizeof(double) sur la pile
int t3[] = {5,2,0,1,3,4,7,6};
                        // Réserve 8 * sizeof(int) sur le tas
int t4[6] = {1,1};      // Initialise les 2 premières valeurs
char s1[] = "git";      // Équivalent à s1[4] = {'g','i','t','\0'};
char s2[4] = "ACGT"; // Équivalent à s2[4] = {'A','C','G','T'};
```

Chaînes de caractères

- Cas **particulier** de tableau
- Chaînes **littérales** délimitées par des guillemets " "
- Chaîne **bien formée**: doit terminer par le caractère \0
- On verra la bibliothèque `string.h` plus tard

Structures et unions

- Regroupent en un même bloc des données **hétérogènes**
- **Structures**: concaténation
- **Union**: alternative
- **Alignement**: compilateur décale selon l'architecture
- Opérateur `.` pour accéder à un **champ**

```
struct choice {
    bool is_number;
    union {
        float number;
        enum {YES, NO, MAYBE} answer;
    };
};
struct choice c;
// Accès: c.is_number, c.number, c.answer
```

- Affectation en bloc possible avec **conversion explicite**
- Peuvent être **imbriquées** et **combinées**
- Peuvent être **anonymes**
- `typdef` (**abusif**): permet d'omettre `struct`, `union` (et `enum`)

Fonctions

- **Unité de base** d'un programme avec les types
- Facilite la **réutilisation** et la **factorisation**
- Doivent être **bien nommées**
- Avec une syntaxe et une logique **uniforme**
- **Terminologie**: paramètres, arguments, fonction pure, à effets de bord, récursive
- **Valeur de retour**: une seule valeur permise ou `void`
- Paramètres et valeur de retour passés par **copie**
- On peut passer par **adresse** (on va y revenir)
- Fonction **spéciale**: `main` avec paramètres `argc` et `argv`
- Fonction utile: `printf`, pour affichage **formaté**
- Peuvent utiliser des variables **statiques**: durée de vie, tout le programme, portée limitée à la fonction

Compilation et Makefiles

GCC = *GNU Compiler Collection*

- `-c`: compilation seulement
- `-o`: spécifie le nom du fichier produit
- `-std=STD`: spécifie le standard
- `-Wall` et `-Wextra`: plus d'avertissements

Makefiles

- Règle **explicite**:

```
<cible>: <prérequis>
<tab><recette>
```

- **Variables**: définition avec `nom=valeur`, accès avec `$(nom)`
- **Cibles spéciales**: `.PHONY`
- **Invocation**: `make` ou `make <cible>`

Préprocesseur

Précompilation

- Directives interprétées par le préprocesseur **avant** la compilation
- Symbole # au **début** de la ligne
- On peut insérer des **espaces** entre # et la directive

Directives permises

- `#include`: **inclusion** d'un fichier externe
- `#define`: définition d'un **symbole** ou d'une **macro**
- `#undef`: **annulation** d'un symbole ou d'une macro
- `#ifdef`/`#ifndef`: **vérifie** si une macro est définie ou non définie
- `#if`/`#else`/`#elif`/`#endif`: structure **conditionnelle**
- `#error`: indique une **erreur fatale**
- `#pragma`: pour des traitements plus **spécifiques**

Présentation du TP1

Travail pratique 1

- Date de remise: **18 juin**
- **20%** de la note totale
- Doit être fait **seul**
- Dépôt GitLab: doit être *forké*, sa visibilité mise à *privé*, puis l'accès en mode *Developer* doit être donné à `blondin_al`
- Description du travail: le fichier source `sujet.md` sera disponible dans le dépôt cloné, ne pas le modifier
- À **compléter**: `canvascii.c`, `Makefile`, `.gitignore`, `README.md`
- À **modifier**: dans `check.bats`, supprimer `skip` pour activer le test

Objectifs

- **Initiation à C**: manipuler `stdin`, `stdout`, `argc` et `argv`
- **Makefile**: automatiser tâches, dont la compilation
- **Git**: faire des *commits* propres et atomiques, `.gitignore`
- **Documentation**: fichier `README`, documenter le code source

Exercices

Exercices

1. Écrire un programme qui

   - définit un **type** `struct square_matrix` permettant de représenter une matrice carrée de doubles (vous pouvez supposer que l'ordre de la matrice est limité à 10);
   - définit une **fonction** `initialize_matrix(m,n,v)` qui crée une matrice `m` de dimensions `n x n` et qui initialise chacune des entrées de la matrice à `v`;
   - définit une **fonction** `print_matrix(m)` permettant d'afficher le contenu de la matrice sur `stdout`.

2. Écrire un programme nommé `somme.c` qui prend exactement un argument de la forme `a,b,c`, où `a`, `b` et `c` sont des entiers, et qui affiche la somme des trois nombres sur `stdout`:

```
$ gcc -o somme somme.c
$ ./somme 1,2,3
6
```

INF3135
Construction et maintenance de logiciels

Cours 4: Outils de développement logiciel

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Exercices

Exercices

1. Écrire un programme qui

   - définit un **type** `struct square_matrix` permettant de représenter une matrice carrée de doubles (vous pouvez supposer que l'ordre de la matrice est limité à 10);
   - définit une **fonction** `initialize_matrix(m,n,v)` qui crée une matrice `m` de dimensions `n x n` et qui initialise chacune des entrées de la matrice à `v`;
   - définit une **fonction** `print_matrix(m)` permettant d'afficher le contenu de la matrice sur `stdout`.

2. Écrire un programme nommé `somme.c` qui prend exactement un argument de la forme `a,b,c`, où `a`, `b` et `c` sont des entiers, et qui affiche la somme des trois nombres sur `stdout`:

```
$ gcc -o somme somme.c
$ ./somme 1,2,3
6
```

Travail pratique 1

Résumé des capsules

Style de programmation

- C a été **standardisé** dans les années 80 (ANSI C89/C90)
- Mais aucun standard de **programmation** proposé

Quelques exemples

- Indian Hill
- NASA
- Noyau Linux (Linus Torvalds)
- GNU
- GNOME

« The Single Most Important Rule »

« Check the surrounding code and try to imitate it. »

— Extrait du site de GNOME

Documentation

Code source (*docstrings*)

- **Modules**: en-tête de fichier
- **Types**: sémantique des types
- **Fonctions**: description, paramètres et valeur de retour

Utilisateur

- **Manuel d'utilisation**: fichier `README`
- Bien exploiter le format Markdown

Bats

```
@test "Addition" {
    resultat="$(echo $((1 + 2)))"    # Pas obligé d'utiliser run
    [ "$resultat" -eq 3 ]            # Teste si sortie de echo est 3
}

@test "Avec run" {
    run echo $((1 + 2))              # Avec run
    [ "$status" -eq 0 ]              # Teste code de retour de echo
    [ "$output" == "3" ]             # Teste si sortie de echo est 3
}

@test "Plusieurs lignes" {
    run echo -e "ligne 1\nligne 2"   # Avec run
    [ "${lines[0]}" == "ligne 1" ]   # Teste contenu de la 1re ligne
    [ "${lines[1]}" == "ligne2" ]    # Teste contenu de la 2e ligne
}

@test "Un test désactivé" {
    skip                             # On désactive le test
    run echo "un autre test"
    [ "$status" -eq 1 ]              # Teste si echo échoue
}
```

Git

- `git log`: voir l'historique
- `git gr`: voir le graphe de l'historique
- `git init`: initialiser un dépôt
- `git clone`: copier un dépôt existant
- `git st`: voir l'état du projet
- `git checkout`: changer l'état du projet
- `git diff`: voir les différences
- `git show`: montrer un *commit*
- `git add`: indexer des modifications
- `git commit`: valider des modifications
- `git reset`: réinitialiser l'état d'un projet
- `git remote`: gérer des dépôts distants
- `git fetch`: télécharger des références
- `git pull`: récupérer des modifications
- `git push`: partager des modifications

GitLab-CI

```
# Mise à jour de apt et installation de Bats
before_script:
    - apt-get update -qq
    - git clone "https://github.com/bats-core/bats-core.git" /tmp/bats
    - mkdir -p /tmp/local
    - bash /tmp/bats/install.sh /tmp/local
    - export PATH="$PATH:/tmp/local/bin"

# Pour vérifier la compilation
build:
    stage: build
    script:
        - make

# Tests unitaires
test:
    stage: test
    script:
        - make test
```

- Possible de spécifier l'**image** Docker
- Structuration des *pipelines*
- Peut conserver les **résultats** (artéfacts)

Quiz 1

Quiz 1

Contenu

- **3** questions courtes ou à choix multiples
- **1** question à développement (compléter un programme)
- Entre 26 mai **20h00** et 27 mai **23h00**
- Durée: **1 heure**
- Lien Moodle

« Règles »

- Droit à documentation **écrite**
- Droit de lire des trucs sur le *web*
- Droit de tester des programmes sur ordi
- Interdiction de **discuter** avec autres étudiants
- Interdiction de **demander de l'aide** en ligne ou en personne
- Assurez-vous d'avoir une **bonne connexion**
- Assurez-vous d'avoir un **environnement de développement** en C

INF3135
Construction et maintenance de logiciels

Cours 5: Pointeurs

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Quiz 1

Retour sur le quiz 1

Statistiques

- **3** questions courtes ou à choix multiples
- **1** question longue
- **Moyenne**: 76.3%
- Questions courtes **bien réussies**
- Question longue **plus variable**

Problèmes rencontrés

- Envoi automatique du quiz
- Pas possible de déposer un fichier
- Pas le temps de joindre le fichier
- Copier-coller ne fonctionnait pas
- Autres?

Travail pratique 1

Travail pratique 1

- Date de remise: **18 juin**
- **20%** de la note totale
- Doit être fait **seul**
- Dépôt GitLab: doit être *forké*, sa visibilité mise à *privé*, puis l'accès en mode *Developer* doit être donné à `blondin_al`
- Description du travail: dans le fichier `sujet.md`
- À **compléter**: `canvascii.c`, `Makefile`, `.gitignore`, `README.md`
- À **modifier**: dans `check.bats`, supprimer `skip` pour activer le test

Résumé du chapitre 2

Chapitre 2: Outils de développement logiciel

Style de programmation

Présentation du code, syntaxe, nomenclature, organisation des fichiers

Documentation

Docstrings, Javadoc, Markdown

Bats

Installation, tests unitaires, variables spéciales (`status`, `output`, `lines`), invocation, `skip`, protocole TAP

Git

`log`, `gr`, `init`, `clone`, `status`, `checkout`, `diff`, `show`, `add`, `commit`, `reset`, `remote`, `fetch`, `pull`, `push`, etc.

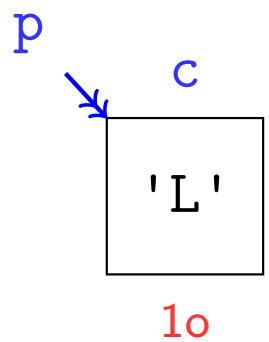
GitLab-CI

Fichier `.gitlab-ci.yml`, lancement de tests, lecture de *logs*

Résumé du chapitre 3

Adresse et pointeur

- **Adresse**: indique emplacement d'une donnée
- **Opérateur** `&`: retourne l'adresse d'une *left-value*
- **Pointeur**: *left-value* qui contient une adresse
- Pointeurs **typés**: `int*`, `double**`, `struct point*`, etc.
- Plusieurs **types**: nul, constant (`const`), générique (`void*`), de fonction



Deux « sortes » de pointeurs constants

- `const <type> *p`: pointeur en lecture seule
- `<type> *const q`: pointeur constant

Opérations sur les pointeurs

Déférencement

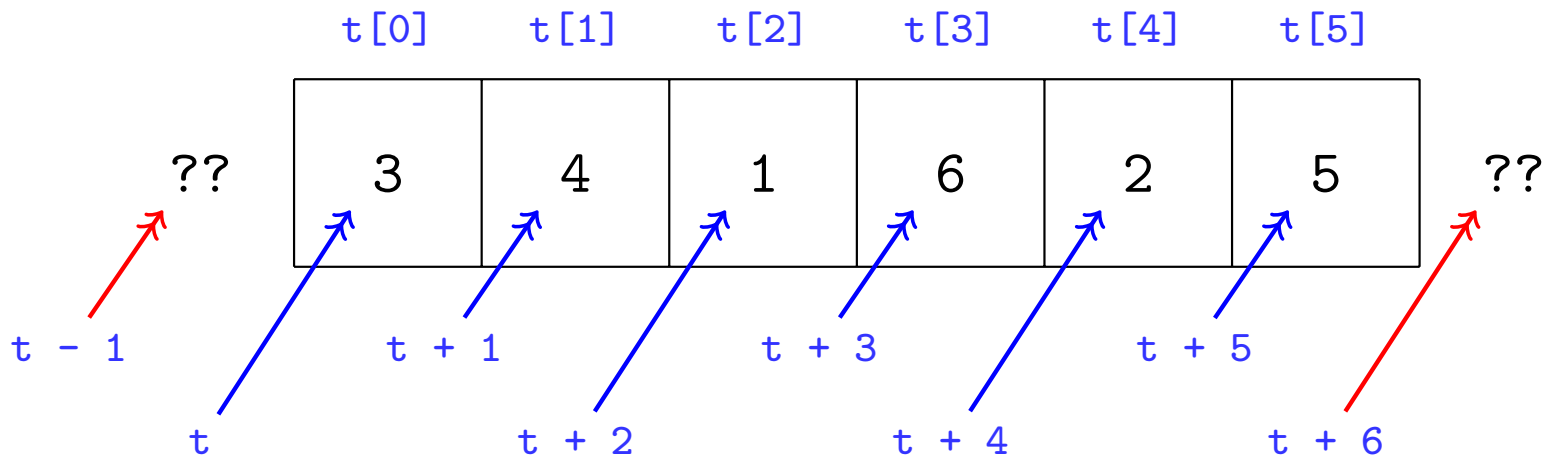
- `*`: accès à la donnée « pointée »
- `->`: déréférencement du champ d'une structure

Autres opérations

- `=`: affectation
- `(type*)`: conversion
- `==`/`!=`: égalité et différences d'adresses
- `<=`, `>=`, `<`, `>`: comparaison d'adresses
- `+`/`-`: retourne un pointeur décalé
- `++`/`--`: incrémentation et décrémentation

Tableaux et arithmétique des pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
```



- On a `t + i == &t[i]`
- De façon équivalente, `*(t + i) == t[i]`

Chaînes de caractères

```
char s1[10];     // Tableau de caractères de taille fixe
char *s2;        // Pointeur vers début d'une chaîne
const char *s2; // Chaîne en lecture seule
```

Bibliothèque `string.h`:

- `strcat`: concatène une chaîne à la suite d'une autre
- `strncat`: concatène une chaîne à une autre en tronquant
- `strcpy`: copie une chaîne dans une autre
- `strncpy`: copie une chaîne dans une autre en tronquant
- `strlen`: longueur d'une chaîne
- `strcmp`: compare deux chaînes
- `strncmp`: compare deux chaînes en tronquant
- `strchr`: cherche un caractère dans une chaîne de gauche à droite
- `strrchr`: cherche un caractère dans une chaîne de droite à gauche
- `strstr`: cherche une chaîne dans une autre de gauche à droite
- `strrstr`: cherche une chaîne dans une autre de droite à gauche
- `strtok`: segmente une chaîne en morceaux (*tokens*)

Pointeurs de fonctions

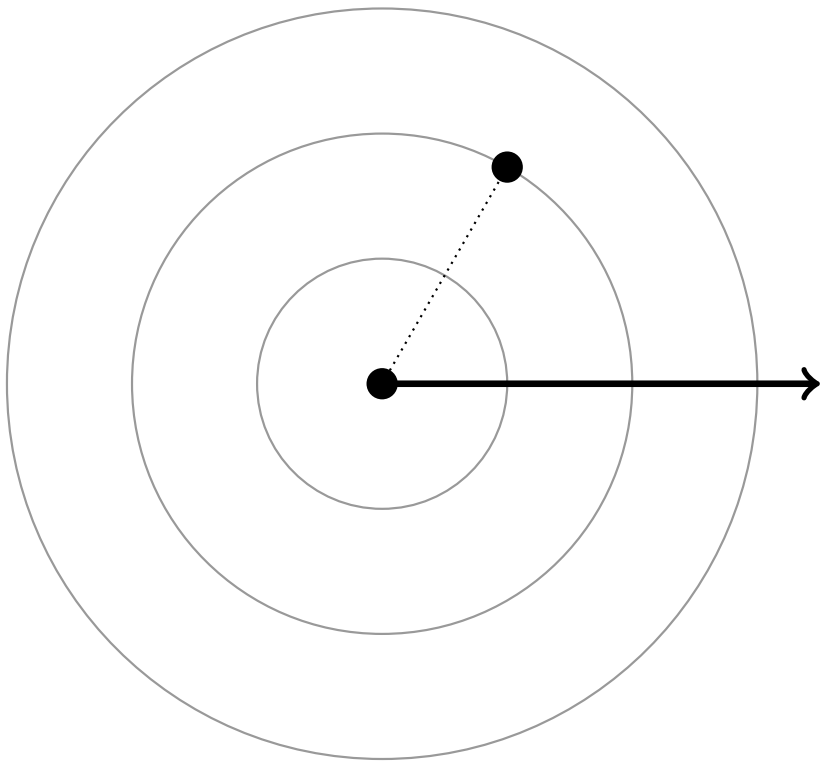
```c
// Pointeur de fonction de type int -> int
int (*f)(int x);
// Pointeur de fonction de type (int, int) -> int
int (*g)(int x, int y);
```

Bibliothèque `stdlib.h`:

```c
// Trie les valeurs d'un tableau
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int (*compar)(const void *, const void *));

// Effectue une fouille binaire dans un tableau
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

Exercice

1. Écrire une fonction `strlower` qui transforme une chaîne en minuscules
2. Écrire un programme qui trie un ensemble de points 2D selon l'ordre **polaire**, c'est-à-dire d'abord selon la distance par rapport à l'origine et, quand il y a égalité, selon l'angle



INF3135
Construction et maintenance de logiciels

Cours 6: Entrées et sorties

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

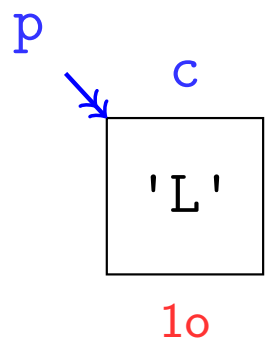
Été 2020

Table des matières

Résumé du chapitre 3

Adresse et pointeur

- **Adresse**: indique emplacement d'une donnée
- **Opérateur** `&`: retourne l'adresse d'une *left-value*
- **Pointeur**: *left-value* qui contient une adresse
- Pointeurs **typés**: `int*`, `double**`, `struct point*`, etc.
- Plusieurs **types**: nul, constant (`const`), générique (`void*`), de fonction



Deux « sortes » de pointeurs constants

- `const <type> *p`: pointeur en lecture seule
- `<type> *const q`: pointeur constant

Opérations sur les pointeurs

Déférencement

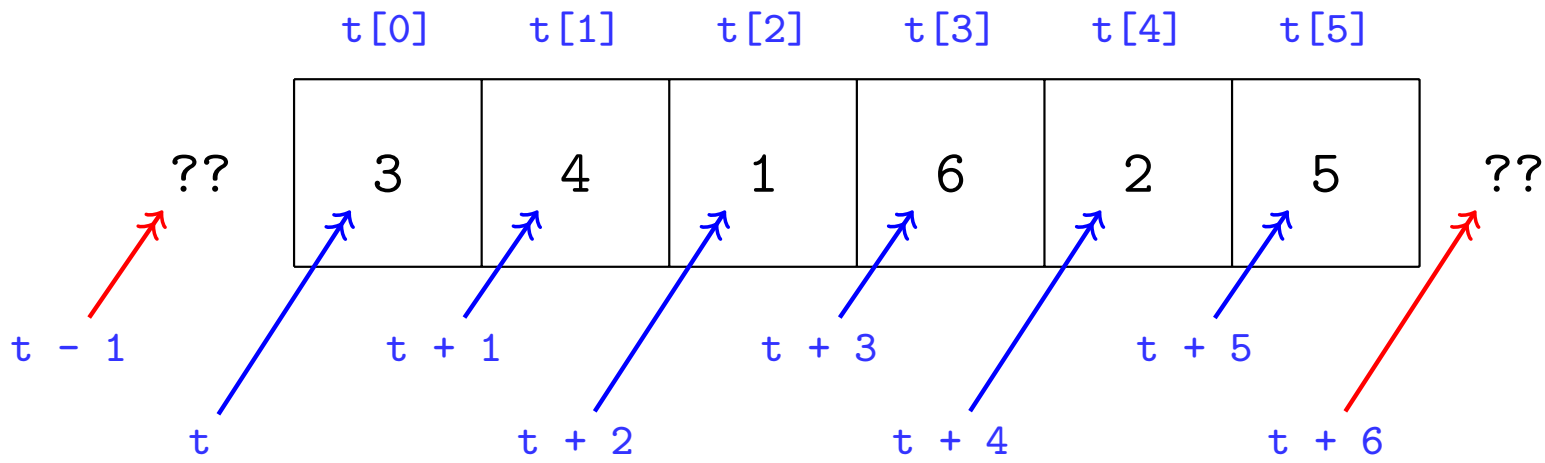
- `*`: accès à la donnée « pointée »
- `->`: déréférencement du champ d'une structure

Autres opérations

- `=`: affectation
- `(type*)`: conversion
- `==`/`!=`: égalité et différences d'adresses
- `<=`, `>=`, `<`, `>`: comparaison d'adresses
- `+`/`-`: retourne un pointeur décalé
- `++`/`--`: incrémentation et décrémentation

Tableaux et arithmétique des pointeurs

```
int t[] = {3, 4, 1, 6, 2, 5};
```



- On a `t + i == &t[i]`
- De façon équivalente, `*(t + i) == t[i]`

Chaînes de caractères

```
char s1[10];     // Tableau de caractères de taille fixe
char *s2;        // Pointeur vers début d'une chaîne
const char *s2; // Chaîne en lecture seule
```

Bibliothèque `string.h`:

- `strcat`: concatène une chaîne à la suite d'une autre
- `strncat`: concatène une chaîne à une autre en tronquant
- `strcpy`: copie une chaîne dans une autre
- `strncpy`: copie une chaîne dans une autre en tronquant
- `strlen`: longueur d'une chaîne
- `strcmp`: compare deux chaînes
- `strncmp`: compare deux chaînes en tronquant
- `strchr`: cherche un caractère dans une chaîne de gauche à droite
- `strrchr`: cherche un caractère dans une chaîne de droite à gauche
- `strstr`: cherche une chaîne dans une autre de gauche à droite
- `strrstr`: cherche une chaîne dans une autre de droite à gauche
- `strtok`: segmente une chaîne en morceaux (*tokens*)

Pointeurs de fonctions

```c
// Pointeur de fonction de type int -> int
int (*f)(int x);
// Pointeur de fonction de type (int, int) -> int
int (*g)(int x, int y);
```

Bibliothèque `stdlib.h`:

```c
// Trie les valeurs d'un tableau
void qsort(void *base,
           size_t nmemb,
           size_t size,
           int (*compar)(const void *, const void *));

// Effectue une fouille binaire dans un tableau
void *bsearch(const void *key,
              const void *base,
              size_t nmemb,
              size_t size,
              int (*compar)(const void *, const void *));
```

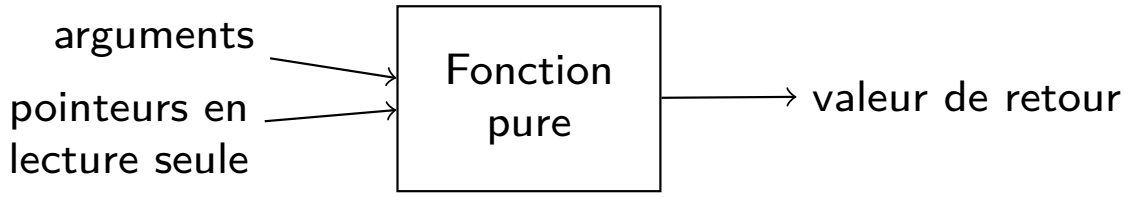
Travail pratique 1

Travail pratique 1

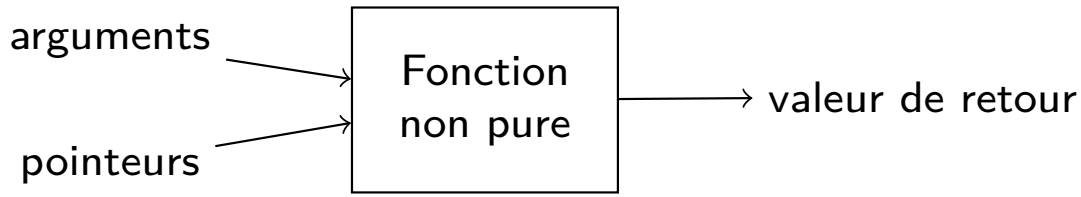
- Date de remise: **19 juin**, à **23h59**
- **20%** de la note totale
- Doit être fait **seul**
- Dépôt GitLab: doit être *forké*, sa visibilité mise à *privé*, puis l'accès en mode *Developer* doit être donné à `blondin_al`
- Description du travail: dans le fichier `sujet.md`
- À **compléter**: `canvascii.c`, `Makefile`, `.gitignore`, `README.md`
- À **modifier**: dans `check.bats`, supprimer `skip` pour activer le test

Résumé du chapitre 4

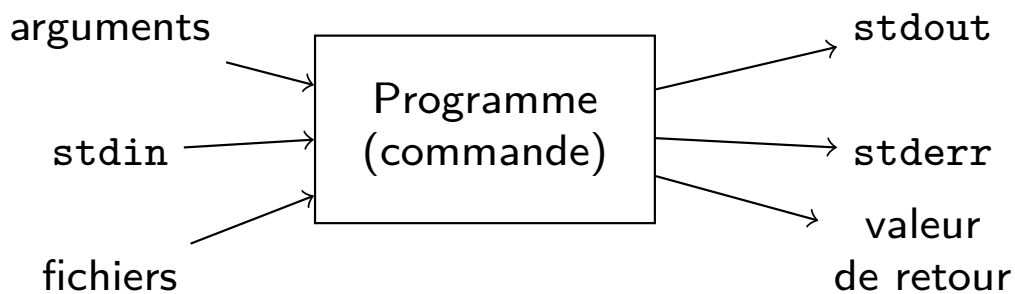
Entrées et sorties



Entrées



Sorties



La bibliothèque `stdio.h`

- *stdio* = *standard input output*
- Inclusion avec `#include <stdio.h>`

Macros:

- `EOF`: caractère de fin de fichier
- `stdin`, `stdout`, `stderr`: canaux standards
- `NULL`: pointeur nul, ...

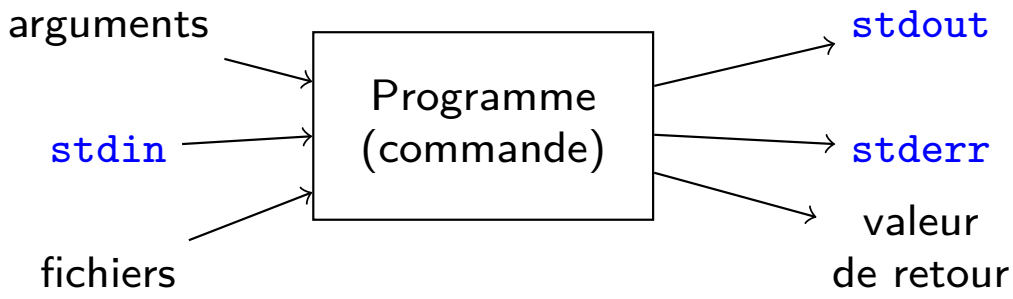
Types:

- `FILE`: flux (*stream*)
- `size_t`: taille en octets
- `fpos_t`: position dans un flux, ...

Variables externes: `optarg`, `opterr`, `optind`, `optopt` (gestion des arguments)

Plusieurs dizaines de **fonctions** (`man stdio`)

Canaux



3 canaux standards

- `stdin`: entrée standard (canal 0)
- `stdout`: sortie standard (canal 1)
- `stderr`: sortie d'erreur standard (canal 2)

Comportement par défaut (peut être redéfini)

- `stdin`: saisie clavier (*line buffered*)
- `stdout`: affichage sur le terminal (*line buffered*)
- `stderr`: affichage sur le terminal (*unbuffered*)

Redirections (1/2)

- Par défaut, `stdin` lit la saisie clavier
- Et `stdout`/`stderr` écrivent sur le terminal
- Ces comportements peuvent être modifiés avec des **redirections**
- Les redirections sont gérées par le **shell**
- Elles ne sont donc **pas gérées** par `argc` et `argv`

Syntaxe

- `commande < fichier`: redirige `fichier` sur `stdin`
- `commande > fichier`: redirige `stdout` dans `fichier`
- `commande 2> fichier`: redirige `stderr` dans `fichier`

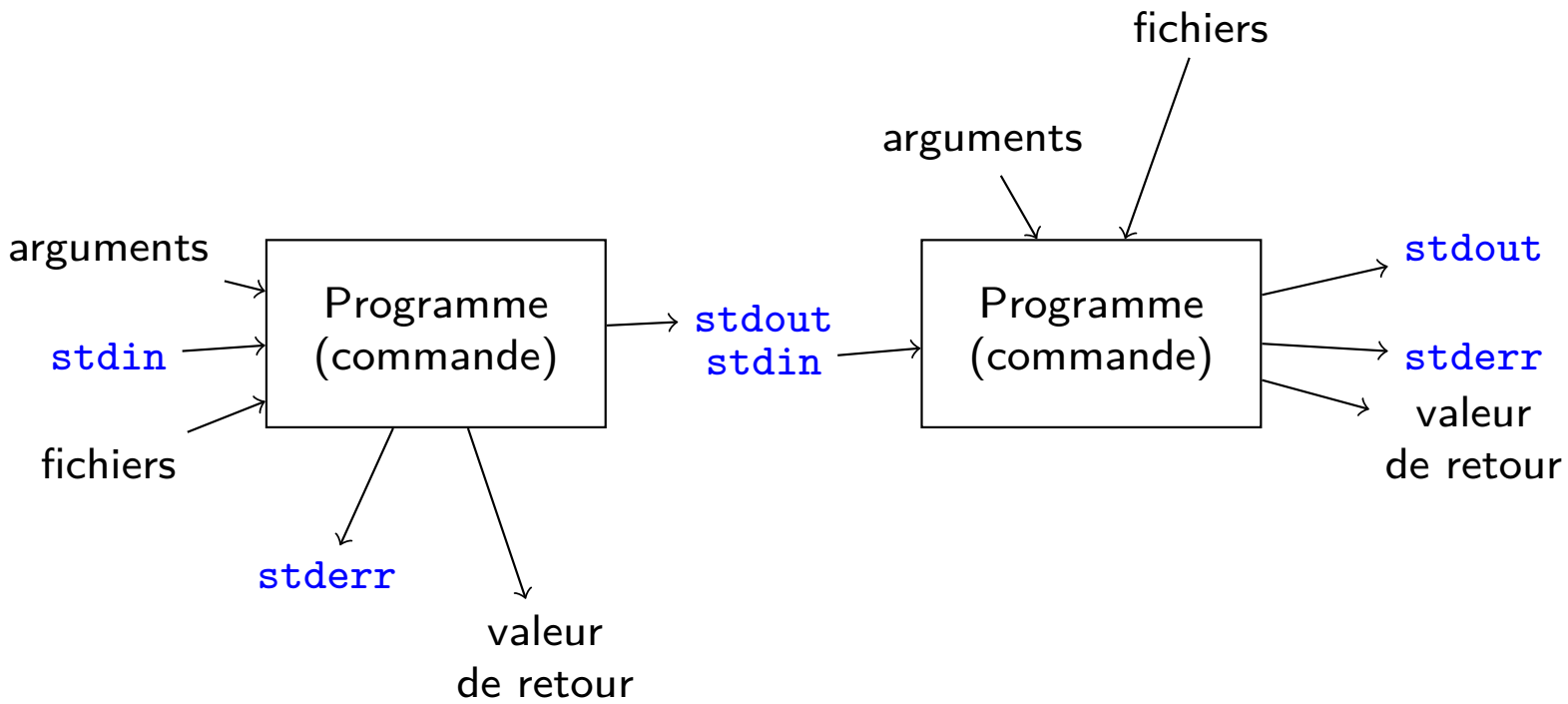
Redirections (2/2)

```c
#include <stdio.h>

int main(void) {
    char c = getchar();
    if (c == 'y')
        printf("Yes!\n");
    else
        fprintf(stderr, "No!\n");
    return 0;
}
```

```
$ cat redirections.y
yoyo
yaourt
$ cat redirections.z
zèbre
zoo
$ gcc redirections.c -o redirections
$ ./redirections < redirections.y
Yes!
$ ./redirections < redirections.z
No!
$ ./redirections < redirections.z 2> /dev/null
```

Tubes

- Permet d'**enchaîner** des programmes
- Le contenu écrit sur `stdout` par la première commande
- Est lu sur `stdin` par la deuxième commande
- **Syntaxe**: `commande1 | commande2 | ... | commandeN`



Filtres utiles

Filtre: programme souvent utilisé dans un tube

- `sort`: trie les lignes d'un flux
- `uniq`: supprime les doublons consécutifs
- `grep`: filtre selon une expression régulière
- `fmt`: formatte des données
- `pr`: formatte du texte pour impression
- `head`: affiche les premières lignes d'un flux
- `tail`: affiche les dernières lignes d'un flux
- `tr`: traduit caractère par caractère
- `sed`: transforme du texte
- `awk`: transforme du texte

Plus de détails dans **INF1070**

Consulter le manuel (`man`)

Exemple de filtres

Fichier `maj.c`:

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

int main(void) {
    char c;
    while ((c = getchar()) != EOF) {
        putchar(toupper(c));
    }
    return 0;
}
```

```
$ gcc maj.c -o maj
$ head -n 2 maj.c | ./maj
#INCLUDE <STDIO.H>
#INCLUDE <CTYPE.H>
$ head -n 2 maj.c | ./maj | tail -n 1
#INCLUDE <CTYPE.H>
$ grep 'char' maj.c | ./maj
    CHAR C;
    WHILE ((C = GETCHAR()) != EOF) {
        PUTCHAR(TOUPPER(C));
```

Valeur de retour

- En Unix, tout programme retourne une **valeur entière**
- Lorsque son exécution est **terminée**

Sémantique

- 0: le programme s'est terminé **« normalement »**
- $\neq$ 0: le programme s'est terminé **« anormalement »**

Récupérer la valeur de retour

- Contenue dans la **variable spéciale** $?
- Valeur de retour de la **dernière commande**

Valeur de retour en C

Fonction `main`

- Valeur retournée à l'aide de `return`
- Doit être **entière**
- Peut être **négative**
- Par défaut, retourne 0
- **Bonne pratique**: toujours spécifier la valeur de retour

La fonction `exit`

```c
void exit(int status);
```

- Permet de **terminer** l'exécution du programme proprement
- Vide et ferme les **flux** encore ouverts
- Supprime les **fichiers temporaires**

Combinaisons de commandes

- On peut **combiner** des commandes avec ; && et ||
- Le comportement dépend de la **valeur de retour**
- ;: deux commandes consécutives indépendantes
- &&: 2e commande exécutée seulement si la 1re réussit
- ||: 2e commande exécutée seulement si la 1re échoue

```
$ echo "commande" && echo $?
commande
0
$ echo "commande" || echo $?
commande
$ cat fichier.inexistant && echo $?
cat: fichier.inexistant: No such file or directory
$ cat fichier.inexistant ; echo $?
cat: fichier.inexistant: No such file or directory
1
$ cat fichier.inexistant || echo $?
cat: fichier.inexistant: No such file or directory
1
```

INF3135
Construction et maintenance de logiciels

Cours 7: Révision

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

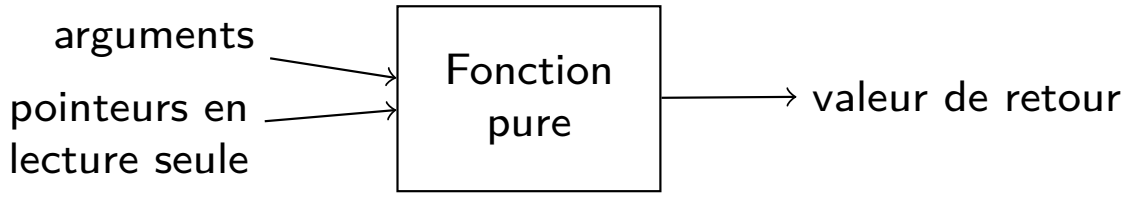
Travail pratique 1

Travail pratique 1

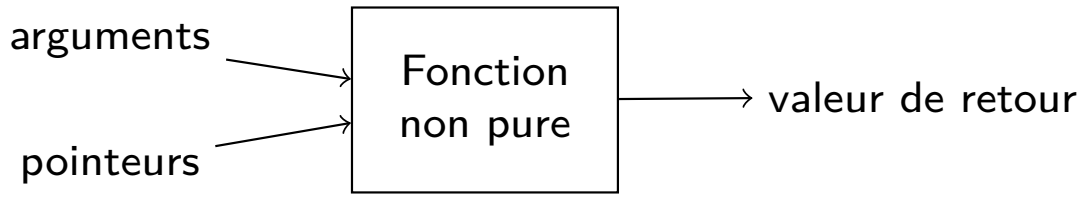
- Date de remise: **19 juin**, à **23h59**
- **20%** de la note totale
- Doit être fait **seul**
- Dépôt GitLab: doit être *forké*, sa visibilité mise à *privé*, puis l'accès en mode *Developer* doit être donné à `blondin_al`
- Description du travail: dans le fichier `sujet.md`
- À **compléter**: `canvascii.c`, `Makefile`, `.gitignore`, `README.md`
- À **modifier**: dans `check.bats`, supprimer `skip` pour activer le test

Résumé du chapitre 4

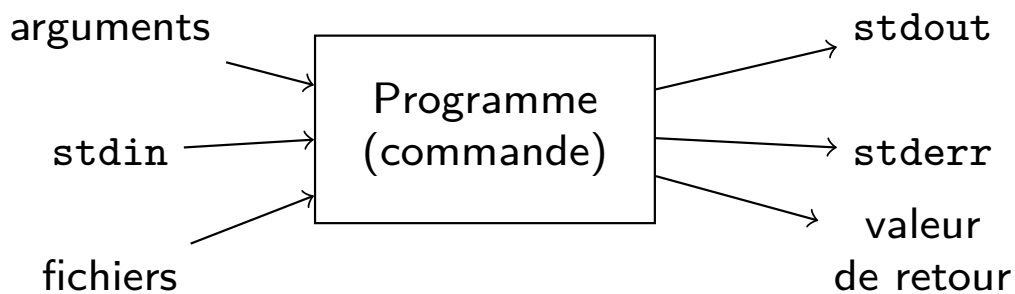
Entrées et sorties



Entrées



Sorties



La bibliothèque `stdio.h`

Macros:

- `EOF`: caractère de fin de fichier
- `stdin`, `stdout`, `stderr`: canaux standards
- `NULL`: pointeur nul, ...

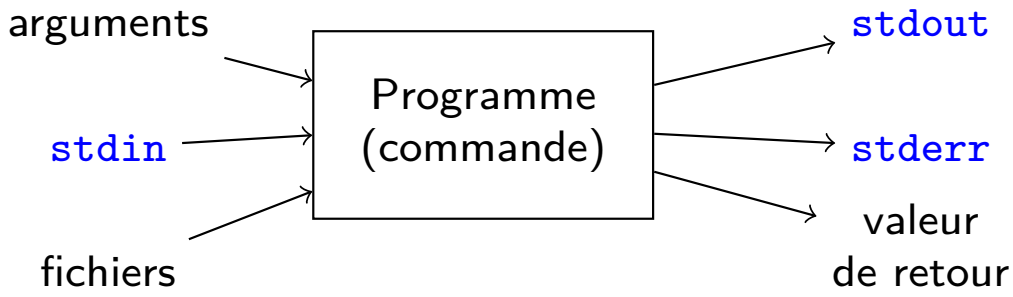
Types:

- `FILE`: flux (*stream*)
- `size_t`: taille en octets
- `fpos_t`: position dans un flux, ...

Variables externes: `optarg`, `opterr`, `optind`, `optopt` (gestion des arguments)

Plusieurs dizaines de **fonctions** (`man stdio`)

Canaux



3 canaux standards

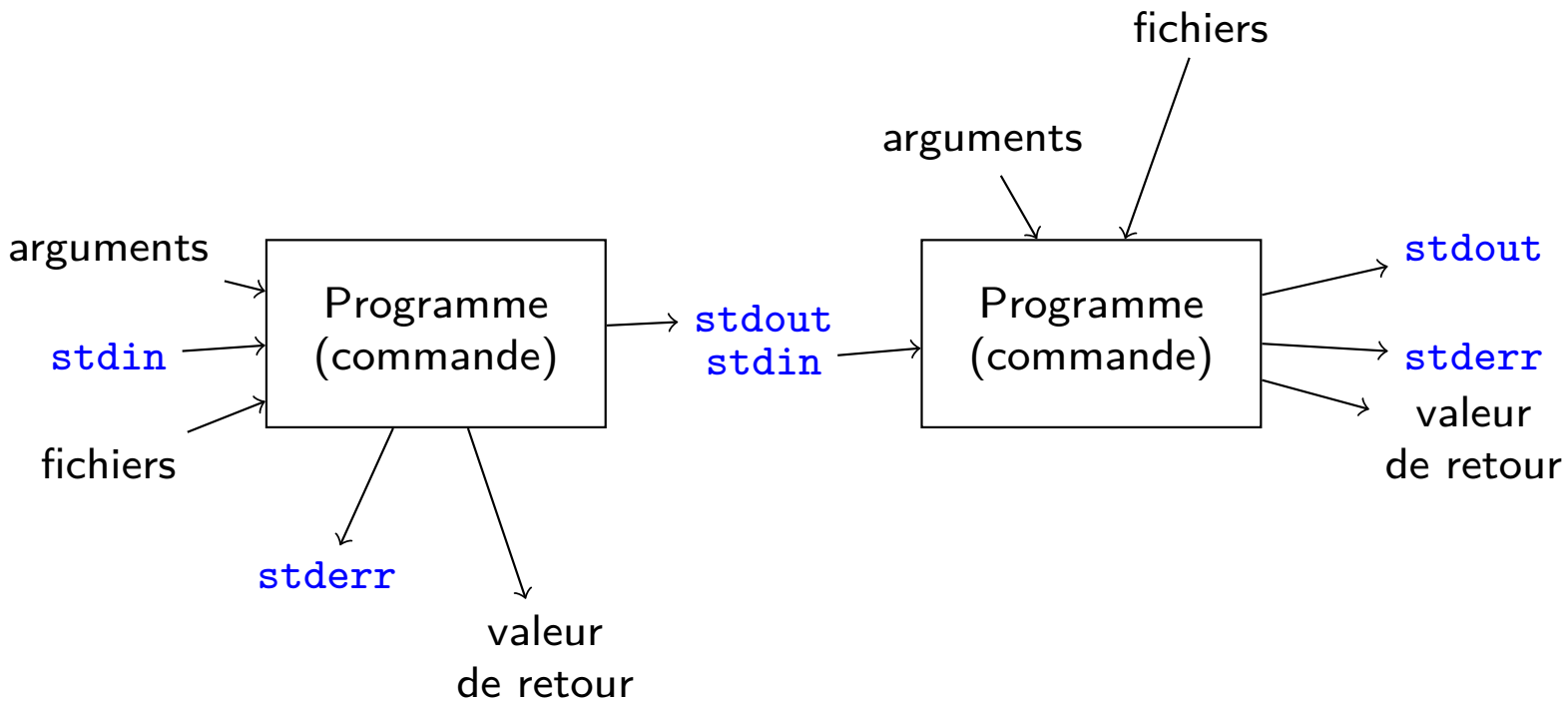
- `stdin`: entrée standard (canal 0)
- `stdout`: sortie standard (canal 1)
- `stderr`: sortie d'erreur standard (canal 2)

Redirections

- `commande < fichier`: redirige `fichier` sur `stdin`
- `commande > fichier`: redirige `stdout` dans `fichier`
- `commande 2> fichier`: redirige `stderr` dans `fichier`

Tubes

- Permet d'**enchaîner** des programmes
- Le contenu écrit sur `stdout` par la première commande
- Est lu sur `stdin` par la deuxième commande
- **Syntaxe**: `commande1 | commande2 | ... | commandeN`



Valeur de retour

Généralités

- 0: le programme s'est terminé **« normalement »**
- $\neq$ 0: le programme s'est terminé **« anormalement »**
- Valeur de retour de la **dernière commande**
- Contenue dans la **variable spéciale** `$?`

En C

- **Valeur** retournée par la fonction `main`
- Ou **argument** passé à la fonction `exit`

Combinaisons de commandes

- Avec `;` `&&` et `||`
- Valeur de retour **influence** `&&` et `||`

Commandes utiles

- **Graphviz**: https://graphviz.org/, pour générer des graphes, format DOT, en ligne de commande
- **QPDF**: http://qpdf.sourceforge.net/, pour manipuler des documents PDF
- **ImageMagick**: https://imagemagick.org/index.php, pour manipuler des images
- **FFMPEG**: https://ffmpeg.org/, pour manipuler des fichiers audio et vidéo
- **Gnuplot**: http://www.gnuplot.info/, pour générer des graphiques, des courbes, des surfaces, etc.

Quiz 2

Quiz 2

Contenu évalué

- **Chapitre 2:** Outils de développement logiciels
- **Chapitre 3:** Pointeurs
- **Chapitre 4:** Entrées et sorties
- Contenu des laboratoires

Forme: 1 heure 10, plage de 24 heures

- **3** questions courtes ou à choix multiples
- **1** question longue

Avant le quiz

Préparation

- Écouter/réécouter les **capsules**
- Réviser les **cours**
- Faire les **labos**
- Regarder les **solutions** des labos
- Attention à la **qualité**: généralité, efficacité, simplicité, lisibilité

Environnement

- Bonne **connexion** réseau
- **Environnement** de développement
- Terminal
- Débogueur si nécessaire
- Bien vérifier que vos réponses sont enregistrées
- Bien gérer votre temps

Exercices

Exercices

1. Écrivez un programme nommé `distr` qui lit sur l'entrée standard une **liste de mots** et qui écrit sur la sortie standard la **distribution de la longueur** des mots
2. Utilisez **Gnuplot** pour visualiser le résultat
3. Modifiez le programme précédent pour qu'il accepte **1 argument** correspondant à un nom du fichier à lire en entrée plutôt que sur `stdin`
4. Modifiez le programme précédent pour qu'il accepte **1 ou 2 arguments** correspondant à des noms de fichiers à lire en entrée et en sortie plutôt que sur `stdin` et `stdout`

INF3135
Construction et maintenance de logiciels

Cours 8: Structures de données

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Travaux pratiques

Travail pratique 1

- Date de remise: **19 juin**, à **23h59**
- **56 projets** remis (sur 64 inscrits)
- **Correction**: vers vendredi le **3 juillet**

Observations générales

- Attention de ne pas s'y prendre à la **dernière minute**
- Mettre en place le **Makefile** rapidement
- Mettre en place la **suite de tests** rapidement
- Vérifier **en continu** l'état du projet sur GitLab

Travail pratique 2

- Date de remise: **24 juillet**, à **23h59**
- 20% de la **note finale**
- Doit être fait **seul**

Sujet

- Disponible d'ici **vendredi**
- Sera présenté au **prochain cours**
- **Objectifs**: apporter des modifications à un programme, utilisation avancée de Git, modularité en C
- **Bibliothèques**: Jansson et Cairo

Résumé du chapitre 5

Généralités

Structure de données

Organisation **logique** d'un ensemble de données

Plusieurs objectifs

- **Simplifier** le traitement
- Offrir des opérations **efficaces**
- Économiser de l'espace **mémoire**

Interface et implémentation

- **Interface**: opérations supportées (type abstrait)
- **Implémentation**: organisation des données en mémoire, actions effectuées pour réaliser les opérations

Invariants et opérations

Invariant

- **Propriété** qui doit être satisfaite en tout temps
- Généralement vérifiable à l'aide d'une **fonction** booléenne

Opération

- Toute fonction qui **modifie** la structure de données
- Doit toujours préserver les **invariants**

Exemples

- **Chaîne de caractères**: termine par `'\0'`
- **Liste simplement chaînée**: le dernier noeud pointe vers `NULL`
- **Arbre binaire de recherche**: les clés respectent l'ordre, ...

Allocation dynamique

- Jusqu'à maintenant: mémoire réservée de façon **statique**
- Or, cette information n'est **pas toujours connu** à l'avance
- **Solution**: allouer l'espace mémoire de façon **dynamique**
- Dans la bibliothèque `stdlib.h`:

```
// Réserve un bloc de taille `size`
void *malloc(size_t size);
// Libère l'espace mémoire pointé par `ptr`
void free(void *ptr);
// Réserve un bloc de taille `nmemb * size` initialisé à 0
void *calloc(size_t nmemb, size_t size);
// Redimensionne un bloc de taille `size` déjà alloué dynamiquement
void *realloc(void *ptr, size_t size);
// Redimensionne un bloc de taille `nmemb * size`
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Mémoire

Fuite de mémoire

- Mémoire **réservée** mais non **référencée**
- Provoquée lorsqu'on appelle `malloc` ou `calloc`
- Et qu'on oublie de libérer avec `free`
- Souvent « **caché** » derrière une autre fonction (`strdup`)

Solutions

- Préférer un passage par **adresse**
- Et utiliser `malloc/calloc/free` seulement lorsqu'inévitable
- Fournir une fonction **complémentaire** qui libère l'espace alloué

Valgrind (`http://valgrind.org/`)

Permet de détecter des erreurs de **gestion de mémoire**

Structures de données

Piles (LIFO)

Implémentée avec liste simplement chaînée

File (FIFO)

Voir exercice à la fin

Tableaux dynamiques

Utilisation de `realloc`

Tableaux multidimensionnels

Utilisation de doubles pointeurs, initialisation et suppression

Arbres binaires de recherche

- Structure arborescente, avec clé, illustrée avec `treemap`
- Astuce des doubles pointeurs pour l'insertion

Exercices

Exercices

Compléter l'implémentation d'une **file** (queue):

```
// Initialise une file
void queue_initialize(queue *q);

// Indique si une file est vide
bool queue_is_empty(const queue *q);

// Ajoute un élément en fin de file
void queue_push(queue *q, unsigned int value);

// Récupère l'élément en tête de file
char queue_pop(queue *q);

// Affiche une file sur stdout
void queue_print(const queue *q);

// Détruit une file
void queue_delete(queue *q);
```

INF3135
Construction et maintenance de logiciels

Cours 9: Présentation du TP2

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Retour sur le quiz 2

Quiz 2

- **Date**: 23 juin dernier
- **Nombre de remises**: 57
- **Moyenne**: 7,58/10
- **Plus rapide**: 11 min 52 sec (et 10/10!)
- Pas trop de problèmes **techniques** (copier-coller)

Composition

- **Chapitre 2: Outils de développement logiciel**: 1/4
- **Chapitre 3: Pointeurs**: 1/4
- **Chapitre 4: Entrées et sorties**: 1/4
- **Question longue**: 1/4

Donc **256** quiz possibles

Résumé du chapitre 5

Généralités

Structure de données

Organisation **logique** d'un ensemble de données

Plusieurs objectifs

- **Simplifier** le traitement
- Offrir des opérations **efficaces**
- Économiser de l'espace **mémoire**

Interface et implémentation

- **Interface**: opérations supportées (type abstrait)
- **Implémentation**: organisation des données en mémoire, actions effectuées pour réaliser les opérations

Invariants et opérations

Invariant

- **Propriété** qui doit être satisfaite en tout temps
- Généralement vérifiable à l'aide d'une **fonction** booléenne

Opération

- Toute fonction qui **modifie** la structure de données
- Doit toujours préserver les **invariants**

Exemples

- **Chaîne de caractères**: termine par `'\0'`
- **Liste simplement chaînée**: le dernier noeud pointe vers `NULL`
- **Arbre binaire de recherche**: les clés respectent l'ordre, ...

Allocation dynamique

- Jusqu'à maintenant: mémoire réservée de façon **statique**
- Or, cette information n'est **pas toujours connu** à l'avance
- **Solution**: allouer l'espace mémoire de façon **dynamique**
- Dans la bibliothèque `stdlib.h`:

```
// Réserve un bloc de taille `size`
void *malloc(size_t size);
// Libère l'espace mémoire pointé par `ptr`
void free(void *ptr);
// Réserve un bloc de taille `nmemb * size` initialisé à 0
void *calloc(size_t nmemb, size_t size);
// Redimensionne un bloc de taille `size` déjà alloué dynamiquement
void *realloc(void *ptr, size_t size);
// Redimensionne un bloc de taille `nmemb * size`
void *reallocarray(void *ptr, size_t nmemb, size_t size);
```

Mémoire

Fuite de mémoire

- Mémoire **réservée** mais non **référencée**
- Provoquée lorsqu'on appelle `malloc` ou `calloc`
- Et qu'on oublie de libérer avec `free`
- Souvent **« caché »** derrière une autre fonction (`strdup`)

Solutions

- Préférer un passage par **adresse**
- Et utiliser `malloc/calloc/free` seulement lorsqu'inévitable
- Fournir une fonction **complémentaire** qui libère l'espace alloué

Valgrind (`http://valgrind.org/`)

Permet de détecter des erreurs de **gestion de mémoire**

Structures de données

Piles (LIFO)

Implémentée avec liste simplement chaînée

File (FIFO)

Voir exercice à la fin

Tableaux dynamiques

Utilisation de `realloc`

Tableaux multidimensionnels

Utilisation de doubles pointeurs, initialisation et suppression

Arbres binaires de recherche

- Structure arborescente, avec clé, illustrée avec `treemap`
- Astuce des doubles pointeurs pour l'insertion

Travail pratique 2

Travail pratique 2

- Date de remise: **26 juillet**, à **23h59**
- 20% de la **note finale**
- Doit être fait **seul** aussi
- **Dépôt**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp2
- **Sujet**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp2/-/blob/sujet/sujet.md
- Vous devez faire un *fork* du dépôt
- Et organiser votre développement en **branches**

Objectifs

- Se **familiariser** avec un logiciel développé en C par quelqu'un d'autre
- Apprendre à utiliser des **bibliothèques tierces** à l'intérieur d'un programme C, en consultant la documentation disponible
- Organiser le développement des modifications à l'aide de **branches**
- Soumettre les **modifications** en utilisant des **requêtes d'intégration** (*merge requests*)
- **Documenter** convenablement des requêtes d'intégration à l'aide du format Markdown
- S'assurer que les modifications apportées sont adéquates en proposant ou en mettant à jour un **cadre de tests** qui montre que les modifications n'entraînent pas de régression

Installation des dépendances

Dépendances

- PKG-config: pour la compilation et l'édition des liens
- Cairo, pour dessiner
- Jansson, pour manipuler le format JSON
- Bats, pour les tests unitaires « externes »
- Libtap, pour les tests unitaires « internes »

Installation

- Avec un **gestionnaire de paquets**: Apt, Homebrew, etc.
- Lancer `ldconfig` après avoir installé Libtap

Utilisation plus avancée de Git

- Résolution de **conflits**
- Organisation en **branches**
- **Rebasements** interactifs

Démonstration

- Résoudre un conflit
- `git stash`
- `git branch`
- `git checkout -b`
- `git cherry-pick`
- `git rebase`
- `git rebase -i`

INF3135
Construction et maintenance de logiciels

Cours 10: Modularité

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Entente d'évaluation

Proposition de modifications

- Déplacer la date de remise du TP2 au **31 juillet**
- Déplacer le quiz 3 au **21 juillet**
- Déplacer la date de remise du TP3 au **23 août** (date limite du registrariat: 28 août)
- Rendre le TP3 optionnel, sur une base individuelle
  - TP1 = 30% et TP2 = 30%
  - TP1 = 20%, TP2 = 20%, TP3 = 20%

Travail pratique 1

Travail pratique 1

Style de programmation

- **Syntaxe**: indentation, aération, longueur de ligne, etc.
- Pas de variables **globales**
- **Factorisation**: identifier les redondances et les faire disparaître

Git

- **Granularité** des *commits*
- **Messages** courts et significatifs
- **Syntaxe** uniforme (majuscule pas de point, verbe)

Travail pratique 2

Travail pratique 2

- **20%** de la note finale
- Fait **seul**
- Date de remise: **31 juillet**
- Le travail doit être réparti sur des **branches**
- Attention à la rédaction des **requêtes d'intégration**
- Attention à la qualité des *commits* et de leur message
- Mettre à jour votre .gitconfig
- Ne pas mettre de *commits* sur la branche `master`: utilisée pour se **synchroniser** avec le dépôt parent
- Récupérer les versions les plus à jour

Modularité

Modularité

Définition (extraite de Wikipedia)

« Modular programming is a software design technique that emphasizes separating the functionality of a program into independent, interchangeable modules, such that each contains everything necessary to execute only one aspect of the desired functionality. »

Caractéristiques

- **Séparation**: les préoccupations sont divisées en composantes
- **Indépendance**: les dépendances entre modules sont minimales
- **Interchangeabilité**: facilité de remplacer une composante
- **Spécifique**: un module règle une préoccupation précise

Terminologie

Varie selon le langage

- **Montage** (*assembly*): spécifique à Microsoft
- **Module**: un seul fichier ou un ensemble de fichiers
- **Paquet** (*package*): ensemble de modules
- **Composante**: une partie d'un système complexe

Exemples

- **Java**: un paquet (*package*)
- **C**: une paire de fichiers `.h`/`.c` (ou juste `.c`)
- **C++**: une paire de fichiers `.hpp`/`.cpp` (ou juste `.cpp`)
- **Python**: module = fichier, paquet = ensemble de modules
- **Haskell**: un fichier

Contenu d'un module

Interface

- Ce qui est **fourni** et **requis**
- Généralement visible de façon **« publique »**
- Documentation décrivant **utilisation**

Implémentation

- **Mise en oeuvre** de ce qui est déclaré dans l'interface
- Généralement **« privé »**
- Documentation décrivant le **développement**

Couplage et cohésion

Couplage (inter-modules)

« In software engineering, coupling is the degree of interdependence between software modules; a measure of how closely connected two routines or modules are; the strength of the relationships between modules. »

Cohésion (intra-module)

« In computer programming, cohesion refers to the degree to which the elements inside a module belong together. »

Objectifs

- **Minimiser** le couplage
- **Maximiser** la cohésion

Retour sur la précompilation

Directives au préprocesseur

- **Préfixées** par le symbole #
- **Lues et interprétées** avant même de procéder à la compilation
- Remplacement **textuel**

Exemples

- `#include`
- `#define`
- `#if`
- `#endif`
- `#ifndef`

Symboles

- Pour définir un **symbole** ou une macro, on utilise la directive

```
#define <identifiant> <valeur>
```

- Remplace toutes les **occurrences** de `<identifiant>` (comme mot) par `<valeur>`
- La valeur est donnée par **le reste de la ligne**
- Pour affecter une valeur sur **plusieurs lignes**, il faut utiliser le caractère \
- La **portée** du symbole s'étend jusqu'à la **fin du fichier** dans lequel il est défini
- Sauf si on trouve une commande

```
#undef <identificateur>
```

Exemple

Fichier `preproc.c`:

```c
#include <stdio.h>

#define i x

/*
 * Commentaire quelconque
 */
int main() {
    int i = 6, j;
    if (i) {
#undef i
        j = i * 2;
    }
    return 0;
}
```

Après `gcc -E preproc.c`:

```
# 1 "preproc.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 325 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "preproc.c" 2

# 1 "/usr/include/stdio.h" 1 3 4
# 64 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/sys/cdefs.h" 1 3 4
# 506 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_symbol_aliasing.h" 1 3 4
# 507 "/usr/include/sys/cdefs.h" 2 3 4
# 572 "/usr/include/sys/cdefs.h" 3 4
# 1 "/usr/include/sys/_posix_availability.h" 1 3 4
# 573 "/usr/include/sys/cdefs.h" 2 3 4
# 65 "/usr/include/stdio.h" 2 3 4
# 1 "/usr/include/Availability.h" 1 3 4
# 153 "/usr/include/Availability.h" 3 4
# 1 "/usr/include/AvailabilityInternal.h" 1 3 4
# 154 "/usr/include/Availability.h" 2 3 4
# 66 "/usr/include/stdio.h" 2 3 4

# 1 "/usr/include/_types.h" 1 3 4

⋮
```

Précompilation

```
# Pour voir le résultat de la précompilation
$ gcc -E fichier.c
# Pour conserver les commentaires
$ gcc -E -C fichier.c
# Pour enlever les lignes de la forme #<i> <valeur>
$ gcc -E -P fichier.c
```

- Pour éviter toute substitution **inattendue**...
- ...définir les symboles en **majuscules** exclusivement;

Définition de symboles à la compilation

- Il est possible de définir des symboles à la compilation seulement:

```
$ gcc -DLINUX fichier.c
```

- C'est équivalent à mettre la directive suivante dans `fichier.c`:

```
#define LINUX
```

- On peut également donner une **valeur** au symbole:

```
$ gcc -DLANGUE=FR fichier.c
```

- C'est équivalent à

```
#define LANGUE FR
```

Symboles prédéfinis

Fichier `predefini.c`:

```c
#include <stdio.h>

int main() {
    printf("Nom du fichier source courant: %s\n", __FILE__);
    printf("Numéro de la ligne courante: %d\n", __LINE__);
    printf("Date de compilation: %s\n", __DATE__);
    printf("Heure de compilation: %s\n", __TIME__);
    printf("Compilateur conforme à la norme ISO? %s\n",
           __STDC__ == 1 ? "oui" : "non");
    return 0;
}
```

Affiche:

```
Nom du fichier source courant: predefini.c
Numéro de la ligne courante: 5
Date de compilation: Nov  5 2019
Heure de compilation: 09:32:14
Compilateur conforme à la norme ISO? oui
```

Macro-fonctions

- Une **macro-fonction** est un symbole **paramétrable**
- **Syntaxe**:

```
#define f(x1,x2,...,xn) <corps>
```

- Le remplacement ne se fait que pour les **occurrences** de la forme

```
f(v1,v2,...,vn)
```

Exemple: la fonction ABS

Fichier `abs.c` :

```c
#include <stdio.h>

#define ABS(x) ((x) > 0 ? (x) : -(x))

int main() {
    int i = -6, j, k;
    j = ABS(i);
    k = ABS;
    printf("ABS(%d) = %d\n", i, j);
    return 0;
}
```

Après `gcc -E -P abs.c` :

```c
// Du code provenant de stdio.h

int main() {
    int i = -6, j, k;
    j = ((i) > 0 ? (i) : -(i));
    k = ABS;
    printf("ABS(%d) = %d\n", i, j);
    return 0;
}
```

Seule la **première** des trois occurrences de ABS est remplacée

Exemple: la fonction CARRE

Fichier `carre.c` :

```c
#include <stdio.h>

#define CARRE(x) ((x) * (x))
#define CARRE1(x) x * x
#define CARRE2(x) (x * x)

int main() {
    int x = 6, j, k, m, n;
    j = -CARRE1(x+1);
    k = -CARRE2(x+1);
    m = -CARRE(x+1);
    n = -CARRE(x++);
}
```

Après `gcc -E -P carre.c` :

```c
// Du code provenant de stdio.h

int main() {
    int x = 6, j, k, m, n;
    j = -x+1 * x+1;
    k = -(x+1 * x+1);
    m = -((x+1) * (x+1));
    n = -((x++) * (x++));
}
```

Seule la valeur de la variable `m` est celle attendue

Dangers associés aux macro-fonctions

- Mauvaise **substitution** si le corps et les paramètres ne sont pas correctement **parenthésés**
- Les paramètres peuvent être évalués **plusieurs fois**
- Erreurs lorsqu'il y a des **effets de bord**
- Ineffacité lors d'évaluations **multiples**

Conclusion

- Éviter d'utiliser les **macro-fonctions**
- Sauf dans de **rares** cas
- Et favoriser l'utilisation de fonctions de la façon habituelle.

Directives conditionnelles

Les directives

```
#if
#elif
#else
#ifdef
#ifndef
#endif
```

permettent d'indiquer au précompilateur d'effectuer certains traitements **avant compilation**

Utilisations fréquentes

– Gestion du paramétrage de **différentes versions** d'un même programme:

```
#ifdef LINUX
#  include "linux.h"
#endif
#ifdef MAC_OS
#  include "mac_os.h"
#endif
```

– Blocage des **inclusions multiples** des en-tête:

```
#ifndef PILE_H
#define PILE_H

[...]

#endif
```

Modules en C

Modules en C

- Typiquement, un **module** en C est divisé en **deux** fichiers
- Un premier `fichier.h`, qui contient l'**interface** ou l'en-tête (*header*)
- Et un second `fichier.c` qui contient l'implémentation de cette interface

Exemple

- `stack.h`: interface d'une pile
- `stack.c`: implémentation
- `test_stack.c`: utilisation de l'interface

Cycle de compilation

- **Étape 1**: compilation des fichiers sources

```
$ gcc -c stack.c
$ gcc -c test_stack.c
```

- **Étape 2**: édition des liens

```
$ gcc -o test_stack stack.o test_stack.o
```

- **Étape 3**: lancement de l'exécutable

```
$ ./test_stack
```

Qualité d'une interface

- Utilisation de synonymes adéquats (avec `typedef`) pour clarifier le rôle des arguments et de façon **uniforme**
- Placer le **type manipulé** comme premier argument
- **Uniformiser** l'ordre des autres arguments
- Inclure des **noms significatifs** de paramètres
- **Préfixer** ou **suffixer** le nom des fonctions et des types définis
- Uniformiser l'indirection: `struct foo` ou `struct foo *` partout
- Utiliser des valeurs d'**état** plutôt que des **nombres** arbitraires indiquant une erreur

Séparer l'interface de l'implémentation

- On peut alors utiliser les **services** offerts par l'interface sans se soucier de ce qui se passe réellement (approche en **boîte noire**)
- On **cache** à l'utilisateur les données/calculs privés pour que la communication avec les autres modules se fasse à l'aide de fonctions seulement (principe d'**encapsulation**)
- Si besoin est, on peut fournir **différentes implémentations** pour une même interface, afin d'améliorer les **performances**
- Plus facile à **maintenir** et à **tester**

Makefiles

Compilation de modules

- Lorsqu'un projet utilise plusieurs modules, il devient pénible de tout compiler **manuellement**
- **Solution**: utiliser un `Makefile`

Nous allons explorer les éléments suivants:

- Les règles implicites
- La fonction `wildcard`
- La fonction `patsubst`

Cas d'étude

Considérons un projet dans lequel on trouve **trois modules**:

- `game.h`/`game.c`
- `menu.h`/`menu.c`
- `credits.h`/`credits.c`

et **un** fichier principal:

- `main.c`

Dépendances explicites

```
game: game.o menu.o credits.o main.o
    gcc -o game game.o menu.o credits.o main.o

main.o: main.c
    gcc -c main.c

game.o: game.h game.c
    gcc -c game.c

menu.o: menu.h menu.c
    gcc -c menu.c

credits.o: credits.h credits.c
    gcc -c credits.c
```

Règles implicites

- La première amélioration possible est d'utiliser des **règles implicites**
- La syntaxe est la suivante

```
%.o: %.c
    gcc -c $<
```

- Cette règle indique que pour générer un fichier avec l'extension `.o`, il suffit de prendre le même fichier avec l'extension `.c` puis de lui appliquer la commande `gcc -c`

La fonction `wildcard`

- La deuxième amélioration consiste à utiliser la fonction `wildcard`
- La syntaxe est la suivante

```
$(wildcard *.c)
```

- Cet appel de fonction remplace l'expression par tous les fichiers avec l'extension `.c` dans le répertoire courant, séparés par des espaces
- Dans notre exemple, on obtiendrait

```
credits.c game.c main.c menu.c
```

La fonction patsubst

- La troisième amélioration consiste à utiliser la fonction `patsubst`
- La syntaxe est la suivante

```
$(patsubst %.c,%.o,$(wildcard *.c))
```

- Cette expression remplace toutes les extensions `.c` avec l'extension `.o`
- Dans notre exemple, on obtiendrait

```
credits.o game.o main.o menu.o
```

Makefile complet

```
CC = gcc
CFLAGS = -Wall
LFLAGS =
OBJECTS = $(patsubst %.c,%.o,$(wildcard *.c))
EXEC = main

$(EXEC): $(OBJECTS)
    $(CC) $(LFLAGS) -o $(EXEC) $(OBJECTS)

%.o: %.c
    $(CC) $(CFLAGS) -o $@ -c $<

.PHONY: clean

clean:
    rm -f $(OBJECTS) $(EXEC)
```

Bibliothèques

Édition des liens

Rappel sur les **étapes** de compilation:

- **Compilation**: `.c` $\rightarrow$ `.o`
- **Édition des liens**: `.o` $\rightarrow$ exécutable

Comment GCC gère-t-il ces deux étapes?

- **Compilation**: trouver les **en-tête** (fichiers `.h`)
- **Édition des liens**: trouver les fichiers binaires (fichiers `.o`) correspondants

Question

À quel **endroit** GCC cherche ces fichiers?

Emplacement des fichiers d'en-tête

- À la **compilation**, GCC tente de trouver les fichiers d'**en-tête** seulement (fichiers `.h`)
- Sur une installation **typique Unix**, GCC inspecte les répertoires suivants:

```
/usr/local/include
libdir/gcc/target/version/include
/usr/target/include
/usr/include
```

- Si le fichier `.h` se trouve ailleurs, il faut le **spécifier** avec l'option `-I` (pour *include*):

```
$ gcc -I<chemin> ...
```

Attention

Éviter les chemins **absolus** en **dur**, sinon le code n'est pas portable

Emplacement des binaires

- À l'**édition des liens**, GCC tente de trouver les **implémentations** correspondates
- Il inspecte **plusieurs répertoires**, qu'on peut connaître via la commande (sur Linux):

```
$ gcc -v hello.c -Wl,--verbose
```

- Si votre bibliothèque se trouve ailleurs, il faut le **spécifier**:

```
$ gcc -L<chemin> ...
```

Attention

Ici aussi, éviter les chemins **absolus** en **dur**

L'utilitaire `pkg-config`

- L'utilisation de **chemins absolus en dur** n'est pas acceptable si on souhaite qu'une application soit portable
- Une solution à ce problème consiste à utiliser le programme PKG-config
- Généralement, il suffit d'entrer

```
$ pkg-config --cflags <nom-bibliotheque>
```

pour obtenir les chemins contenant les **en-tête**

- Puis

```
$ pkg-config --libs <nom-bibliotheque>
```

pour obtenir les chemins contenant les **implémentations**

Deux types de bibliothèques

Statique:

- Extension: `.a` en Unix, `.lib` sous Windows
- La bibliothèque est **incluse** dans l'exécutable
- **Avantage:** réduit les dépendances
- **Inconvénient:** exécutables plus volumineux

Dynamique:

- Extension: `.so` en Unix, `.dll` sous Windows
- La bibliothèque est **liée dynamiquement**
- **Avantage:** évite les **redondances**, exécutables moins volumineux
- **Inconvénient:** nécessite une installation, problèmes de version

Exemple: Vec3D (1/2)

- Supposons que nous avons conçu une bibliothèque supportant la manipulation de **vecteurs**
- Voir les fichiers `vec3d.h` et `vec3d.c`
- Tout d'abord, on **compile** le fichier `vec3d.c` en objet `vec3d.o`:

```
$ gcc -o vec3d.o -c vec3d.c
```

- Ensuite, on crée la bibliothèque statique:

```
$ ar -cvq libvec3d.a vec3d.o
```

- On peut ensuite l'inclure via l'instruction en autant que l'**en-tête** et l'**implémentation** soient disponibles

```
#include <vec3d.h>
```

Exemple: Vec3D (2/2)

- Par exemple, supposons que les fichiers `vec3d.h` et `libvec3d.a` se trouvent respectivement dans les répertoires

```
/Users/blondin_al/clib/include
/Users/blondin_al/clib/lib
```

- Alors il suffit de compiler avec la commande

```
$ gcc -I/Users/blondin_al/clib/include \
>     -c test_vec3d.c
```

- Puis de compléter l'**édition des liens** avec

```
$ gcc -L/Users/blondin_al/clib/lib -o \
>     test_vec3d test_vec3d.o -lvec3d
```

Les bibliothèques `unistd.h` et `getopts.h`

- Facilite le **traitement** des arguments récupérés par la fonction `main`
- Autrement dit, simplifie le traitement de `argc` et `argv`

Deux types d'**options**:

- **Courtes:** (`unistd.h`) un tiret, suivi d'une lettre, par exemple

```
$ ls -als
$ gcc -o tp1 tp1.c
```

- **Longues:** deux tirets, suivis d'un mot pouvant contenir des tirets, par exemple:

```
$ valgrind --leak-check=yes ./prog
$ ./isomap --help
```

- La bibliothèque `getopts.h` permet de gérer les options courtes et longues simultanément

La bibliothèque Cairo (1/2)

- Cairo est une bibliothèque permettant de dessiner des images **vectorielles**
- Elle supporte différents **formats**: PNG, PDF, SVG, etc.
- Site officiel: https://www.cairographics.org/
- Pour compiler/faire l'édition des liens, mieux vaut utiliser PKG-config

```
EXEC = hello
CFLAGS = $(shell pkg-config --cflags cairo)
LFLAGS = $(shell pkg-config --libs cairo)

$(EXEC): $(EXEC).o
	gcc $< -o $@ $(LFLAGS)

$(EXEC).o: $(EXEC).c
	gcc -o $@ $(CFLAGS) -c $<

.PHONY: clean

clean:
	rm -f *.o $(EXEC)
```

La bibliothèque Cairo (2/2)

```c
#include <cairo.h>

int main (int argc, char *argv[]) {
    cairo_surface_t *surface
        = cairo_image_surface_create(CAIRO_FORMAT_ARGB32,
                                     240, 80);
    cairo_t *cr = cairo_create (surface);

    cairo_select_font_face(cr, "serif",
        CAIRO_FONT_SLANT_NORMAL, CAIRO_FONT_WEIGHT_BOLD);
    cairo_set_font_size (cr, 32.0);
    cairo_set_source_rgb (cr, 0.0, 0.0, 1.0);
    cairo_move_to (cr, 10.0, 50.0);
    cairo_show_text (cr, "Hello, world");

    cairo_destroy (cr);
    cairo_surface_write_to_png (surface, "hello.png");
    cairo_surface_destroy (surface);
    return 0;
}
```

La bibliothèque SDL

- SDL est une autre bibliothèque C permettant de concevoir des applications graphiques
- Plusieurs autres applications sont basées sur SDL (Pygame, Kivy, etc.)
- Versions majeures: SDL**1.2** et SDL**2.0**
- Interaction de bas niveau avec les périphériques **graphiques** et **audio**
- Supporte seulement les formats BMP et WAV par défaut
- Bibliothèques **compagnonne** pour support PNG et MP3
- Voir l'application Maze

INF3135
Construction et maintenance de logiciels

Cours 11: Les branches en Git

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Dates et contenu à venir

Point sur le cours

Dates

- **Remise du TP2**: 31 juillet
- **Quiz 3**: 21 juillet
- **Sujet du TP3**: avant le 31 juillet
- **Remise du TP3**: 23 août
- **TP3**: optionnel, sur une base individuelle

Contenu

- **Chapitre 6**: modularité
- **Chapitre 7**: maintenance
- **Chapitre 8**: tests et intégration continue

Quiz 3

Quiz 3

Matière évaluée

- Date: **21 juillet**
- **Chapitre 5**: structures de données
- **Chapitre 6**: modularité
- **Labo 8**: modules
- **Labo 9**: les branches sous Git

Forme du quiz

- 1 question courte sur chapitre 5
- 1 question courte sur chapitre 6
- 1 question courte sur Git
- 1 question longue

Travail pratique 2

Travail pratique 2

- **20%** de la note finale
- Fait **seul**
- Date de remise: **31 juillet**
- Le travail doit être réparti sur des **branches**
- Attention à la rédaction des **requêtes d'intégration**
- Attention à la qualité des *commits* et de leur message
- Mettre à jour votre .gitconfig
- Ne pas mettre de *commits* sur la branche `master`: utilisée pour se **synchroniser** avec le dépôt parent
- Récupérer les versions les plus à jour

Modularité

Modularité

Caractéristiques

- **Séparation**: les préoccupations sont divisées en composantes
- **Indépendance**: les dépendances entre modules sont minimales
- **Interchangeabilité**: facilité de remplacer une composante
- **Spécifique**: un module règle une préoccupation précise
- **Réutilisation**: un module est souvent réutilisable

Remarques

- Varie selon le **langage**
- Notion d'**interface** et d'**implémentation**
- Séparer les **préoccupations**

Fonctions variadiques

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

int max(int n, ...) {
    va_list list;
    va_start(list, n);
    if (n <= 0) return 0;
    int m = va_arg(list, int);
    for (unsigned int i = 1; i < n; ++i) {
        int m2 = va_arg(list, int);
        m = m2 > m ? m2 : m;
    }
    va_end(list);
    return m;
}

int main (void) {
    printf("%d ", max(3, 8, 2, -3));
    printf("%d\n", max(5, -4, -1, -8, 7, 6));
    return 0;
}
```

Résultat:

```
8 7
```

Précompilation

Directives `#define`/`#undef`

- Macros
- Macros-fonctions
- Attention aux pièges

Directives conditionnelles

- `#ifdef`/`#ifndef`: vérifie si une macro est définie ou non définie
- `#if`/`#else`/`#elif`/`#endif`: structure conditionnelle

Fonctions et macro-fonctions variadiques

Nombre quelconque d'**arguments**

Bibliothèque

Définition (extraite de Wikipedia)

*« En informatique, une bibliothèque logicielle est une collection de **routines**, qui peuvent être déjà compilées et prêtes à être utilisées par des programmes. Les bibliothèques sont enregistrées dans des fichiers semblables, voire identiques aux fichiers de programmes, sous la forme d'une collection de fichiers de code objet rassemblés accompagnée d'un index permettant de **retrouver facilement** chaque routine. Le mot librairie est souvent utilisé à tort pour désigner une bibliothèque logicielle. Il s'agit d'un anglicisme fautif dû à un **faux-ami** (library). »*

En C

- Il suffit d'ajouter `#include <bibliotheque.h>`
- Puis, lors de la **compilation** et de l'**édition des liens**,
- on indique à GCC où trouver les **fichiers** nécessaires

Compilation et édition des liens

Rappel sur les **étapes** de compilation:

- **Compilation**: `.c` → `.o`
- **Édition des liens**: `.o` → exécutable

Comment GCC gère-t-il ces deux étapes?

- **Compilation**: trouver les **en-tête** (fichiers `.h`)
- **Édition des liens**: trouver les fichiers **binaires**
- **Plusieurs formats** possibles: `.o`, `.a`, `.so`, `.dll`, ...

Deux syntaxes possibles

- À quel **endroit** GCC cherche ces fichiers?
- `#include "module.h"`: cherche dans le répertoire courant
- `#include <module.h>`: cherche dans le système

Emplacement des fichiers d'en-tête

- À la **compilation**
- GCC cherche seulement les fichiers d'**en-tête** (`.h`)
- Sur une installation **typique Unix**:

```
/usr/local/include
libdir/gcc/target/version/include
/usr/target/include
/usr/include
```

- Si le fichier `.h` est **ailleurs**, il faut le **spécifier**
- À l'aide de l'option `-I` (pour *include*):

```
$ gcc -I<chemin> ...
```

Attention

- Éviter les chemins **absolus** en **dur**
- Sinon le code n'est pas **portable**

Emplacement des binaires

- À l'**édition des liens**
- GCC cherche les implémentations **binaires** (`.o`, `.so`, `.dll`, …)
- Il inspecte **plusieurs répertoires**
- Pour les **connaître** (sur Linux), utiliser `ldconfig -v`:

```
$ ldconfig -v 2>/dev/null | grep -v ^$'\t'
/usr/lib/x86_64-linux-gnu/libfakeroot:
    libfakeroot-0.so -> libfakeroot-tcp.so
/lib/i386-linux-gnu:
    libwrap.so.0 -> libwrap.so.0.7.6
    libnss_dns.so.2 -> libnss_dns-2.27.so
[...]
```

- Si la bibliothèque se trouve ailleurs, il faut le **spécifier**:

```
$ gcc -L<chemin> ...
```

Attention

Ici aussi, éviter les chemins **absolus** en **dur**

L'utilitaire `pkg-config`

- L'utilisation de **chemins absolus en dur** n'est pas acceptable
- Si on souhaite qu'une application soit **portable**
- **Solution**: utiliser le programme PKG-config
- Pour les **inclusions** (`-I`):

```
$ pkg-config --cflags bibliotheque
```

- Pour les **binaires** (`-L` et `-l`):

```
$ pkg-config --libs bibliotheque
```

- Remplacer `bibliotheque` par la bibliothèque correspondante: `cairo`, `tap`, `jansson`, etc.

Retour sur les Makefiles

- Les règles à motifs (*pattern rule*)
- `wildcard`: lister des fichiers (*glob*)
- `patsubst`: substituer des motifs
- `shell`: invoquer une commande shell
- `filter-out`: retire un motif d'une liste de mots
- `realpath`: résoudre un chemin (simplifier et liens symboliques)
- `dir`: semblable à la commande `dirname`
- `abspath`: chemin absolu d'un fichier
- `lastword`: récupère le dernier mot d'une liste de mots
- ...

Voir la documentation officielle pour plus d'informations

Règles à motifs

- Permet de déclarer des règles **générales**
- Le caractère % est utilisé pour indiquer la **substitution**
- $<: première dépendance
- $@: cible

```
%.o: %.c
    gcc -c $(CFLAGS) $< -o $@
```

- On peut **restreindre** les cibles visées
- À l'aide d'une règle **statique**:

```
objects = geometry.o graph.o isomap.o map.o queue.o tile.o

$(objects): %.o: %.c %.h
    gcc -c $(CFLAGS) $< -o $@
```

Fonctions utiles (1/2)

- La fonction `wildcard`:

```
# Tous les fichiers avec extension .c
$(wildcard *.c)
# Tous les fichiers commençant avec test et finissant par .c
$(wildcard test*.c)
```

- La fonction `patsubst`:

```
# Fichiers objets souhaités
$(patsubst %.o, %.c, $(wildcard *.c))
# Exécutables des tests souhaités
$(patsubst %, %.c, $(wildcard test*.c))
```

- La fonction `shell`:

```
# Options de GCC pour la compilation
CFLAGS = "-std=c11 -Wall -Wextra $(shell pkg-config --cflags cairo)
    "
# Options de GCC pour l'édition des liens
LFLAGS = "$(shell pkg-config --libs cairo)"
```

Fonctions utiles (2/2)

- La fonction `filter-out`:

```
# Pour retirer les tests de la liste
test_c_files = $(wildcard test*.c)
c_files = $(filter-out $(test_c_files), $(wildcard *.c))
objects=$(patsubst %.o, %.c, $(c_files))
```

- Les fonctions `realpath`, `dir` et `abspath`:

```
# Récupérer le chemin absolu du répertoire parent d'un Makefile
# $(lastword $(MAKEFILE_LIST)) récupère le nom du Makefile courant
# Ensuite on calcule avec $(abspath ...) le chemin absolu
# Puis $(dir ...) permet de récupérer le répertoire parent
# Et finalement on prend le chemin simplifié avec $(realpath ...)
current_make = $(lastword $(MAKEFILE_LIST))
root_dir := $(realpath $(dir $(abspath $(current_make)))/..)
```

Les branches sous Git

Les branches sous Git

Branches

- `git branch -a`: lister toutes les branches
- `git checkout name`: passer sur la branche `name`
- `git checkout -b name`: créer une branche `name` à partir de `HEAD`
- `git checkout -B name`: même si la branche existe déjà
- `git branch -d`: supprimer une branche fusionnée
- `git branch -D`: supprimer une branche
- `git rebase`: rebaser une branche
- `git rebase -i`: rebasement interactif

Intégration

- `git cherry-pick`: sélectionner un *commit* et l'appliquer
- `git merge`: fusionner deux branches

INF3135
Construction et maintenance de logiciels

Cours 12: Intégration

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Quiz 3

Quiz 3

Matière évaluée

- Date: **21 juillet**
- **Chapitre 5**: structures de données
- **Chapitre 6**: modularité
- **Labo 8**: modules
- **Labo 9**: les branches sous Git

Forme du quiz

- 1 question courte sur chapitre 5
- 1 question courte sur chapitre 6
- 1 question courte sur Git
- 1 question longue

Travail pratique 2

Travail pratique 2

- **20%** de la note finale
- Fait **seul**
- Date de remise: **31 juillet**
- Le travail doit être réparti sur des **branches**
- Attention à la rédaction des **requêtes d'intégration**
- Attention à la qualité des *commits* et de leur message
- Mettre à jour votre .gitconfig
- Ne pas mettre de *commits* sur la branche `master`: utilisée pour se **synchroniser** avec le dépôt parent
- Récupérer les versions les plus à jour

Intégration

Les branches sous Git

- Apporter des modifications à un projet
- Compléter une *issue*
- Faire une requête d'intégration (*merge request*)

INF3135
Construction et maintenance de logiciels

Cours 12: Intégration

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Quiz 3

Quiz 3

Matière évaluée

- Date: **21 juillet**
- **Chapitre 5**: structures de données
- **Chapitre 6**: modularité
- **Labo 8**: modules
- **Labo 9**: les branches sous Git

Forme du quiz

- 1 question courte sur chapitre 5
- 1 question courte sur chapitre 6
- 1 question courte sur Git
- 1 question longue

Travail pratique 2

Travail pratique 2

- **20%** de la note finale
- Fait **seul**
- Date de remise: **31 juillet**
- Le travail doit être réparti sur des **branches**
- Attention à la rédaction des **requêtes d'intégration**
- Attention à la qualité des *commits* et de leur message
- Mettre à jour votre .gitconfig
- Ne pas mettre de *commits* sur la branche `master`: utilisée pour se **synchroniser** avec le dépôt parent
- Récupérer les versions les plus à jour

Intégration

Les branches sous Git

- Apporter des modifications à un projet
- Compléter une *issue*
- Faire une requête d'intégration (*merge request*)

INF3135
Construction et maintenance de logiciels

Cours 13: Tests

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Quiz

Quiz

Quiz 3

- **Moyenne**: 68.7%

Quiz 4

Matière:

- Chapitre 8: tests
- Labo 10: intégration
- Labo 11: tests

Contenu:

- Une question courte sur la gestion de la **mémoire**
- Une question courte sur l'**intégration**
- Une question courte sur les **tests**
- Une question longue sur les **tests**

Travail pratique 3

Travail pratique 3

- **Dépôt**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3
- **Sujet**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3/-/tree/sujet/sujet
- Développer un petit **jeu vidéo**
- Inspiré du jeu Super Hexagon de Terry Cavanagh
- Avec bibliothèque SDL2
- **Coquille** de base disponible
- Peut être fait en **équipe** d'au plus 3
- **Attention!** je vérifie qui fait quoi

Tests

Pourquoi tester?

- Détecter des **bogues**
- Et éventuellement les **corriger**
- Avoir une plus grande **confiance** en notre programme
- S'assurer de ne pas introduire de **régression**

Idéal

- Prouver que notre programme est **sans bogue**
- **Impossible** dans la majorité des cas
- Sauf pour des **petits programmes**
- Ou en utilisant un outil de **vérification formelle**
- **Champ d'étude**: analyse de programme

Problème de l'arrêt (*halting problem*)

Problème

- Peut-on écrire un programme `G` qui
- Étant donné un **programme** `P`
- Décide si `P` **termine toujours**
- Peu importe les valeurs en **entrée**?

Réponse

- **1936**: Turing a prouvé qu'un tel programme `G` ne peut pas exister
- Plus formellement, le problème est **indécidable** (Gödel)

Conséquence

Impossible de **prouver** qu'un programme arbitraire est **sans bogue**

Exemple: collatz

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

void print_collatz_sequence(unsigned int n) {
    while (true) {
        printf("%d ", n);
        if (n == 1)
            break;
        else if (n % 2 == 0)
            n = n / 2;
        else
            n = 3 * n + 1;
    }
}

int main(int argc, char *argv[]) {
    print_collatz_sequence(atoi(argv[1]));
    return 0;
}
```

Résultat:

```
$ gcc collatz.c -o collatz
$ ./collatz 85
85 256 128 64 32 16 8 4 2 1
```

Différents types de tests (1/3)

Compilation (*build*)

- **Compilation** correcte
- Édition des **liens**
- Avec **bibliothèques** tierces
- Ou autres **dépendances**

Intégration

- Interaction correcte **entre** les modules
- Autant **compilation**
- Que tests **unitaires**

Différents types de tests (2/3)

Unitaires

- Teste un aspect **spécifique**
- De façon **individuelle**

Fonctionnels

- **Comportement** du programme
- Est en adéquation avec ce qui était **demandé**
- Respecte le **cahier des charges**

Non-régression

- Pas de perte de **fonctionnement**
- Ou de perte importante de **performance**

Différents types de tests (3/3)

Configuration

- Fonctionne dans des **environnements** variés
- **Appareils** (ordinateur, mobile, console, ...)
- **Distribution** (Linux, MacOS, Windows, ...)
- **Architecture** (32 bits, 64 bits, autre processeur, ...)

Performance

- **Rapidité** du programme
- Utilisation de **mémoire**

Installation

L'application s'installe correctement

Propriétés d'un bon test

- **Juste**: il teste bien ce qu'il faut
- **Robuste**: il ne plante pas
- **Pur**: il est sans effet de bord
- **Reproductible**: il a le même comportement peu importe l'environnement
- **Pertinent**: il augmente notre confiance
- **Non redondant**: il teste quelque chose de distinct d'un autre test
- **Efficace**: il prend un temps raisonnable
- **Automatisable**: il ne nécessite pas d'intervention humaine

Automatisation

Tests automatisés

- Vérification **textuelle**, notamment à l'aide de *regex*
- Utilisation des **canaux standards** (`stdin`, `stdout`, `stderr`)
- Ou accès direct au **contenu** (tests internes)
- Construction de **scénarios**

Tests manuels

- Quand automatisation **pas possible**
- Vérification **humaine**
- Évaluation **visuelle**, **sonore**, ...
- Suite d'**événements**, parfois **asynchrones**
- Prennent du **temps** et souvent **coûteux**
- Typique des applications **graphiques**
- Tests d'**interface d'utilisation**

Tests shell

- De nombreuses **commandes shell** permettent de tester
- Avec la **sémantique** habituelle (0: succès, $\neq$0: erreur)

Exemples:

```
# Vérifie si README.md contient un code permanent
#    -q: mode silencieux
#    -E: expression étendue
$ grep -qE "[A-Z]{4}[0-9]{8}" README.md

# Vérifie si bidon.c existe dans un sous-dossier de /tmp
$ find /tmp -name bidon.c | grep -q .

# Vérifie si fichier1 et fichier 2 sont presque identiques
#    -i: ignorer la casse
#    -w: ignorer les espaces
$ diff -iw fichier1 fichier2

# Vérifie si une commande engendre une fuite mémoire
# Valgrind retourne 0 si aucune fuite, 1 sinon
$ valgrind --leak-check=yes --error-exitcode=1 ./prog; echo $?
```

La commande `test`

Vérifier le type des fichiers et compare des valeurs:

```
test EXPRESSION [OPTION]
```

- Si l'expression est **vraie** alors la commande retourne 0
- Sinon elle retourne 1

Exemples:

```
# Vérifie si 1 < 2 (comparaison numérique)
$ test 1 -lt 2; echo $?
0
# Vérifie si linux = linux (en tant que chaînes)
$ test $(echo "linux") = "linux"; echo $?
0
# Vérifie s'il y a un Makefile dans le répertoire courant
$ test -f Makefile; echo $?
0
# Vérifie s'il y a un répertoire code dans le répertoire courant
$ test -d code; echo $?
0
```

Tests sur chaînes de caractères

```
test CHAINE1 OPERATEUR CHAINE2
```

Exemples:

```
# Vérifie si deux chaînes sont égales
$ test "linux" = "Linux"; echo $?
1
# Vérifie si deux chaînes sont différentes
$ test "linux" != "Linux"; echo $?
0
# Vérifie si une chaîne est vide
$ test -z ""; echo $?
0
$ test -z "linux"; echo $?
1
# Vérifie si une chaîne est non vide
$ test -n ""; echo $?
1
$ test -n "linux"; echo $?
0
```

Tests sur les valeurs numériques

```
test VALEUR1 OPERATEUR VALEUR2
```

Exemples:

```
# Vérifie si deux valeurs sont égales
$ test 1 -eq 1
# Vérifie si deux valeurs sont différentes
$ test 1 -ne 2
# Vérifie une inégalité
$ test 1 -lt 2
$ test 2 -le 2
$ test 2 -gt 1
$ test 1 -ge 1
# Attention à différencier `=` de `-eq`
$ test "01" = 1; echo $?
1
$ test "01" -eq 1; echo $?
0
```

Tests sur les fichiers

```
test OPTION CHEMIN
```

Exemples:

```
# Est-ce que le chemin existe?
$ test -e /usr/local/bin; echo $?
0
# Est-ce que le chemin est un fichier?
$ test -f /usr/local/bin/bats; echo $?
0
# Est-ce que le chemin est un répertoire?
$ test -f /usr/local/bin; echo $?
0
# Est-ce que le chemin est un fichier non vide?
$ touch nouveau
$ test -s nouveau; echo $?
1
# Est-ce que le chemin est accessible en lecture?
$ test -r chemin
# Est-ce que le chemin est accessible en écriture?
$ test -w chemin
# Est-ce que le chemin est exécutable?
$ test -x chemin
```

Opérateurs logiques

```
test EXPRESSION1 OPERATEUR EXPRESSION2
```

Exemples:

```
# ET logique
$ test expr1 -a expr2
# OU logique
$ test expr1 -o expr2
# NON logique
$ test ! expr
# Retourne vrai si chemin est un fichier vide
test -f chemin -a ! -s chemin
```

Syntaxe allégée

La syntaxe

```
test EXPRESSION
```

est équivalente à

```
[ EXPRESSION ]
```

Exemples:

```
$ [ -f Makefile ]; echo $?
1
$ [ -d bin ]; echo $?
0
```

- Les **espaces** après [et avant] sont importants
- Le caractère] est optionnel (mais plus joli)

Tests Bash

Syntaxe avec **doubles crochets**:

```
[[ EXPRESSION ]]
```

Opérateurs possibles:

- && et ||: connecteur logiques ET et OU
- (et): pour parenthéser
- < et >: comparaison lexicographique de chaînes
- ==: la 2e opérande est un motif de type *glob*
- =~: la 2e opérande est une expression régulière étendue

```
# La première expression correspond avec b = ? et * = njour
# La deuxième aussi avec onjou
#    [un]       alternative entre u et n
#    j?         j optionnel
#    (...){2}   motif répété exactement deux fois
$ [[ bonjour == ?o* && bonjour =~ (o[un]j?){2} ]]
$ echo $?
0
```

Bats

- *Bats = Bash Automated Testing System*
- **Lien**: `https://github.com/bats-core/bats-core`
- **Image**: DockerHub

  *« Bats is a TAP-compliant testing framework for **Bash**. It provides a simple way to verify that the UNIX programs you write behave as expected.*
  *A Bats test file is a Bash script with special syntax for defining test cases. Under the hood, each test case is **just a function with a description**. »*

- Autrement dit, il suffit d'utiliser des **commandes** de test
- Et on peut profiter des **expressions régulières**

Exemples de tests Bats

```
valgrind_options="--error-exitcode=1 --leak-check=full"

# On vérifie s'il y a une fuite mémoire
@test "No leak with default program" {
    run valgrind $valgrind_options ./program
    [ "$status" -eq 0 ]
}

# On vérifie que le fichier diagram.dot est valide
@test "File diagram.dot is valid" {
    run neato diagram.dot
    [ "$status" -eq 0 ]
}

# On vérifie les premières lignes
@test "Show help with -h" {
    run ./program -h
    [ "$status" -eq 0 ]
    [ "${lines[0]}" = "Usage: ./program [-h|--help]" ]
    [[ "${lines[0]}" == "[Uu]sage" ]]
    [[ "${lines[0]}" =~ "h(elp)?" ]]
}
```

Exemple plus complexe (1/2)

- Programme `tournament.c` qui **génère** une grille de tournoi
- En lisant sur l'**entrée standard**
- Chaque ligne correspond à un **joueur** ou une **équipe**
- On veut **limiter à 20** pour des raisons d'affichage

```
$ cat examples/tennis.in
Djokovic
Nadal
Federer
Murray
$ ./tournament -s table < examples/tennis.in
ID  Player    Day 1   Day 2   Day 3
--  --------  ------  ------  ------
 1  Djokovic  4       2       3
 2  Nadal     3       1       4
 3  Federer   2       4       1
 4  Murray    1       3       2
```

Exemple plus complexe (2/2)

```
# Vérifier l'affichage à espace près
#     -Z   ignorer les espaces en fin de ligne
#     -B   ignorer les lignes vides
@test "Tennis example with default options" {
    run diff -ZB examples/tennis-default.out \
                 <(./tournament < examples/tennis.in)
    [ "$status" -eq 0 ]
}

# Détecter le cas où on a plus de 20 équipes
@test "Too many players" {
    run ./tournament < examples/soccer-long.in
    [[ "$output" =~ "Error.*too many players" ]]
    [ "$status" -eq 1 ]
}

# Permettre le cas où on a exactement 20 équipes
# On ne peut pas utiliser | en combinaison avec run
@test "Twenty players is ok" {
    head -n 20 examples/soccer-long.in | ./tournament
    [ "$?" -eq 0 ]
}
```

Développement guidé par les tests

3 lois

Extraites de Wikipedia:

1. Vous devez écrire un test qui échoue avant de pouvoir écrire le code de production correspondant.
2. Vous devez écrire une seule assertion à la fois, qui fait échouer le test ou qui échoue à la compilation.
3. Vous devez écrire le minimum de code de production pour que l'assertion du test actuellement en échec soit satisfaite.

Proposée par **Robert C. Martin** (2014)

Cycle de développement en 5 étapes

1. Ajouter un test ou plusieurs tests

Qui mettent en évidence le comportement souhaité

2. Lancer tous les tests

Les nouveaux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les nouveaux tests devraient réussir, les anciens aussi

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

Exemple: le module set (1/4)

Interface (fichier `set.h`):

```
struct set {                    // A set of integers
    int *elements;              // Its elements
    unsigned int size;          // Its cardinality
    unsigned int capacity; // Its capacity
};

// Create an empty set
struct set *set_create(void);
// Delete a set
void set_delete(struct set *set);
// Check if a set is empty
bool set_is_empty(const struct set *set);
// Check if a set contains an element
bool set_contains(const struct set *set, int element);
// Add an element to a set
void set_add(struct set *set, int element);
// Print a set to stdout
void set_print(const struct set *set);
```

Exemple: le module set (2/4)

Implémentation (fichier `set.c`):

```
struct set *set_create(void) {
    struct set *set = malloc(sizeof(struct set));
    set->elements = malloc(sizeof(int));
    set->capacity = 1;
    set->size = 0;
    return set;
}

void set_delete(struct set *set) {
    free(set->elements);
    free(set);
}

bool set_is_empty(const struct set *set) {
    return set->size == 0;
}

void set_print(const struct set *set) {
    printf("{");
    for (unsigned int i = 0; i < set->size; ++i) {
        if (i > 0) printf(", ");
        printf("%d", set->elements[i]);
    }
    printf("}");
}
```

Exemple: le module set (3/4)

Implémentation (fichier `set.c`):

```
int compare_ints(const void *i1, const void *i2) {
    return *(int*)i1 - *(int*)i2;
}

bool set_contains(const struct set *set,
                  int element) {
    return bsearch(&element, set->elements, set->size,
                   sizeof(int), compare_ints) != NULL;
}

void set_add(struct set *set,
             int element) {
    unsigned int i = 0;
    while (i < set->size && set->elements[i] < element)
        ++i;
    if (set->elements[i] == element) return;
    if (set->size == set->capacity) {
        set->capacity *= 2;
        set->elements = realloc(set->elements, set->capacity * sizeof(int));
    }
    for (unsigned int j = set->size; j > i; --j)
        set->elements[j] = set->elements[j - 1];
    set->elements[i] = element;
    ++set->size;
}
```

Exemple: le module set (4/4)

Tests (fichier `test_set.c`):

```c
#include "set.h"
#include <tap.h>

int main(void) {
    struct set *set = set_create();
    ok(set_is_empty(set), "created set is empty");
    ok(set->size == 0, "size of set is 0");
    set_add(set, 3);
    set_add(set, 5);
    set_add(set, 2);
    diag("Adding 3, 5, 2");
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is 3");
    ok(set_contains(set, 3), "set contains 3");
    ok(!set_contains(set, 1), "set does not contains 1");
    diag("Adding 5 again");
    set_add(set, 5);
    printf("# set = "); set_print(set); printf("\n");
    ok(set->size == 3, "size of set is still 3");
    set_delete(set);
    return 0;
}
```

Ajout de la fonction de suppression d'un élément

1. Ajouter un test ou plusieurs tests

Deux cas: élément **présent** ou **absent**

2. Lancer tous les tests

Les deux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les deux nouveaux tests et les anciens devraient réussir

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

INF3135
Construction et maintenance de logiciels

Cours 14: Développement continu

Alexandre Blondin Massé

Université du Québec à Montréal
Département d'informatique

Été 2020

Table des matières

Quiz 4

Quiz 4

Matière

- Chapitre 8: tests
- Labo 10: intégration
- Labo 11: tests

Contenu

- Une question courte sur la gestion de la **mémoire**
- Une question courte sur l'**intégration**
- Une question courte sur les **tests**
- Une question longue sur les **tests**

Travail pratique 3

Travail pratique 3

- **Dépôt**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3
- **Sujet**: https://gitlab.info.uqam.ca/inf3135-ete2020/inf3135-ete2020-tp3/-/tree/sujet/sujet
- Développer un petit **jeu vidéo**
- Inspiré du jeu Super Hexagon de Terry Cavanagh
- Avec bibliothèque SDL2
- **Coquille** de base disponible
- Peut être fait en **équipe** d'au plus 3
- **Attention!** je vérifie qui fait quoi

Tests

Différents types de tests

- **Compilation**: incluant bibliothèques tierces
- **Intégration**: communication intermodulaire
- **Unitaires**: teste un aspect spécifique
- **Fonctionnels**: en adéquation avec le cahier des charges
- **Non-régression**: pas de perte de fonctionnement ou d'efficacité
- **Configuration**: fonctionne dans des environnements variés
- **Performance**: consomme pas trop de temps ou de mémoire
- **Installation**: s'installe correctement sur un système

Propriétés d'un bon test

- **Juste**: il teste bien ce qu'il faut
- **Robuste**: il ne plante pas
- **Pur**: il est sans effet de bord
- **Reproductible**: il a le même comportement peu importe l'environnement
- **Pertinent**: il augmente notre confiance
- **Non redondant**: il teste quelque chose de distinct d'un autre test
- **Efficace**: il prend un temps raisonnable
- **Automatisable**: il ne nécessite pas d'intervention humaine

Tests shell

- De nombreuses **commandes shell** permettent de tester
- Avec la **sémantique** habituelle (0: succès, $\neq$0: erreur)

Commandes utiles

- `grep`: recherche un motif
- `diff`: compare le contenu de deux fichiers
- `valgrind`: teste la gestion de la mémoire
- `test EXPR`: teste sur des fichiers et sur des valeurs
- $\rightarrow$ équivalent à `[ EXPR ]`
- $\rightarrow$ extension Bash `[[ EXPR ]]`

Tests internes/externes

Internes

- Basés sur l'**implémentation**
- Notamment des **structures** utilisées (répétitives, conditionnelles)
- **Cas particulier**: assertions (avec `assert.h`)
- **Outils**: graphe de flux, graphe d'appels de fonctions, ...
- **Bibliothèque** spécifique au langage, par exemple Libtap

Externes

- Basés sur l'**interface**
- Permet de valider la **documentation**
- **2 niveaux**: module et application
- **Outils**:
- module $\rightarrow$ bibliothèques
- application $\rightarrow$ le shell, Bats

Développement guidé par les tests

1. Ajouter un test ou plusieurs tests

Qui mettent en évidence le comportement souhaité

2. Lancer tous les tests

Les nouveaux tests devraient échouer

3. Écrire du code

Pas besoin d'être parfait

4. Lancer tous les tests

Les nouveaux tests devraient réussir, les anciens aussi

5. Factorisation et nettoyage

En s'assurant que les tests réussissent toujours

Développement continu

Quelques définitions (tirées de Wikipedia)

« **L'intégration continue** est un ensemble de pratiques utilisées en génie logiciel consistant à vérifier à chaque modification de code source que le résultat des modifications ne produit pas de régression dans l'application développée »

« La **livraison continue** est une approche d'ingénierie logicielle dans laquelle les équipes produisent des logiciels dans des cycles courts, ce qui permet de le mettre à disposition à n'importe quel moment »

« Le **déploiement continu**, est une approche d'ingénierie logicielle dans laquelle les fonctionnalités logicielles sont livrées fréquemment par le biais de déploiements automatisés. »

« Le **devops** ou **DevOps** est un mouvement en ingénierie informatique et une pratique technique visant à l'unification du développement logiciel (dev) et de l'administration des infrastructures informatiques (ops), notamment l'administration système. »

Chaîne DevOps

- **Planifier** (*plan*): identifier besoins/obstacles, définir politiques
- **Créer** (*create*): coder, programmer, compiler
- **Vérifier** (*verify*): tester à différents niveaux
- **Empaqueter** (*package*): préparer la livraison, dépendances
- **Livrer** (*release*): coordination des livraisons, déploiement
- **Configurer** (*configure*): stockage, infrastructure, environnement
- **Surveiller** (*monitor*): mesures diverses, catalogue des bogues

Outils

Jira, Trello, Github, Bitbucket, GitLab, Jenkins, Travis-CI, Autotools, CMake, GitLab-CI, Docker, ...

Suite

- **GitLab-CI**: développement continu
- **Docker**: manipulation de conteneurs

GitLab-CI

GitLab-CI

- Permet de mettre en place des **pipelines** de traitement
- **Lien**: `https://docs.gitlab.com/ee/ci/`
- **Gratuit** jusqu'à 2000 minutes par mois sur GitLab.com

Extrait de la page d'accueil:

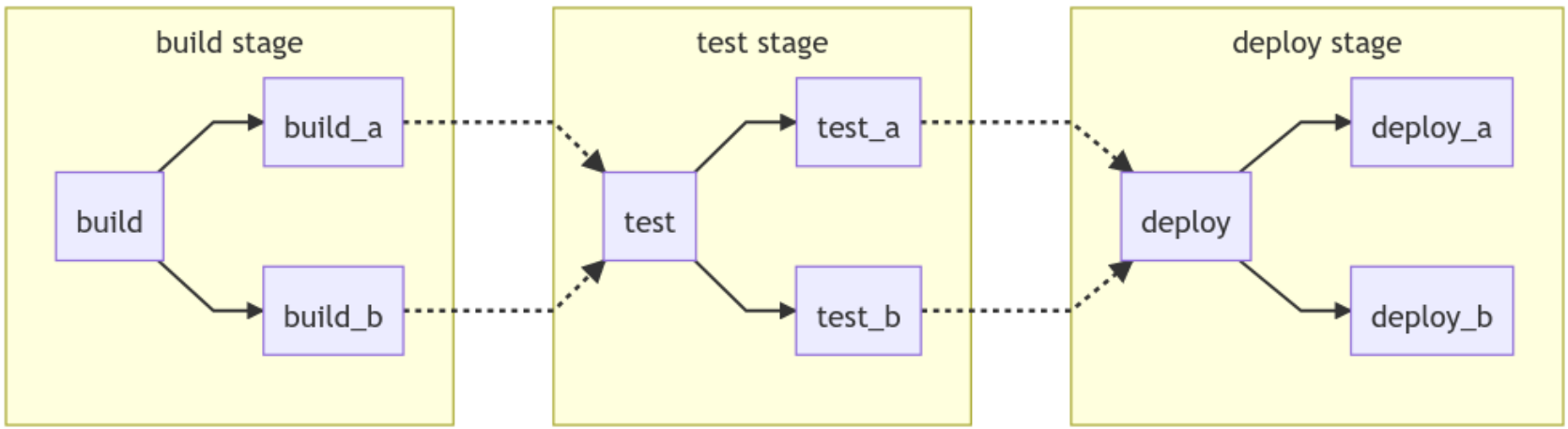
« GitLab CI/CD is a tool built into GitLab for software development through the continuous methodologies:

- Continuous Integration (CI)
- Continuous Delivery (CD)
- Continuous Deployment (CD) »

Gère autant l'**intégration** que la **livraison** et le **déploiement**

Pipeline

- **Ensemble** de tâches à compléter
- Structuré sous forme de **graphe acyclique**
- Possible aussi de relations **hiérarchiques** (parent/enfant)
- Voir architectures de pipeline pour plus d'informations
- 3 étapes (*stages*) par défaut: `build`, `test`, `deploy`
- 2 résultats possibles pour chaque tâche: succès (0) ou échec ($\neq$0)
- Tâches d'une même étape exécutées en **parallèle**
- Tâches d'une étape suivante lancées seulement si **toutes** les tâches de l'étape précédente ont réussi



Mise en place (1/2)

- À l'aide d'un fichier de **configuration** nommé `.gitlab-ci.yml`
- Qui doit respecter la syntaxe YAML
- **Exemple**:

```yaml
stages:
  - build
  - test
  - deploy

image: alpine

build_a:
  stage: build
  script:
    - echo "This job builds something."

build_b:
  stage: build
  script:
    - echo "This job builds something else."

test_a:
  stage: test
  script:
    - echo "This job tests something. It will only run when all jobs in the"
    - echo "build stage are complete."
```

Mise en place (2/2)

```
test_b:
  stage: test
  script:
    - echo "This job tests something else. It will only run when all jobs in"
    - echo "the build stage are complete too. It will start at about the"
    - echo "same time as test_a."

deploy_a:
  stage: deploy
  script:
    - echo "This job deploys something. It will only run when all jobs in"
    - echo "the test stage complete."

deploy_b:
  stage: deploy
  script:
    - echo "This job deploys something else. It will only run when all jobs"
    - echo "in the test stage complete. It will start at about the same time"
    - echo "as deploy_a."
```

- Déclarer des **dépendances** plus fines: `needs`
- **Hiérarchiser** des pipelines: `trigger`
- Exécuter un script **avant** ou **après** chaque tâche: `before_script` et `after_script`, ...

Artéfacts

- **Fichier** ou **répertoire** produit lors d'une tâche
- Permet aux tâches de **communiquer** entre elles
- Voir page sur artéfacts pour plus d'informations
- **Exemple**:

```
test:
    stage: test
    script:
        - make test
    artifacts:
        paths:
            - ./tests/isomap.png
        expire_in: 1 week
```

- Télécharger tous les artéfacts ou seulement le fichier

```
# URL de l'archive
https://example.com/<namespace>/<project>/-/jobs/artifacts/<ref>/download?job
    =<job_name>
# URL du fichier
https://example.com/<namespace>/<project>/-/jobs/artifacts/<ref>/raw/<
    path_to_file>?job=<job_name>
```

Image

- Les tâches sont exécutées dans un « environnement » précis
- On appelle cet « environnement » une **image**
- Spécifié à l'aide du mot-clé `image`
- Par défaut, l'image est `ruby:2.1`
- Par défaut, les images sont récupérées sur Docker Hub

Registre de conteneurs

- GitLab supporte aussi un registre de conteneurs
- En anglais, *container registry*
- Permet d'héberger des images par **projet**
- Plutôt que de mettre des images **publiques** sur Docker Hub

Docker

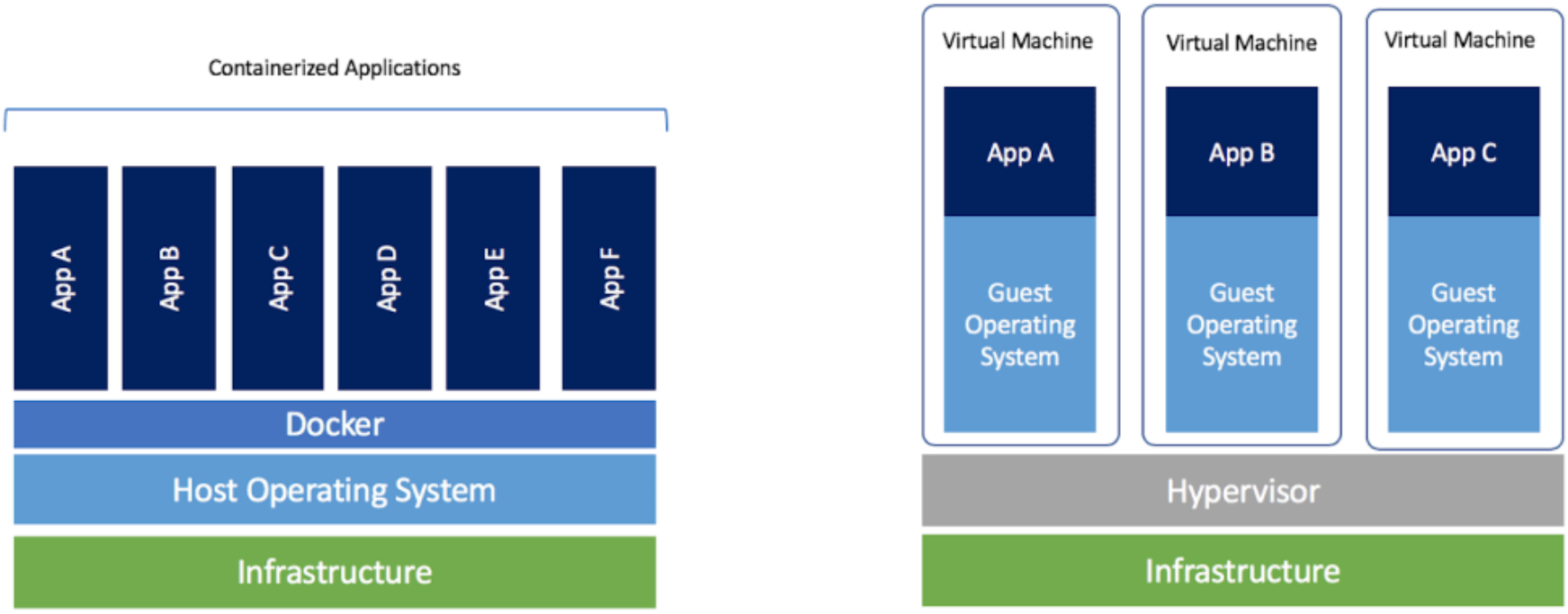
Docker

- Site officiel: `https://www.docker.com/`
- Permet de manipuler des **conteneurs**

Extrait de docker.com

« A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries and settings. »

Comparaison entre conteneur et machine virtuelle (1/2)



(Source: blog de Jenny Fong)

Comparaison entre conteneur et machine virtuelle (2/2)

Machine virtuelle

- Virtualisation au niveau **matériel**
- Plus **lourd**
- Accès plus **lent**
- Performance **limitée**
- Isolation **complète** et donc plus **sécuritaire**

Conteneurs

- Virtualisation au niveau **logiciel**
- Plus **léger**
- Accès plus **rapide** et meilleur **passage à l'échelle**
- Performance **native**
- Isolation au niveau des **processus** et donc moins **sécuritaire**

Installation

- **Dépend** de la distribution
- **Debian** et dérivée (comme Ubuntu, Mint):

```
# Mise à jour des paquets
$ sudo apt update
# Optionnel: déinstaller anciennes versions
$ sudo apt remove docker docker-engine docker.io containerd runc
# Installation
$ sudo apt install docker.io
$ sudo systemctl start docker.
# Ou: sudo service docker start.
```

- **MacOS**: Docker Desktop pour MacOS
- **Windows**: Docker Desktop pour Windows
- Un peu de configuration à faire pour utiliser en **ligne de commande**

Quelques opérations

- run: **lancer** une commande dans un conteneur
- image: **gérer** les images
- images: **lister** les images
- build: **construire** une image
- login: se **connecter** à un registre
- logout: se **déconnecter** d'un registre
- pull: **télécharger** une image
- push: **téléverser** une image
- volume: **gérer** les volumes

Voir documentation complète pour plus d'informations

Dockerfile (1/2)

- Décrit comment **construire** des images
- Essentiellement, c'est un **script shell**
- Mais avec plus de **fonctionnalités**:

Quelques commandes

- `FROM`: précise l'image de base
- `RUN`: exécute une commande
- `EXPOSE`: expose un port
- `ENV`: assigne une variable d'environnement
- `WORKDIR`: change le répertoire courant
- `VOLUME`: monte un volume, ...

Dockerfile (2/2)

```
# Précise l'image de départ
FROM ubuntu:18.04

# Installe des paquets
RUN apt-get update && apt-get install -y \
    build-essential \
    git \
    graphviz \
    libcairo-dev \
    libjansson-dev \
    pandoc \
    valgrind \
    pkg-config

# Installe manuellement Libtap
RUN git clone https://github.com/zorgnax/libtap.git && \
    cd libtap && \
    make && \
    make install && \
    ldconfig

# Installe manuellement Bats
WORKDIR /Applications
RUN git clone https://github.com/bats-core/bats-core.git bats-core && \
    bash bats-core/install.sh .

# Pour trouver Bats de n'importe où
ENV PATH="${PATH}:/Applications/bin"
```