LES BIZARRERIES DU C

Ou **comment faire du C dégueulasse** ?

```
#include                                <math.h>
#include                               <sys/time.h>
#include                               <X11/Xlib.h>
#include                              <X11/keysym.h>
                                       double L ,o ,P
                                       ,_=dt,T,Z,D=1,d,
                                       s[999],E,h= 8,I,
                                       J,K,W[999],M,m,O
                                      ,n[999],j=33e-3,i=
                                      1E3,r,t, u,v ,w,S=
                                      74.5,l=221,X=7.26,
                                      a,B,A=32.2,c, F,H;
                                      int N,q, C, y,p,U;
                                     Window z; char f[52]
                                  ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
 ; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
 0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
 ; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
 sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
 *T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
 ]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
  *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
   XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                    XEvent z; XNextEvent(e ,&z);
                                        ++*((N=XLookupKeysym
                                         (&z.xkey,0))-IT?
                                         N-LT? UP-N?& E:&
                                         J:& u: &h); --*(
                                         DN -N? N-DT ?N==
                                         RT?&u: & W:&h:&J
                                          ); } m=15*F/l;
                                          c+=(I=M/ l,l*H
                                          +I*M+a*X)*_; H
                                          =A*r+v*X-F*l+(
                                          E=.1+X*4.9/l,t
                                          =T*m/32-I*T/24
                                           )/S; K=F*M+(
                                           h* 1e4/l-(T+
                                           E*5*T*E)/3e2
                                           )/S-X*d-B*A;
                                           a=2.63 /l*d;
                                           X+=( d*l-T/S
                                            *(.19*E +a
                                            *.64+J/1e3
                                            )-M* v +A*
                                            Z)*_; l +=
                                            K *_; W=d;
                                            sprintf(f,
                                            "%5d  %3d"
                                            "%7d",p =l
                                           /1.7,(C=9E3+
                                  O*57.3)%6550,(int)i); d+=T*(.45-14/l*
                                 X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                                 *I-m* 52+E*94 *D-t*.38+u*.21*E) /1e2+W*
                                 179*v)/2312; select(p=0,0,0,0,&G); v-=(
                                  W*F-T*(.63*m-I*.006*m*E*19-D*25-.11*u
                                   )/107e2)*_; D=cos(o); E=sin(o); } }
```

Thomas Sayen

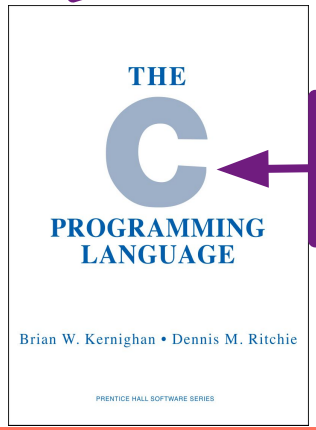
HISTOIRE SIMPLIFIÉE DU C

1972

Création du C par Dennis Ritchie

1978 - K&R

Spécification non-officielle du C. Livre coécrit par Ritchie et Brian Kernighan, d'où K&R.



1983 → 1989 - ANSI & ISO (1990)

Standardisation du C, le C89 (aussi appelé ANSI C) devient la première norme officielle.

1999

C99 : 2e norme du C.
C'est le standard le plus utilisé, car supporté par ~100% des machines et ajouts depuis C89.

2011 (C11) - 2018 (C17 ou C18)

Changements et ajouts mineurs.

2023 (C23)

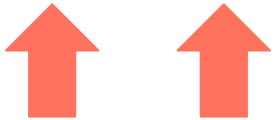
Modernisation notable du langage (suppression de vieilleries) et emprunts au C++ (nullptr / constexpr), expérimental à ce jour (février 2025)

Article de blog sur le K&R

QUELLE VERSION ?

À ce jour (mars 2025) :

C89	C99	C11	C17	C23
Extrêmement portable	Extrêmement portable + nombreux ajouts	Ajout (entre autres) de multithreading	Correction de bugs	Plus moderne mais expérimental



Par défaut
avec GCC 14.2.0
et Clang 19.1.0

Cheat sheet C11→C23

Comment détecter avec quelle version du C notre code est compilé ?

IBM I supporte C99 maximum (§ Industry Standards)

Le C23 étant encore expérimental, GCC 14.2.0
n'a pas encore mis __STDC_VERSION__ à jour.

EXEMPLES UTILES

LES BOOLÉENS

C89 :
→ pas de type booléen !
→ int souvent utilisé

```c
int main(void) {
    int status;
}
```

```c
int main(void) {
    int status = 1;
}
```

Depuis C99 : §7.16
→bool, true, false = macros
→_Bool = mot-clef (§A.1.2)

```c
int main(void) {
    _Bool status;
}
```

```c
#include <stdbool.h>

int main(void) {
    _Bool status = true;
}
```

```c
#include <stdbool.h>

int main(void) {
    bool status;
}
```

```c
#include <stdbool.h>

int main(void) {
    bool status = true;
}
```

Depuis C23 : §6.4.2
→bool, true, false = mot-clefs
→_Bool = mot-clef

```c
int main(void) {
    bool status;
}
```

```c
int main(void) {
    bool status = true;
}
```

INITIALISER LE N-IÈME ÉLÉMENT

```c
int main(void) {
    int array[10] = { 0 };
}
```

index	0	1	2	3	4	5	6	7	8	9
array	0	0	0	0	0	0	0	0	0	0

```c
int main(void) {
    int array[10] = { [4] = 15 };
}
```

index	0	1	2	3	4	5	6	7	8	9
array	0	0	0	0	15	0	0	0	0	0

```c
int main(void) {
    int array[10] = { [4] = 15, [7] = -283 };
}
```

index	0	1	2	3	4	5	6	7	8	9
array	0	0	0	0	15	0	0	-283	0	0

C99 §6.7.8/6

UN MAP/DICTIONNAIRE EN C99

```python
CAVALIER = 0
FOU = 1
PION = 2
REINE = 3
ROI = 4
TOUR = 5

VALEUR_PIECE = {
    CAVALIER: 3,
    FOU: 3,
    PION: 1,
    REINE: 9,
    ROI: 100,
    TOUR: 5
}
```

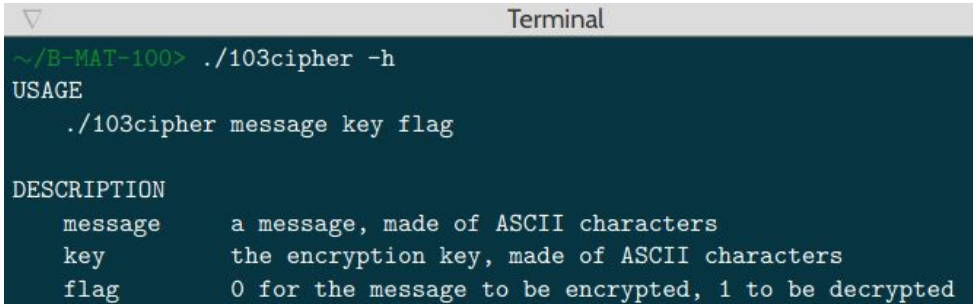
Python 3
(dictionnaire)

```c
enum piece_echecs {
    CAVALIER,
    FOU,
    PION,
    REINE,
    ROI,
    TOUR,
    PIECE_ECHECS_MAX
};

int valeur_piece[PIECE_ECHECS_MAX] = {
    [CAVALIER] = 3,
    [FOU] = 3,
    [PION] = 1,
    [REINE] = 9,
    [ROI] = 100,
    [TOUR] = 5
};

int main(void) {
    return valeur_piece[CAVALIER];
}
```

C99

--HELP

```c
#include <stdio.h>

int main(void) {
    puts("USAGE");
    puts("\t./103cipher message key flag");
    puts("");
    puts("DESCRIPTION");
    puts("\tmessage a message, made of ASCII characters");
    puts("\tkey the encryption key, made of ASCII characters");
    puts("\tflag 0 for the message to be encrypted, 1 to be decrypted");
}
```



--HELP

```c
#include <stdio.h>

int main(void) {
    puts(
        "USAGE\n"
        "\t./103cipher message key flag\n"
        "\n"
        "DESCRIPTION\n"
        "\tmessage a message, made of ASCII characters\n"
        "\tkey the encryption key, made of ASCII characters\n"
        "\tflag 0 for the message to be encrypted, 1 to be decrypted"
    );
}
```

Terminal

```
~/B-MAT-100> ./103cipher -h
USAGE
    ./103cipher message key flag

DESCRIPTION
    message     a message, made of ASCII characters
    key         the encryption key, made of ASCII characters
    flag        0 for the message to be encrypted, 1 to be decrypted
```

C89 §6.1.4 (Example)
C99 §6.4.5/7

--HELP (C++)

```cpp
#include <iostream>

int main() {
    std::cout <<
R"(USAGE
    ./103cipher message key flag

DESCRIPTION
    message a message, made of ASCII characters
    key the encryption key, made of ASCII characters
    flag 0 for the message to be encrypted, 1 to be decrypted
)";

}
```

Terminal

```
~/B-MAT-100> ./103cipher -h
USAGE
    ./103cipher message key flag

DESCRIPTION
    message     a message, made of ASCII characters
    key         the encryption key, made of ASCII characters
    flag        0 for the message to be encrypted, 1 to be decrypted
```

Cppreference, section “Raw string literal”

#include <inttypes.h>

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

int main(void) {
    uint32_t n;
    scanf("%" SCNu32, &n);
    printf("%" PRIu32 "\n", n);
}
```

“%” “u” → “%u”

uint32_t = unsigned char ? ⇒ %hhu
= unsigned short ? ⇒ %hu
= unsigned ? ⇒ %u
= unsigned long ? ⇒ %lu
= unsigned long long ? ⇒ %llu

PRIu32 / SCNu32 → bon flag pour printf/scanf

C99 §7.8.1

UN PETIT PARSEUR

```c
#include <stdbool.h>
#include <stddef.h> // NULL

typedef struct argv_parser {
    bool help;
    bool verbose;
    const char* input_file;
    const char* output_file;
} argv_parser_t;

int main(int argc, char* argv[]) {
    argv_parser_t parser;
    parser.help = false;
    parser.verbose = false;
    parser.input_file = NULL;
    parser.output_file = NULL;
}
```

./a.out [-h] [-v] input [-o output]

Structure non initialisée

Puis on initialise chaque champ

UN PETIT PARSEUR

```c
int main(int argc, char* argv[]) {
    argv_parser_t parser = {
        false,
        false,
        NULL,
        NULL
    };
}
```

On initialise tout directement, mais peu lisible (quel false correspond à quoi ?)

```c
int main(int argc, char* argv[]) {
    argv_parser_t parser = {
        .help = false,
        .verbose = false,
        .input_file = NULL,
        .output_file = NULL
    };
}
```

On initialise tout directement **et** c'est lisible, on écrit le nom de chaque champ ainsi que sa valeur.

C99 §6.7.8/7 (designated initializer)

MACROS & IDENTIFICATEURS PRÉDÉFINIS

```c
#include <stdio.h>

int main(void) {
    printf(
        "Fichier %s compilé le %s à %s\n",
        __FILE__,
        __DATE__,
        __TIME__
    );
    printf(
        "Fonction %s à la ligne %d : %s\n",
        __func__,
        __LINE__,
        "Bonjour"
    );
}
```

__FILE__ (macro) :
→ chaîne de caractères littérale (entre guillemets “”) contenant le nom du fichier

__DATE__* & __TIME__* (macros) :
→ date et heure auxquelles a été compilé le fichier

__func__ (const char []):
→ nom de la fonction dans laquelle on se trouve

__LINE__ (macro) :
→ numéro de la ligne à laquelle on se trouve

C89 §6.8.8 // C99 §6.10.8 (macros) C99 §6.4.2.2 (__func__ uniquement) * __DATE__, __TIME__ et les locales

FONCTION DE LOG

```c
#include <stdio.h>

void log_msg(const char* msg) {
    printf("LOG dans la fonction %s à la ligne %d : %s\n", __func__, __LINE__, msg);
}

int main(void) {
    log_msg("Bonjour");
}
```

On appelle la fonction et on s'y déplace

LOG dans la fonction **log_msg** à la ligne **4** : Bonjour

PLUTÔT UNE MACRO DE LOG...

```c
#include <stdio.h>

#define log_msg(msg) printf("LOG dans la fonction %s à la ligne %d : %s\n", __func__, __LINE__, msg);

int main(void) {
    log_msg("Bonjour");
}
```

LOG dans la fonction **main** à la ligne **6** : Bonjour

Le préprocesseur a copié-collé la macro, on ne se déplace pas

Regarder la sortie du préprocesseur

MACRO DE LOG – UN PEU DE VARIADIQUE

```
1   #include <stdio.h>
2
3   #define log_msg(...)                                                    \
4   printf("LOG dans la fonction %s à la ligne %d : ", __func__, __LINE__); \
5   printf(__VA_ARGS__);                                                    \
6   putchar('\n');
7
8   int main(int argc, char* argv[]) {
9       log_msg("Bonjour j'ai %d argument(s)", argc);
10  }
```

Macro variadique, C99 §6.10.3/10

C99 §6.10.3/5, §6.10.3.1/2

./a.out helloworld
→ LOG dans la fonction **main** à la ligne 9 : Bonjour j'ai 2 argument(s)

MEMCPY… INDÉFINI ?

Comment déplacer des éléments dans un tableau ?

- Avant : { 0, 1, 2, 3 }
- Après : { 0, 0, 1, 2 }

Comportement indéfini !

```c
#include <string.h>

int main(void) {
    int arr[4] = { 0, 1, 2, 3 };

    memcpy(&arr[1], &arr[0], sizeof(int) * 3);
}
```

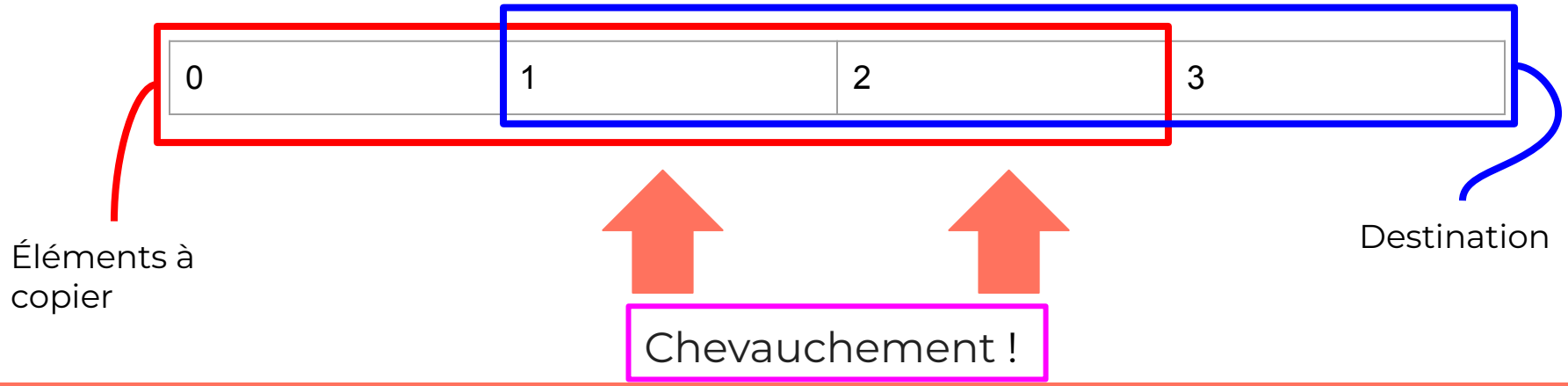
C89 §7.11.2.1 (Description)
C99 §7.21.2.1/2

Comportement de memcpy sur 6 compilateurs différents

MEMMOVE À LA RESCOUSSE

man (3) memcpy :

```
DESCRIPTION
       The memcpy() function copies n bytes from memory area src to memory area dest.
       The memory areas must not overlap.  Use  memmove(3) if the memory areas do overlap.
```



POURQUOI MEMMOVE ?

Résumé de C99 Rationale §7.21.2, à propos des fonctions de copie :

- Une fonction de copie doit fonctionner **même si les zones de mémoire se chevauchent**
- Une fonction de copie doit **être rapide** (et utiliser efficacement le hardware)
- Contradiction ⇒ gérer le chevauchement dégrade les performances
  - → **memcpy** pour la rapidité (ne gère pas le chevauchement)
  - → **memmove** pour la fonctionnalité (gère le chevauchement)

LES POINTEURS RESTREINTS

C89 §7.11.2.1

```
void* memcpy(void* dest, const void* src, size_t n);
void* memcpy(void* restrict dest, const void* restrict src, size_t n);
```

C99 §7.21.2.1

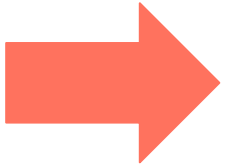
Un pointeur marqué **restrict** doit être le **seul** à accéder à sa zone de mémoire !
→ Sinon comportement indéfini
C99 §6.7.3/7, §6.7.3.1

Objectif: **optimisation !**

Pour aller plus loin :
- StackOverflow
- Wikipédia

SIZEOF(STRUCT) ET LES MATHS...

```c
struct char_plus_int {
    char c;
    int n;
};


#include <stdio.h>

int main(void) {
    printf(
        "char = %zu byte(s)\n"
        "int = %zu byte(s)\n"
        "struct { char, int } = %zu byte(s)\n",
        sizeof(char),
        sizeof(int),
        sizeof(struct char_plus_int)
    );
}
```



char = 1 byte(s) *
int = 4 byte(s) *
struct = 8 byte(s) *

Donc 1 + 4 = 8 ?

* Dépend du système

L’ALIGNEMENT EN MÉMOIRE

On s’attend à une structure de **5 octets** :

Hors de la structure

Adresse	0	1	2	3	4	5	6	7
Contenu	char	int	int	int	int	….	…	…

L'ALIGNEMENT EN MÉMOIRE

En réalité, on a une structure de **8 octets** :

int = 4 octets, doit* être **aligné** sur 4 octets.
⇒ Il est stocké à une adresse **multiple de 4**

```
struct char_plus_int {
    char c;
    uint8_t padding[3];
    int n;
};
```

Il existe des mots-clefs pour modifier/connaître l'alignement.

Adresse	0	1	2	3	4	5	6	7
Contenu	char	padding	padding	padding	int	int	int	int

3 octets de padding entre le char et l'int, pour décaler l'int.
Ils sont ajoutés par le compilateur.

* Pour plus de détails

LES CHAMPS DE BIT

Champ de bit → *bit-field* ou *bitfield* en anglais

```
struct {
    unsigned n;
    unsigned m : 2; // 0 --> 2^2 - 1
                    // 0 --> 4 - 1
                    // 0 --> 3
};
```

n est un champ normal.

m est un **champ de bits**, de 2 bits précisément.
m est un unsigned **tronqué** à 2 bits au lieu de généralement 32 ou 64 bits
m ne peut stocker une valeur **que de 0 à 3** *

Certaines restrictions s'appliquent aux champs de bits, les principales étant :

- On ne peut pas prendre son adresse avec & …

(C89 §6.5.2.1 note de bas de page n°59, C99 §6.7.2.1, note de bas de page n°103)

- Ni sa taille avec sizeof

(C89 §6.3.3.4 (*Constraints*), C99 §6.5.3.4/1)

* m est non-signé, donc l'overflow s'applique comme sur un unsigned normal, ça repart de 0. Si m était signé, alors l'overflow causerait un comportement indéfini

C89 §6.5.2.1, §6.7.2.1

LES CHAMPS DE BIT // <limits.h>

Champ de bit → *bit-field* ou *bitfield* en anglais

```c
struct coordonnée_échiquier {
    // 3 bits = 8 valeurs (2^3 = 8)
    uint8_t rangée : 3;
    uint8_t colonne : 3;
};
```

Usage bas niveau, **nombre de bits** à :

- Lire (protocoles réseau, décompression binaire)
- Écrire (compression binaire, systèmes embarqués spécifiques)
- **Nécessaire** pour fonctionner

Application spécifique, la **valeur maximale d'une donnée** est fixe et connue :

- Ici, coordonnées aux échecs (maximum 8)
- **Optimisation** (économie de mémoire)

Valeur minimale/maximale d'un type : <limits.h> (C89 §5.2.4.2.1, C99 §5.2.4.2.1)

- INT_MIN / INT_MAX
- UINT_MIN / UINT_MAX
- CHAR_MIN / CHAR_MAX
- ...

LES UNIONS

```
struct {
    int n;
    char c;
};
```

Structure :

- Contient **tous** les champs
- Prend de l'espace mémoire, mais...
- On peut lire et écrire dans tous les champs

```
union {
    int n;
    char c;
};
```

Union :

- Contient **un seul** des champs à la fois
- Économie de mémoire
- On peut écrire dans tous les champs, mais écrase la valeur du dernier champ modifié
- On peut lire tous les champs, mais seul celui dans lequel on a écrit aura un valeur cohérente

union + struct + champs de bits = float (*type-punning* explicitement autorisé en C mais indéfini en C++)

LES UNIONS – ÉTUDE DE (C)SFML

```c
typedef enum {
    sfEvtClosed,
    sfEvtKeyPressed,
    sfEvtMouseMoved
} sfEventType;
```

Type d'event

```c
typedef struct {
    char keyPressed;
} sfKeyEvent;

typedef struct {
    float x;
    float y;
} sfMouseMoveEvent;
```

Structure pour chaque event

```c
typedef struct {
    sfEventType type;
    union {
        sfKeyEvent key;
        sfMouseMoveEvent mouseMove;
    } event;
} sfEvent;
```

Structure event, avec un type et une union

Ici, l'union contient **l'unique** structure associée à l'event.

Le sfEvent original est directement une union et non une struct, car le sfEventType est le 1er champ et chaque structure d'event contient le sfEventType en 1er champ.
Source : CSFML

#MACRO

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

bool fail(void) {
    return false;
}

#define ASSERT(x) if (!(x)) {                                   \
    fprintf(stderr, "Assertion '" #x "' failed !\n");           \
    exit(1);                                                    \
}

int main(void) {
    ASSERT(fail());
}
```

Si x vaut fail()
→ #x vaut “fail()”
→ Utile pour debug

C89 §6.8.3.2
C99 §6.10.3.2

MACRO##MACRO

```
void cmd_start(void) {}
void cmd_end(void) {}
void cmd_cd(void) {}
```

Fonctions pour chaque commande

```
#define COMMAND(name) { #name, cmd_##name }
```

{ “start”, cmd_start }

```
typedef struct {
    const char* name;
    void (*f)(void);
} command_t;
```

Une commande = un nom et une fonction

```
const command_t commands[] = {
    COMMAND(start),
    COMMAND(end),
    COMMAND(cd)
};
```

C89 §6.8.3.3
C99 §6.10.3.3

C11 – DES MACROS GÉNÉRIQUES

```c
#include <stdio.h>

float invsqrtf(float f) { ... }
double invsqrt(double d) { ... }
long double invsqrtl(long double ld) { ... }

#define INVSQRT(x) _Generic(x,  \
    float: invsqrtf(x),         \
    double: invsqrt(x),         \
    long double: invsqrtl(x)    \
)

int main(void) {
    printf("%f\n", INVSQRT(4.f));
    printf("%f\n", INVSQRT(4.));
    printf("%Lf\n", INVSQRT(4.L));
}
```

Algorithme spécialisé selon le type de flottant

Invsqrt $\frac{1}{\sqrt{x}}$

#include <tgmath.h>
C99 §7.22

C11 §6.5.1.1

Les fonctions de maths sont souvent sales… Enfin optimisées.

Pour aller plus loin

C23 – LA MODERNISATION

Récapitulatif complet sur cppreference

```c
#include <stdint.h>

const uint8_t image_data[] = {
    #embed "image.png"
};
```

Inclusion de fichier binaire

```c
int main(void) {
    _BitInt(10) ten_bits = 0;
}
```

Entiers au bit près (pas encore de page cppreference)

constexpr

```c
// SIZE est une macro, copier-coller avant la compilation
// COMPILE !
#define SIZE 3
int array[SIZE] = { 0, 1, 2 };

// const vient indiquer que SIZE ne peut être modifiée,
// mais const n'indique pas que c'est ue constante
// SIZE n'est pas une constante, NE COMPILE PAS !
const int SIZE = 3;
int array[SIZE] = { 0, 1, 2 };

// constexpr (comme en C++) indique que SIZE est
// une constante de compilation
// COMPILE !
constexpr int SIZE = 3;
int array[SIZE] = { 0, 1, 2 };
```

LES BIZARRERIES

LES CLASSIQUES

One liner

```c
#include <stdio.h>
int main(int argc, char* argv[]){for(int i=0;i<argc;i++){printf("'%s'\n",argv[i]);}return 0;}
```

Variables illisibles

```c
int main(void) {
    int _ = 23654;
    int __ = 4589;
    int abcdefg = 7;
    int var0123456 = 1;
    return f1(_, __) + f2(abcdefg) / f3(var0123456);
}
```

Indentation douteuse

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

Abus du préprocesseur

```c
#include <stdio.h>

#define AFFICHE int main
#define LES (void)
#define ENTIERS {
#define DE for (int i =
#define A ; i <=
#define S_IL_TE_PLAIT ; i++){printf("%d\n", i);}}

AFFICHE LES ENTIERS DE 1 A 10 S_IL_TE_PLAIT
```

LES DI/TRIGRAPHES

```
%:include <stdio.h>
%:include <string.h>
int main(void) {
    char str??(??) = "hello world";
    printf("'%s' = %zu character(s)\n", str, strlen(str));
}
```

```
#include <stdio.h>
#include <string.h>
int main(void) {
    char str[] = "hello world";
    printf("'%s' = %zu character(s)\n", str, strlen(str));
}
```

Trigraphes, C89 §5.2.1.1, C99 §5.2.1.1

??=	#	??)	]	??!	|
??(	[	??'	^	??>	}
??/	\	??<	{	??-	~

In all aspects of the language, the six tokens[81)]

<: :> <% %> %: %:%:

behave, respectively, the same as the six tokens

[ ] { } # ##

Digraphes, C95 §6, C99 §6.4.6/3

Trigraphes retirés en C++17 et en C23 !
Pour aller plus loin →N4210 (C++) N2940 (C)

LES DI/TRIGRAPHES

Clavier IBM Model M 1390572 sans les crochets []
1986-1987? Wikipédia

Trigraphes, C89 §5.2.1.1, C99 §5.2.1.1

??= #	??)]	??! |
??([	??' ^	??> }
??/ \	??< {	??- ~

In all aspects of the language, the six tokens[81)]

<: :> <% %> %: %:%:

behave, respectively, the same as the six tokens

[ ] { } # ##

Digraphes, C95 §6, C99 §6.4.6/3

Trigraphes retirés en C++17 et en C23 !
Pour aller plus loin →N4210 (C++) N2940 (C)

LES DI/TRIGRAPHES – PIÈGES

Trigraphes = <u>préprocesseur</u>

```c
int main(void) {
    printf( "What??!\n" );
}
```

```c
int main(void) {
    printf( "What|\n" );
}
```

Digraphes = opérateurs

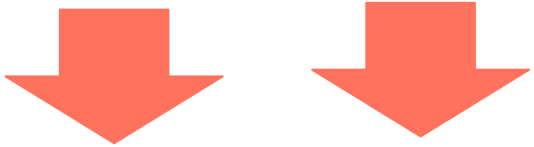
```c
int main(void) {
    printf( "What%:\n" );
}
```

```c
int main(void) {
    printf( "What%:\n" );
}
```

LES DI/TRIGRAPHES – PIÈGES

Trigraphes = **PRÉPROCESSEUR** → remplacés dans les commentaires / chaînes de caractères

```c
int main(void) {
    // returning 1??/
    return 1;
    return 0;
}
```



```c
int main(void) {
    // returning 1 \
    return 1;
    return 0;
}
```



```c
int main(void) {
    /* returning 1
    return 1; */
    return 0;
}
```

```c
int main(void) {
    puts("Hello??/" world");
}
```

```c
int main(void) {
    puts("Hello\" world");
}
```

FONCTIONS K&R

Déclaration / Prototype K&R

```c
int f();
int main(int argc, char* argv[]) {
    return f(argc, argv);
}

int f(int argc, char* argv[]) {
    return argc;
}
```

Valide mais... obsolète !
C89 §6.9.4, §6.9.5
C99 §6.11.6, §6.11.7
C99 §Introduction/2

Définition K&R

```c
int main(argc, argv)
    int argc;
    char* argv[];
{
    return 0;
}
```

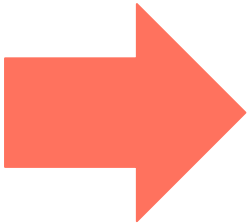
```c
int main(int argc, char* argv[]) {
    return 0;
}
```

Supprimé en C23 ! N2432

COMPTER LES VOITURES...

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

int main(void) {
    char word[80];
    int auto = 0;
    while (scanf("%79s", word) == 1) {
        if (!strcmp(word, "car")
        ||  !strcmp(word, "auto")
        ||  !strcmp(word, "automobile"))
            auto++;
    }
    printf("cars: %d\n", auto);
    return 0;
}
```

Ce code ne compile pas !



auto est un mot-clef !

Source : Une question sur 'auto' que j'ai posé sur StackOverflow

AUTO ? – UN PEU DE B...

B (1969) →C (1972)

```
main()
{
    extrn printf;
    auto x;
    x = 25;
    printf('%d', x);
}
```

```
int n = 4;
```

n est une variable dans la pile (stack), son stockage est dit automatique.

auto existe en C pour des raisons historiques et est inutile depuis longtemps

```
auto int n = 4;
```

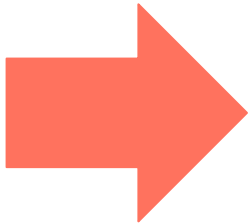
Découvrir le B

Pour aller plus loin → Une question sur 'auto' que j'ai posé sur StackOverflow

AUTO ? – UN DÉTOUR EN C++ ?

C++11 : déduction de type (*type inference* en anglais)

```cpp
#include <array>

std::array<int, 10> empty_array() {
    return {};
}

int main(void) {
    auto array = empty_array();
}
```



```cpp
#include <array>

std::array<int, 10> empty_array() {
  return {};
}

int main() {
  std::array<int, 10> array = empty_array();
}
```

Voir sur CppInsights

Pour aller plus loin:

- auto (cppreference)
- auto vs decltype (StackOverflow)

AUTO ? – ET EN C ?

B*

C89→C17

auto a le même sens qu'en B

C23

auto s'utilise (presque*) comme en C++

* B n'étant qu'un langage de transition vers le C, il n'a pas de logo officiel.

* auto en C est dérivé de l'extension GNU __auto_type, moins puissante que le mot-clef auto en C++, voici quelques différences, et là aussi

INT ? – POURQUOI FAIRE ?

```
__main (void) {}
```

- Compile en C89 → int implicite
- Invalide à partir de C99

C89 §6.3.2.2 (Semantics, paragraphe 2)
C89, §6.5.2 (Constraints, “or no type specifier”)
C99 §Foreword/5

INT ? – POURQUOI FAIRE ?

```
f();

int main(argc, argv)
    char* argv[];
{
    return f(argc, argv);
}

f(argc, argv)
    char* argv[];
{
    return argc;
}
```

f(void) déclare une fonction :
- **Sans argument**
- Qui renvoie un int

f() déclare une fonction :
- Avec un **nombre inconnu** d'arguments
- Qui renvoie un int

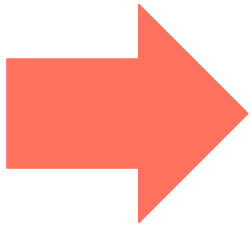
Même avec **-std=c89**, GCC 14.2.0 et Clang 19.1 émettent des avertissements, car certaines de ces fonctionnalités sont aujourd'hui dépréciées et/ou supprimées (mais parfaitement valides en C89).

C89 §6.5.4.3 (Constraints, "*An identifier list declares only…*" + note 71)

INT ? – POURQUOI FAIRE ?

- Compile en C89 → int implicite
- Invalide à partir de C99

```c
int main(void) {
    auto n = 7;
    auto array[] = { 1, 2 };
    auto* first = &array[0];

    return 0;
}
```



```c
int main(void) {
    int n = 7;
    int array[] = { 1, 2 };
    int* first = &array[0];

    return 0;
}
```

DÉCLARE-MOI 2 POINTEURS !

```
int main(void) {
    int* ptr1;
    int* ptr2;
}
```

```
int main(void) {
    int *ptr1;
    int* ptr2;
}
```

```
int main(void) {
    int* ptr1;
    int *ptr2;
}
```

```
int main(void) {
    int *ptr1;
    int *ptr2;
}
```

```
int main(void) {
    int *ptr1, *ptr2;
}
```

Correct !

```
int main(void) {
    int* ptr1, ptr2;
}
```

```
int main(void) {
    int *ptr1, ptr2;
}
```

ptr1 = int**
ptr2 = **int** !

DÉCLARE-MOI 2 POINTEURS !

Réponse courte: le pointeur * s'applique au nom de la variable, et pas à son type.
StackOverflow

```c
int main(void) {
    // pareil pour les tableaux
    // a est un tableau mais b est un int
    int a[12], b;
}
```

```c
int main(void) {
    int* ptr1, ptr2;
}
```

```c
int main(void) {
    int *ptr1, ptr2;
}
```

Réponse longue :

C89 §6.5, C99 §6.7
Une déclaration est composée de...

- Au moins un **spécificateur** qui est... :
  - Un type (non-pointeur) et/ou... // C89 §6.5.2, C99 §6.7.2
  - Une classe(s) de stockage) et/ou... // C89 §6.5.1, C99 §6.7.1
  - Un qualificateur (const, ...) et/ou... // C89 §6.5.3, C99 §6.7.3
  - Un spécificateur de fonction (inline) // C99 seulement, C99 §6.7.4
- Et potentiellement un **déclarateur** ou une **liste de déclarateurs** séparés par une virgule
  - Un déclarateur : // C89 §6.5.4, C99 §6.7.5
    - Un **identificateur** (nom de variable) **et potentiellement...**
      - Un **pointeur** et/ou
      - Un **tableau**
      - etc..
  - Qui peut être initialisé (= valeur) // C89 §6.5.7, C99 §6.7.8

int* ptr1, ptr2
→ type + pointeur + identificateur, identificateur
Pointeur + identificateur = déclarateur
⇒ type + déclarateur, déclarateur
⇒ variable type*, variable type

ptr1 = int*
ptr2 = **int** !

UNE STRING ENTRE APOSTROPHES ?

```
"ABC" = const char str[] = { 'A', 'B', 'C', '\0' };
'ABC' = ???
```

Char	‘A’	‘B’	‘C’	‘\0’
ASCII	0x41	0x42	0x43	0x00

Plusieurs caractères entre apostrophes → un **int** avec une valeur dépendante de l’implémentation (“implementation-defined”)

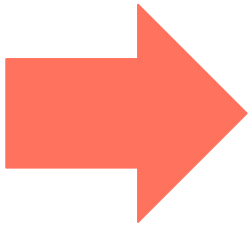
Souvent, ‘ABC’ = 0x414243
On a aussi souvent un **int** de 4 octets et du little endian

- ‘ABC’ = 0x00434241 → “\0CBA”

C89 §6.1.3.4 (Semantics, “*The value of an integer character constant containing more than one character [...] is implementation-defined.*”)
C99 §6.4.4.4/10

UNE STRING ENTRE APOSTROPHES ?

```c
#include <stdio.h>

int main(void) {
    int arr[] = { 'ulas', '!t' };
    puts((char*)arr);
}
```



Affiche **salut!**

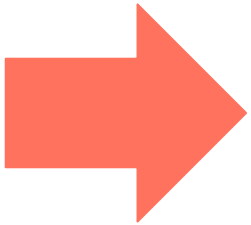
- ‘ulas’ → ‘salu’
- ‘!t’ → ‘t!\0\0’

⇒ On a “salut!\0\0”

Même comportement pour 6 compilateurs

UNE STRING ENTRE APOSTROPHES ?

```c
#include <stdio.h>

int main(void) {
    const char c = 'A';
    const char c2 = 'B';

    switch ((c << 8) | c2) {
    case 'AB':
        puts("--> AB");
        break;

    default:
        putchar(c);
        putchar(c2);
        putchar('\n');
        break;
    }
}
```



Utiliser switch avec plusieurs caractères à la fois (jusqu'à 4 si un int fait 4 octets).

switch ne peut prendre que des entiers (donc pas de string) :

- C89 §6.6.4.2 (Constraints)
- C99 §6.8.4.2/1

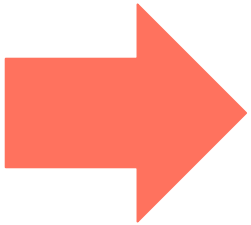
Test sur les 6 compilateurs

Avec des switch imbriqués

UNE STRING ENTRE APOSTROPHES ?

```
enum program_state {
    STOPPED = 'STOP',
    RUNNING = 'RUN!',
    WAITING = 'WAIT',
    ABORTED = 'ABRT'
};
```

État de l'exécution d'un programme



Ancienne méthode de debug, lorsqu'on doit lire un dump de la mémoire, la valeur de l'enum est lisible.

Source (StackOverflow)

(NOT) RETURN 0

Standard / Compilateur	C89 Indéfini* ! C89 §5.1.2.2.3	C99 et + 0 garanti C99 §5.1.2.2.3/1
Clang 19.1.0	0	0
GCC 14.2.0	5	0

Code de retour de main()

```c
#include <stdio.h>

int main(void) {
    printf("hello");
}
```

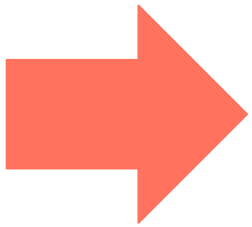
Pas de return

* indéfini, donc ça peut varier à chaque exécution

TABLEAU OU POINTEUR ?

Conversion implicite de tableau en pointeur
C89 §6.2.2.1 (paragraphe 3)
C99 §6.3.2.1/3

```c
#include <stdlib.h> /* NULL */

void f(int array[3]) {
    array = NULL;
}

int main(void) {
    int array[3] = { 0, 1, 2 };
    return 0;
}
```



```c
#include <stdlib.h> /* NULL */

void f(int* array) {
    array = NULL;
}

int main(void) {
    int array[3] = { 0, 1, 2 };
    return 0;
}
```

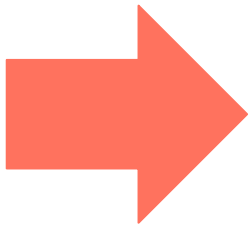
Pointeur implicite

Même chose pour “int array[]” !

UN PETIT TOUR DANS LES CROCHETS

[const] → qualificateur sur le **pointeur implicite**
C99 §6.7.5

```c
#include <stdlib.h> /* NULL */

void f(int array[const]) {
    array = NULL;
}

int main(void) {
    int array[3] = { 0, 1, 2 };
    return 0;
}
```



```c
#include <stdlib.h> /* NULL */

void f(int* const array) {
    array = NULL;
}

int main(void) {
    int array[3] = { 0, 1, 2 };
    return 0;
}
```

Ne compile pas !

UN PETIT TOUR DANS LES CROCHETS

[static n] → indication sur la taille minimale attendue du tableau
C99 §6.7.5.3/7

```c
#include <stdio.h>

void f(int array[static 2]) {
    printf("%d %d\n", array[0], array[1]);
}

int main(void) {
    int array[5] = { 0, 1, 2, 3, 4 };

    f(array);
}
```

static N :

- Tableau d'**<u>au moins</u>** N éléments
- Comportement indéfini si NULL ou moins de N éléments

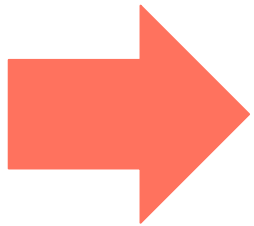
```c
#include <stdio.h>

int main(void) {
    int array[static 1] = { 0, 1 };
}
```

Ne compile pas
→ static autorisé **<u>uniquement</u>** dans les paramètres de fonctions !

UN TABLEAU... SANS TAILLE ?

Pas de taille !

```c
#include <stdio.h>

int i[];

int main(void) {
    printf("%d\n", i[0]);
}
```



```c
#include <stdio.h>

int i[1];

int main(void) {
    printf("%d\n", i[0]);
}
```

C99 §6.9.2/5

UN TABLEAU… SANS TAILLE ?

Pas de taille !

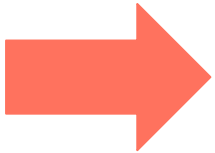
```c
#include <stdio.h>

int main(void) {
    int i[];

    printf("%d\n", i[0]);
}
```

N’est pas une variable globale*
→ ne compile pas !

* Notion de “file scope”, C99 §6.9.2/1

UN TABLEAU… SANS TAILLE ?

```c
#include <stddef.h>

struct string {
    size_t len;
    char* str;
};
```



```c
int main(int argc, char* argv[]) {
    struct string str;
    str.len = strlen(argv[0]);
    str.str = malloc(sizeof(char) * (str.len + 1));
    memcpy(str.str, argv[0], str.len);
}
```

⇒ **Optimisation** !

```c
#include <stddef.h>

struct string {
    size_t len;
    char str[];
};
```

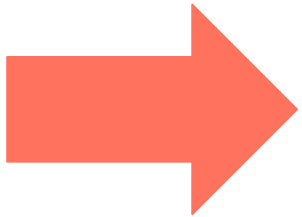
```c
int main(int argc, char* argv[]) {
    const size_t len = strlen(argv[0]);
    struct string* str = malloc(sizeof(struct string) + sizeof(char) * (len + 1));
    str->len = len;
    memcpy(str->str, argv[0], len);
}
```

Pour aller plus loin
Pour aller plus loin (2)
Avantages et inconvénients

C99 §6.7.2.1/16 : “Flexible Array Member”

UN TABLEAU… SANS TAILLE ?

```c
#include <stddef.h>

struct string {
    size_t len;
    char str[];
};

int main(int argc, char* argv[]) {
    struct string str = {
        .len = 4,
        .str = "ABC"
    };
}
```



Ne compile pas !
Impossible d'initialiser un "flexible array member".

C99 §6.7.2.1/17, §6.7.2.1/20

LES POINTEURS NUL(L)S

NULL n'est pas **forcément** égal à 0 en binaire...
- → C89 §7.1.6
- → C99 §7.17/3 ("implementation-defined")
- → Sur certaines vieilles machines, **NULL** n'est pas 0

Mais doit faire comme si c'était 0 :
- → Donc les opérateurs ==, != etc.. se comportent comme avec un "vrai" 0
- → 0 est un pointeur nul (C99, §6.3.2.3/3)
- → Du coup **NULL** peut être 0, 0x00, (void*)0, 0L....
- → if (NULL) → if (NULL != 0)

Donc c'est **standard et portable** de faire :
- if (ptr) → if (ptr != NULL)
- if (!ptr) → if (!(ptr == NULL))

Macro NULL définie dans les en-têtes :

En-tête	C89	C99
<locale.h>	§7.4	§7.11/3
<stddef.h>	§7.1.6	§7.17/3
<stdio.h>	§7.9.1	§7.19.1/3
<stdlib.h>	§7.10	§7.20/3
<string.h>	§7.11.1	§7.21.1/1
<time.h>	§7.12.1	§7.23.1/2
<wchar.h>	N'existe pas !	§7.24.1/3

POSIX exige que NULL soit (void*)0

TABLEAUX ANONYMES

```c
#include <stdio.h>

void print_str_array(const char* strings[]) {
    for (int i = 0; strings[i] != NULL; i++) {
        puts(strings[i]);
    }
}

int main(void) {
    const char* strings[] = { "hello", "world", NULL };
    print_str_array(strings);
}

int main(void) {
    print_str_array((const char*[]){ "hello", "world", NULL });
}
```

“Littéral composé” (*Compound literal)*

(const char*[]){“hello”, “world”}

- Variable anonyme
- De type const char* []
- Contenant 2 éléments
- “hello” et “world”

C99 §Foreword/5
C99 §6.5.2.5

STRUCTURES ANONYMES

```c
#include <stdio.h>

struct person {
    const char* name;
    unsigned age;
};

void print_person(const struct person* person) {
    printf("%s is %u year(s) old\n", person->name, person->age);
}

int main(void) {
    const struct person joe = {
        .name = "Joe",
        .age = 40
    };
    print_person(&joe);
}
```

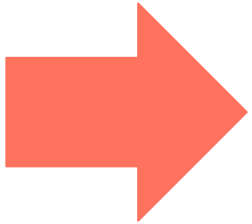
```c
int main(void) {
    print_person(&(struct person){
        .name = "Joe",
        .age = 40
    });
}
```

Littéral composé = *lvalue*
→ On peut prendre son adresse !

C99 §Foreword/5
C99 §6.5.2.5

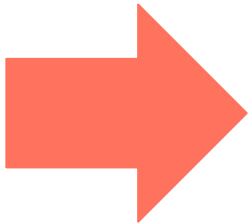
VARIABLES ANONYMES

```c
#include <stdio.h>

int main(void) {
    printf("%d\n", (int){41});
}
```



Fonctionne aussi pour les types simples (non tableaux ni structures)

```c
int main(void) {
    (int){0} = 1;
}
```



(int){0} est une *lvalue* (“une variable”), donc on peut lui assigner une valeur

C99 §Foreword/5
C99 §6.5.2.5

LES CONDITIONS

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

int main(void) {
    int n;

    while(true);
    while(1);
    while(-1);
    while(189);
    while('A');
    while(5.6);
    while(&n);
    while(main);
    while(!NULL);
}
```

C89 §6.6.5
C99 §6.8.5

while (x) s'exécute si x ≠ 0
Même chose pour if et for

Voir Slide "LES POINTEURS NUL(L)S"

LES CONDITIONS

```c
#include <stdio.h>

int main(void) {
    while ((char[]){0}) {
        puts("hello");
    }
}
```

```c
#include <stdio.h>

static char arr[] = {0};

int main(void) {
    while (&arr) {
        puts("hello");
    }
}
```

Slide “TABLEAUX ANONYMES”
→ (char[]){0} est un tableau de char

Slide “TABLEAU OU POINTEUR ?”
→ qui se convertit en un char*

Le tableau a une durée de vie statique
→ C99 §6.5.2.5/9

SWITCH ≠ IF … ELSE

```c
int main(int argc, char* argv[]) {
    switch (argc) {
        case 1:
            puts("No argument.");
            break;

        case 2:
            printf("Arg: '%s'\n", argv[1]);
            break;
    }
    return 0;
}
```

≠

```c
int main(int argc, char* argv[]) {
    if (argc == 1) {
        puts("No argument.");
    } else if (argc == 2) {
        printf("Arg: '%s'\n", argv[1]);
    }
    return 0;
}
```

SWITCH ≠ IF … ELSE

```c
int main(int argc, char* argv[]) {
    switch (argc) {
        case 1:
            puts("No argument.");
            // break;

        case 2:
            printf("Arg: '%s'\n", argv[1]);
            break;
    }
    return 0;
}
```

./a.out
→ Affiche “No argument”
→ Passe au “case 2”
→ Affiche argv[1] alors qu’il est NULL
(%s attend un char* non NULL)
→ Comportement indéfini !

En pratique, il n’y a pas de crash, car printf est souvent implémenté pour gérer NULL avec %s. Sortie “réelle” :

No argument.
Arg: '(null)'

Explication du comportement indéfini sur StackOverflow
C89 §7.1.7, §7.9.6.1
C99 §7.1.4/1, §7.19.6.1/8
(spécificateur %s de printf)

SWITCH ≠ IF … ELSE

```c
int main(int argc, char* argv[]) {
    switch (argc) {
        case 1:
            puts("No argument.");
            // break;

        case 2:
            puts(argv[1]);
            break;
    }
    return 0;
}
```

./a.out
→ Affiche “No argument”
→ Passe au “case 2”
→ Affiche argv[1] alors qu’il est NULL
(puts attend un char* non NULL)
→ Comportement indéfini !

On a aussi un comportement indéfini avec puts, qui est un crash (segfault) en pratique, car les implémentations de puts ne gèrent pas NULL.

C89 §7.1.7, §7.9.7.10
C99 §7.1.4/1, §7.19.7.10

SWITCH ≠ IF … ELSE

```c
int main(int argc, char* argv[]) {
    switch (argc) {
        case 1:
            puts("No argument.");
            break;
        case 2:
            printf("Arg: '%s'\n", argv[1]);
            break;
    }
    return 0;
}
```

“Fallthrough” !

=

```c
int main(int argc, char* argv[]) {
    if (argc == 1) {
        goto one_arg;
    } else if (argc == 2) {
        goto two_args;
    } else {
        goto end;
    }

one_arg:
    puts("No argument");
    goto end;

two_args:
    printf("Arg: '%s'\n", argv[1]);

end:
    return 0;
}
```

Le "Duff's Device" : une ancienne technique qui abuse du switch pour augmenter les performances
Coroutines avec le Duff's Device

SWITCH

```c
#include <stdio.h>

int main(void) {
    switch (0) {
        case 0: puts("Hello");
    }
}
```

Affiche “Hello”

C89 §6.6.4.2 (Semantics, “*If no converted case constant…*”)
C99 §6.8.4.2/5

```c
#include <stdio.h>

int main(void) {
    switch (0) {
        puts("Hello");
    }
}
```

N’affiche rien !

BREAK ET CONTINUE C'EST PAREIL

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    switch (1) {
        while (0) {
        case 1:
            puts("Hello");
            continue;
        }
    }
}
```

while(0)
→ condition fausse
→ on ne rentre plus dans la boucle
→ on sort du switch

Une autre question que j'ai posée à ce sujet sur StackOverflow

LES ACCOLADES ÇA SERT À RIEN

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    if (argc == 1)
        puts("No argument.");
    else if (argc == 2)
        puts("1 argument");
    else
        printf("%d arguments\n", argc - 1);
}
```

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    do puts("Hello"); while (1);
}
```

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

int main(int argc, char* argv[]) {
    while (true) for (int i = 0; i < 5; i++) printf("%d\n", i);
}
```

```c
#include <stdio.h>

int main(int argc, char* argv[]) {
    switch (1) while (0) case 1: switch (0) default: puts("Hello");
}
```

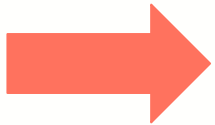
C89 §B.2.3, C99 §A.2.3 ⇒ if/else/while… attendant un *statement*, qui peut être un *expression-statement* qui peut n'être qu'un point-virgule ;

CODE GOLF* – AFFICHER ARGV

En C89

```
main(c,v)char**v;{while(*v)puts(*v++);}
```

```
main(argc, argv)
    /* int argc --> int implicite en C89*/
    char** argv;
{
    while(*argv) {
        puts(*argv++);
    }
}
```



```
main(argc, argv)
    /* int argc --> int implicite en C89 */
    char** argv;
{
    /* argv[argc] est NULL */
    while (argv[0]) {
        puts(argv[0]);
        argv++;
    }
}
```

C89 §5.1.2.2.1
C99 §5.1.2.2.1/2

* Code golf : pratique consistant à faire le code le plus court, en utilisant le moins de caractères/octets possible

INDEX[TABLEAU]

```c
#include <stdio.h>

int main(void) {
    const char str[] = "hello";
    putchar(str[0]);
    putchar(0[str]);
}
```

```c
str[0] = *(str + 0);
0[str] = *(0 + str);
```

C89 §6.3.2.1 (Semantics)
C99 §6.5.2.1/2

DES CLASSES … DE STOCKAGE !

Classe de stockage	Description
typedef	Alias de type
extern	Symbole défini autre part
static	Durée de vie du début à la fin du programme
auto	Stockage automatique (dans la pile)
register *	(Potentiel) stockage dans un registre. Non-addressable (impossible de récupérer l'adresse d'une variable register)

C89 §6.5.1
C99 §6.7.1

Impactent comment est stocké / linké un symbole.
Sauf typedef qui est une classe de stockage par commodité
→ C89 §6.5.1 (Semantics, 1er paragraphe), C99 §6.7.1/3

* Pour aller plus loin à propos de register

INT CONST OU CONST INT ?

C99 (extrait simplifié)

6.7 Declarations

Syntax

1 *declaration:*

declaration-specifiers *init-declarator-list*$_{opt}$ **;** → Pointeur **et/ou** '= valeur'

declaration-specifiers:

storage-class-specifier *declaration-specifiers*$_{opt}$ → Classe de stockage : extern, static...

type-specifier *declaration-specifiers*$_{opt}$ → Type (non pointeur) : int, float, char, void...

type-qualifier *declaration-specifiers*$_{opt}$ → Qualificateur de type : const, volatile, restrict

```
const int static a;
const static int b;
int const static c;
int static const d;
static const int e;
static int const f;
```

Valide !

```
typedef int integer;
int typedef integer;
```

C89 §6.5
C99 §6.7

INT CONST OU CONST INT ?

La rétrocompatibilité !

→ En 34 ans, rien n'a changé ! *

	C89	C99	C23
Autorise une classe de stockage à ne pas être au début	§6.5 (Syntax)	§6.7/1	§6.7.1/1
Déclare cette fonctionnalité obsolète	§6.9.3	§6.11.5/1	§6.11.6/1

```
const int static a;
const static int b;
int const static c;
int static const d;
static const int e;
static int const f;
```

Valide !

```
typedef int integer;
int typedef integer;
```

* Rien n'a changé sur ce point précis.
Même si le C change peu, C23 a beaucoup contribué à moderniser le langage.

COMMENT FAIRE UN TABLEAU ?

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

#define BUFFER_SIZE 1024

int main(void) {
    char buf[BUFFER_SIZE];

    int n;
    scanf("%d", &n);
    char* buf2 = malloc((n + 1) * sizeof(char));
}
```

OU

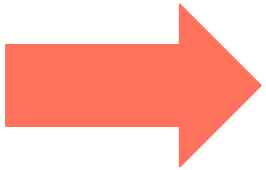
Taille fixe connue à la compilation.
C89 §6.5.4.2 (Constraints)
C99 §6.7.5.2/1

Taille inconnue à la compilation (ici dépend d'un input).
C89 §7.10.3.3
C99 §7.20.3.3

LES VLA

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

#define BUFFER_SIZE 1024

int main(void) {
    char buf[BUFFER_SIZE];

    int n;
    scanf("%d", &n);
    char* buf2 = malloc((n + 1) * sizeof(char));
}
```

ET ?



```c
#include <stdio.h>

int main(void) {
    int n;
    scanf("%d", &n);

    int vla[n];
    vla[0] = 4;
    return vla[0];
}
```

Taille inconnue à la compilation

C99 §6.7.5.2/4

SIZEOF

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(int));
}
```

4 (selon le système)

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(printf("hello")));
}
```

???

C99, §6.5.3.4/2

SIZEOF

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(printf("hello")));
}
```

printf renvoie un int

```c
int printf(const char *restrict format, ...);
```

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(int));
}
```

C99, §6.5.3.4/2

SIZEOF ET LES VLA

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(int[3]));
}
```

3 * 4 → 12 (selon le système)

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(int[printf("AB\n")]));
}
```

???

C99, §6.5.3.4/2

SIZEOF ET LES VLA

```c
#include <stdio.h>

int main(void) {
    printf("%zu\n", sizeof(int[printf("AB\n")]));
}
```

printf écrit 3 caractères → renvoie 3

sizeof(int[3])

3 * 4 → 12 (selon le système)

C99, §6.5.3.4/2

SIZEOF ET LES VLA

sizeof(x) :

- Exécute x si x est un **VLA**
- N'exécute **pas** x sinon

```c
#include <stdio.h>

int main(void) {
    // pas un VLA, int n'est pas exécuté
    printf("%zu\n", sizeof(int));

    // pas un VLA, 4 n'est pas exécuté
    printf("%zu\n", sizeof(4));

    // pas un VLA, printf(...) n'est pas exécuté
    printf("%zu\n", sizeof(printf("hello")));

    // VLA, int[printf(...)] est exécuté
    printf("%zu\n", sizeof(int[printf("AB\n")]));
}
```

C99, §6.5.3.4/2

DU CODE DANS LES ARGUMENTS ??

```c
#include <stdio.h>

void my_putstr(
    char *str,
    int tmp[sizeof(int[printf("%s", str)])]
) {}

int main(void) {
    my_putstr("hello", NULL);
}
```

On affiche str et on renvoie sa longueur.

On n'oublie pas que my_putstr prend un tableau → donc un pointeur
(Slide "TABLEAU OU POINTEUR ?")

Le paramètre est un VLA que l'on ignore.

C99, §6.5.3.4/2

Comportement pas garanti !
En pratique, ne compile pas ou alors marche
C99 §6.7.5.3/7* → un paramètre de type T[] devient T*
C99 §6.9.1/10 → quand on applle la fonction, l'argument tableau doit être converti en pointeur
⇒ Pas clair, donc la norme ne dit pas si le printf doit s'exécuter on non dans c cas-là.

* le commentaire Reddit cite §6.7.6.3/7, qui n'existe pas dans le document que j'ai posté le document ISO et non le brouillon

C11 ET LES VLA

Les VLA deviennent optionnels en C11 :

- §6.7.6.2/4
- §6.10.8.3 (__STDC_NO_VLA__)

Par exemple, CompCert 3.12 (un autre compilateur C) ne supporte pas les VLA :

- il définit __STDC_NO_VLA__
- il ne compile pas les VLA

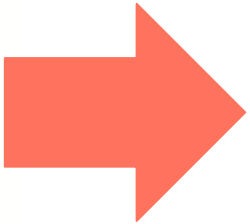
```c
#include <stdio.h>

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

Macro qui vaut 1 si les VLA ne sont pas supportés, sinon elle n'est pas définie.

En C23, les VLA en paramètres de fonction sont à nouveau obligatoires, mais les VLA en temps que variables locales restent optionels

UN PETIT VLA DANS LES CROCHETS

```c
#include <stdio.h>

void print_first(int n, int arr[*]);

void print_first(int n, int arr[n]) {
    printf("%d\n", arr[0]);
}

int main(void) {
    int arr[1] = { 4 };

    print_first(1, arr);
}
```



Dans une **déclaration** (prototype) de fonction **uniquement** :

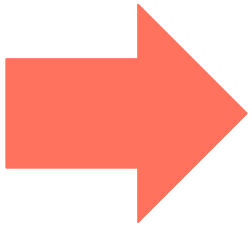
- int array[*] déclare un VLA
- Alors que int array[] déclare un tableau standard

C99, §6.7.5.2/4, §6.7.5.3/12

Pour aller plus loin

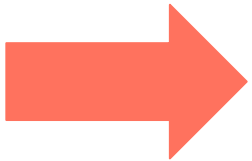
UN PETIT VLA DANS LES CROCHETS

```c
#include <stdio.h>

void print_first(int arr[*]) {
    printf("%d\n", arr[0]);
}

int main(void) {
    int arr[] = { 4 };

    print_first(arr);
}
```



Ne compile pas !
→ int arr[*] est dans une **définition** de fonction, au lieu d'une **déclaration** !

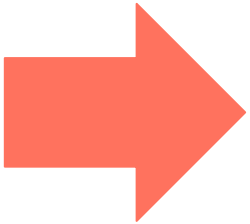
```c
#include <stdio.h>

int main(void) {
    int arr[*] = { 0, 1, 2 };
}
```



Ne compile pas !
→ int arr[*] est dans le **corps** d'une fonction, et non dans sa **déclaration** !

UN PETIT VLA DANS LES CROCHETS

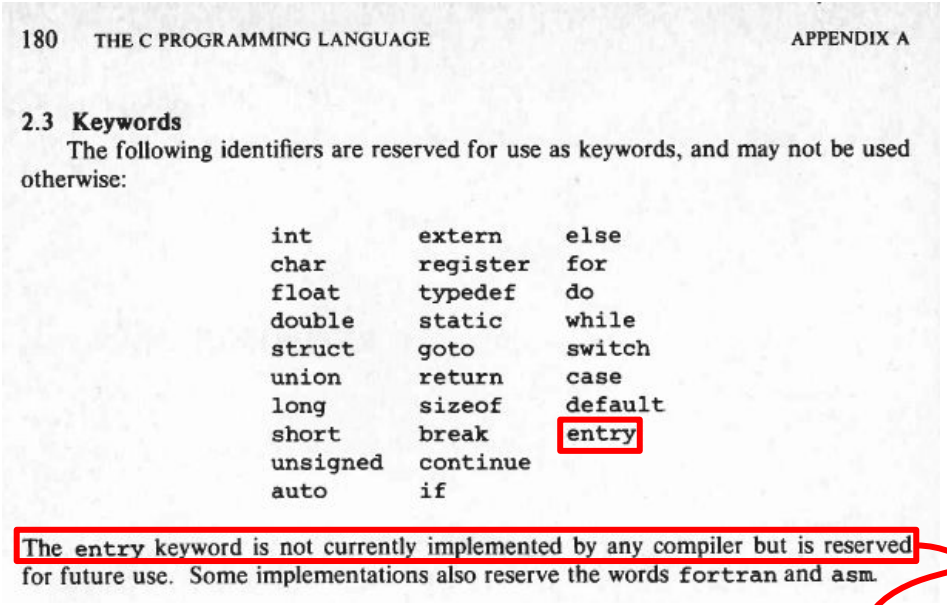
```c
#include <stdio.h>

void print_first(int n, int arr[*]);

void print_first(int n, int arr[1]) {
    printf("%d\n", arr[0]);
}

int main(void) {
    int arr[1] = { 4 };

    print_first(1, arr);
}
```



int[*] est compatible avec int[1]
→ à la fin ce sera un int*
(voir Slide “TABLEAU OU POINTEUR ?”)

AUTOUR DU C

ENTRY



180 THE C PROGRAMMING LANGUAGE APPENDIX A

2.3 Keywords

The following identifiers are reserved for use as keywords, and may not be used otherwise:

int	extern	else
char	register	for
float	typedef	do
double	static	while
struct	goto	switch
union	return	case
long	sizeof	default
short	break	entry
unsigned	continue	
auto	if	

The `entry` keyword is not currently implemented by any compiler but is reserved for future use. Some implementations also reserve the words `fortran` and `asm`.

The ENTRY statement, a nonexecutable statement, looks like this:

```
ENTRY name(argument list)
```

where name is the entry point name, and the optional argument list is made up of variable names, array names, dummy procedure names, or an asterisk. The asterisk, indicating an alternate return, is permitted only in a subroutine.

When an entry name is used to enter a subprogram, execution begins with the first executable statement that follows the ENTRY statement. The flow of control is illustrated in the following diagram.

```
        PROGRAM main
|<---- CALL entry1(val)
|       CALL entry2(val) ------>|
|                               |
|       END                     |
|                               |
|       SUBROUTINE sub          |
|                               |
-----> ENTRY entry1(a)          |
       a = a + 5.0              |
       RETURN ! Return to main  |
                                |
       ENTRY entry2(a) <--------
       a = a + 10.0
       END ! Return to main
```

En FORTRAN

Une question que j'ai posée sur le StackExchange Retrocomputing à ce sujet

Une autre question sur StackOverflow

K&R §A.2.3

#PRAGMA ET GCC

Retiré depuis longtemps !

```
#if 0
/* This was a fun hack, but #pragma seems to start to be useful.
   By failing to recognize it, we pass it through unchanged to cc1.  */

/*
 * the behavior of the #pragma directive is implementation defined.
 * this implementation defines it as follows.
 */
do_pragma ()
{
  close (0);
  if (open ("/dev/tty", O_RDONLY) != 0)
    goto nope;
  close (1);
  if (open ("/dev/tty", O_WRONLY) != 1)
    goto nope;
  execl ("/usr/games/hack", "#pragma", 0);
  execl ("/usr/games/rogue", "#pragma", 0);
  execl ("/usr/new/emacs", "-f", "hanoi", "9", "-kill", 0);
  execl ("/usr/local/emacs", "-f", "hanoi", "9", "-kill", 0);
nope:
  fatal ("You are in a maze of twisty compiler features, all different");
}
#endif
```

GCC 1.21 (copyright 1986-1987), fichier cccp.c ligne 2985

#pragma n'est pas standard ! *

GCC :

- Lance un 1er jeu (nethack)
- Lance un 2e jeu (rogue)
- Lance un 3e jeu (hanoi)
- Affiche un message d'erreur

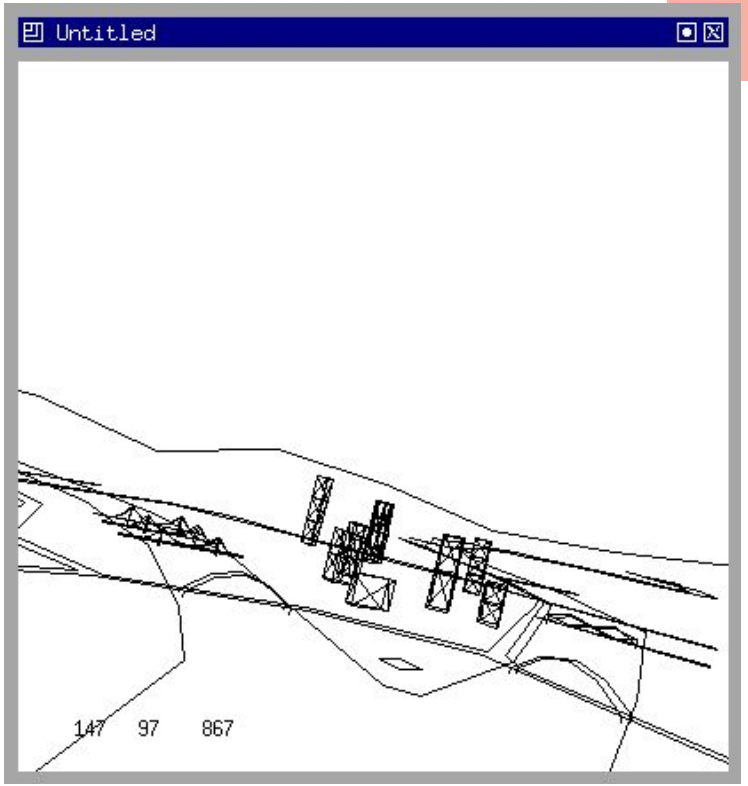
* #pragma (C89 §6.8.6, C99 §6.10.6) et _Pragma (introduit en C99, §6.10.9) sont standards, mais leurs arguments non (sauf quelques exceptions)

Pour en savoir plus

Code Source de GCC 1.21

```
#include                                <math.h>
#include                               <sys/time.h>
#include                               <X11/Xlib.h>
#include                              <X11/keysym.h>
                                       double L ,o ,P
                                      ,_=dt,T,Z,D=1,d,
                                      s[999],E,h= 8,I,
                                      J,K,w[999],M,m,O
                                     ,n[999],j=33e-3,i=
                                     1E3,r,t, u,v ,W,S=
                                     74.5,l=221,X=7.26,
                                     a,B,A=32.2,c, F,H;
                                     int N,q, C, y,p,U;
                                    Window z; char f[52]
                                   ; GC k; main(){ Display*e=
 XOpenDisplay( 0); z=RootWindow(e,0); for (XSetForeground(e,k=XCreateGC (e,z,0,0),BlackPixel(e,0))
; scanf("%lf%lf%lf",y +n,w+y, y+s)+1; y ++); XSelectInput(e,z= XCreateSimpleWindow(e,z,0,0,400,400,
0,0,WhitePixel(e,0) ),KeyPressMask); for(XMapWindow(e,z); ; T=sin(O)){ struct timeval G={ 0,dt*1e6}
; K= cos(j); N=1e4; M+= H*_; Z=D*K; F+=_*P; r=E*K; W=cos( O); m=K*W; H=K*T; O+=D*_*F/ K+d/K*E*_; B=
sin(j); a=B*T*D-E*W; XClearWindow(e,z); t=T*E+ D*B*W; j+=d*_*D-_*F*E; P=W*E*B-T*D; for (o+=(I=D*W+E
*T*B,E*d/K *B+v+B/K*F*D)*_; p<y; ){ T=p[s]+i; E=c-p[w]; D=n[p]-L; K=D*m-B*T-H*E; if(p [n]+w[ p]+p[s
]== 0|K <fabs(W=T*r-I*E +D*P) |fabs(D=t *D+Z *T-a *E)> K)N=1e4; else{ q=W/K *4E2+2e2; C= 2E2+4e2/ K
 *D; N-1E4&& XDrawLine(e ,z,k,N ,U,q,C); N=q; U=C; } ++p; } L+=_* (X*t +P*M+m*l); T=X*X+ l*l+M *M;
  XDrawString(e,z,k ,20,380,f,17); D=v/l*15; i+=(B *l-M*r -X*Z)*_; for(; XPending(e); u *=CS!=N){
                                  XEvent z; XNextEvent(e ,&z);
                                      ++*((N=XLookupKeysym
                                       (&z.xkey,0))-IT?
                                       N-LT? UP-N?& E:&
                                       J:& u: &h); --*(
                                       DN -N? N-DT ?N==
                                       RT?&u: & W:&h:&J
                                        ); } m=15*F/l;
                                        c+=(I=M/ l,l*H
                                        +I*M+a*X)*_; H
                                        =A*r+v*X-F*l+(
                                        E=.1+X*4.9/l,t
                                        =T*m/32-I*T/24
                                         )/S; K=F*M+(
                                         h* 1e4/l-(T+
                                         E*5*T*E)/3e2
                                         )/S-X*d-B*A;
                                         a=2.63 /l*d;
                                         X+=( d*l-T/S
                                          *(.19*E +a
                                          *.64+J/1e3
                                          )-M* v +A*
                                          Z)*_; l +=
                                          K *_; W=d;
                                          sprintf(f,
                                          "%5d  %3d"
                                          "%7d",p =l
                                         /1.7,(C=9E3+
                                  O*57.3)%0550,(int)i); d+=T*(.45-14/l*
                                 X-a*130-J* .14)*_/125e2+F*_*v; P=(T*(47
                                 *I-m* 52+E*94 *D-t*.38+u*.21*E} /1e2+W*
                                 179*v)/2312; select(p=0,0,0,0,&G); v-=(
                                  W*F-T*(.63*m-I*.086+m*E*19-D*25-.11*u
                                   )/107e2)*_; D=cos(o); E=sin(o); } }
```

IOCCC



Site officiel de l'IOCCC / GitHub

Source

DES GOTO DANS UN TABLEAU ?

```c
#include <stdio.h>

void pair_ou_impair(int n) {
    static void* label[] = { &&pair, &&impair };

    printf("%d est ", n);
    goto *label[n % 2];

pair:
    puts("pair");
    return;

impair:
    puts("impair");
}

int main(void) {
    pair_ou_impair(4);
    pair_ou_impair(9);
}
```

→ **performance** !
Évite un appel de fonction.

Utilisé surtout pour les interpréteurs et VM (lecture et exécution du bytecode).

Extension de compilateur
→ non standard !

Supporté par au moins :
- GCC
- Clang

Pour aller plus loin
Un exemple de “tokenizer”
Un exemple de coroutines
Documentation de GCC

UN SWITCH ... ?

```c
#include <stdio.h>

enum http_status {
    HTTP_INFO,
    HTTP_SUCCESS,
    HTTP_REDIRECTION,
    HTTP_CLIENT_ERROR,
    HTTP_SERVER_ERROR,
    HTTP_INVALID
};

enum http_status classify_http_code(int code) {
    switch (code) {
        case 100 ... 199: return HTTP_INFO;
        case 200 ... 299: return HTTP_SUCCESS;
        case 300 ... 399: return HTTP_REDIRECTION;
        case 400 ... 499: return HTTP_CLIENT_ERROR;
        case 500 ... 599: return HTTP_SERVER_ERROR;
        default: return HTTP_INVALID;
    }
}
```

Extension de compilateur
→ non standard !

Supporté par au moins :
- GCC
- Clang

Documentation de GCC

MULTIPLIER LES DÉGÂTS – IF/ELSE

Niveau	1 ⇒ 5	6 ⇒ 10	11 ⇒ 15	15 ⇒ 20
Multiplicateur	1	1.5	2	2.5

```
double get_damage(unsigned damage, unsigned level) {
    if (level <= 5) {
        return damage;
    } else if (level <= 10) {
        return damage * 1.5;
    } else if (level <= 15) {
        return damage * 2;
    } else {
        return damage * 2.5;
    }
}
```

MULTIPLIER LES DÉGÂTS – SWITCH…

Niveau	1 ⇒ 5	6 ⇒ 10	11 ⇒ 15	16 ⇒ 20
Multiplicateur	1	1.5	2	2.5

```
double get_damage(unsigned damage, unsigned level) {
    switch (level) {
        case 1 ... 5: return damage;
        case 6 ... 10: return damage * 1.5;
        case 11 ... 15: return damage * 2;
        default: return damage * 2.5;
    }
}
```

MULTIPLIER LES DÉGÂTS – TABLEAU…

```
double damage_multiplier[] = {
    [1 ... 5] = 1.0,   // Niveau 1 à 5: Dégâts de base
    [6 ... 10] = 1.5,  // Niveau 6 à 10: Dégâts x1.5
    [11 ... 15] = 2.0, // Niveau 11 à 15: Dégâts x2
    [16 ... 20] = 2.5  // Niveau 16 à 20: Dégâts x2.5
};

double get_damage(unsigned damage, unsigned level) {
    return damage_multiplier[level] * damage;
}
```

Extension de compilateur
→ non standard !

Supporté par au moins :
- GCC
- Clang

Documentation de GCC

RESSOURCES

RESSOURCES

Normes :

- K&R 1978
- Norme C89
- Révision C95 *
- Norme C99 (+ Rationale du C99)
- Norme C11
- Norme C17
- Norme C23 (a un historique des modifications depuis C89, voir annexe M)

IMPORTANT

shall = si c'est pas respecté alors comportement indéfini

- C89 §3, §3.16
- C99 §4, §4/2

Autres ressources :

- Un excellent cours sur le C en français
- Code de l'animation d'argv (pastebin) (fait avec ManimGL)
- Un site non-officiel du C
- Sibling Rivalry : C and C++ → Histoire du C89, C99 et C++
- Histoire du C (cppreference)
- Compilateur en ligne (Compiler Explorer)
- Histoire du C détaillée // Histoire du C (autres infos)
- Pourquoi les index des tableaux commencent à 0 et non 1 ?

* : Appelée aussi AMD1 (pour amendement), ce n'est pas une norme, la révision est intégrée dans C99.

CITER LA **NORME** – NOTATIONS

Extrait du C99

6.6 Constant expressions

Syntax

1 *constant-expression:*
conditional-expression

Description

2 A constant expression can be evaluated during translation rather than runtime, and accordingly may be used in any place that a constant may be.

Constraints

3 Constant expressions shall not contain assignment, increment, decrement, function-call, or comma operators, except when they are contained within a subexpression that is not evaluated.[95)]

Section 6.6, on note §6.6

Point n°2 de la section 6.6, on note §6.6/2

SECTIONS DE LA NORME C99

- “Foreword” : informations diverses liées à l’ISO + changements depuis la dernière norme (ici C89)
- Introduction : Pourquoi le document existe ? Qu’est-ce qu’il contient (vaguement) ?
- Section 1 “Scope” : Quels aspects d’un programme C sont évoqués ou non ?
- Section 2 “Normative references” : liens vers d’autres normes ISO pertinentes
- Section 3 “Terms, definitions, and symbols” : du vocabulaire important (argument, bit…)
- Section 4 “Conformance” : Critères généraux que les implémentations doivent respecter
- Section 5 “Environment” : Contraintes externes
    - Comment un fichier C est traité par le compilateur ?
    - Comment est exécuté un programme C ?
    - Exécute-t-on un programme C sur un OS (“hosted”) ou non (“freestanding”) ?
- Section 6 “Language” : La syntaxe
- Section 7 “Library” : La bibliothèque standard, ses en-têtes (“headers”), types et fonctions
- Annexes variées…

ANNEXES DE LA NORME C99

- Annexe A “Language syntax summary” : Liste de tous les types de “tokens” d’un code C (mots-clés, chiffres, opérateurs, chaînes de caractères…)
- Annexe B “Library summary” : Macros & prototypes de fonctions listés par en-tête (header)
- Annexe C “Sequence points” : Endroits du code où tous les effets de bord (“side effects”) doivent être résolus
    - Exemple → i++ introduit un effet de bord, on doit évaluer i puis l’incrémenter
    - ”Sequence Points” sur StackOverflow
- Annexe D “Universal character names for identifiers” : Caractères Unicode autorisés dans des identificateurs (“identifiers”), c’est-à-dire noms de variables, fonctions, macros…
- Annexe E “Implementation limits” : Valeurs minimum des macros de limits.h
- Annexe F “IEC 60559 floating-point arithmetic” : Description du comportement des nombres flottants du C par rapport au standard IEC 60559
- Annexe G “IEC 60559-compatible complex arithmetic” : Comme l’annexe F, mais pour les nombres complexes

ANNEXES DE LA NORME C99

- Annexe H “Language independent arithmetic” : Description des opérations arithmétiques disponibles en C par rapport au standard ISO/IEC 10967–1 (standard qui liste des opérations mathématiques ainsi que leur domaine de définition)
- Annexe I “Common warnings” : Suggestions d’avertissements (“warnings”) pour les compilateurs → simplement des **suggestions** et peuvent totalement être ignorées
- Annexe J “Portability issues” : Liste des comportements (définis à C99 §3.4)…
  - Non spécifiés (*unspecified behavior)* → la norme propose plusieurs options et laisse le choix à l’implémentation
  - Indéfini (*undefined behavior* *) → la norme ne dit pas ce qu’il doit se passer
  - Dépendant de l’implémentation (*implementation-defined behavior*) → comportement non spécifié, l’implémentation doit documenter son choix
  - Dépendant de la locale (*locale-specific behavior*)
  - Extensions courantes des compilateurs/environnements
    - 3e argument dans main pour les variables d’environnement → int main(int argc, char* argv[], char* env[])
    - Mot-clef “asm” pour insérer des instructions en langage d’assemblage dans du C

* Pour aller plus loin à propos des comportements indéfinis

MERCI !

GitHub de la présentation

Merci au subreddit r/C_Programming pour la relecture.