THE QUIRKS OF C

Or **how to write dirty C** ?

```
#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); } }
```

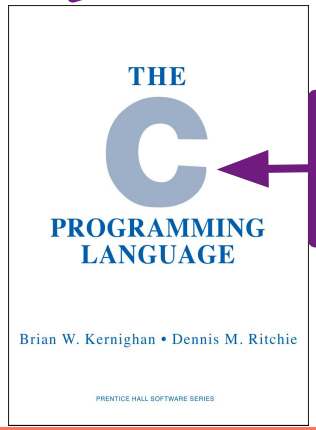
Thomas Sayen

A SIMPLIFIED HISTORY OF C

C is created by Dennis Ritchie

1972

1978 - K&R



Unofficial C standard. A book co-authored by Ritchie et Brian Kernighan, hence K&R.

Standardisation of C, C89 (aka ANSI C) becomes the very first official standard.

1983 → 1989 - ANSI & ISO (1990)

1999

C99 : Second C standard. Most used standard, as it's supported by ~100% of devices, and additions since C89.

Minor changes and additions.

2011 (C11) - 2018 (C17 or C18)

2023 (C23)

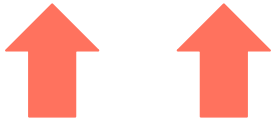
Significant language modernisation (removing old stuff) and importing some C++ ideas (nullptr / constexpr), experimental today (February 2025)

Blog article about K&R

WHICH VERSION ?

To this day (March 2025) :

C89	C99	C11	C17	C23
Extremely portable	Extremely portable + many additions	Added (among others) multithreading	Bug fixes	More modern but experimental



Default standard of GCC 14.2.0 and Clang 19.1.0

Cheat sheet C11→C23

How to detect the C standard used to compile our code ?

IBM I only supports up to C99 (§ Industry Standards)

C23 being still experimental, GCC 14.2.0 hasn't updated __STDC_VERSION__ yet.

USEFUL EXAMPLES

BOOLEANS

C89 :
→no boolean type !
→int often used instead

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

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

Since C99 : §7.16
→bool, true, false = macros
→_Bool = keyword (§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;
}
```

Since C23 : §6.4.2
→bool, true, false = keywords
→_Bool = keyword

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

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

INITIALIZE THE N-TH ELEMENT

```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

A MAP/DICTIONARY IN C99

```
KNIGHT = 0
BISHOP = 1
PAWN = 2
QUEEN = 3
KING = 4
ROOK = 5

PIECE_VALUE = {
    KNIGHT: 3,
    BISHOP: 3,
    PAWN: 1,
    QUEEN: 9,
    KING: 100,
    ROOK: 5
}
```

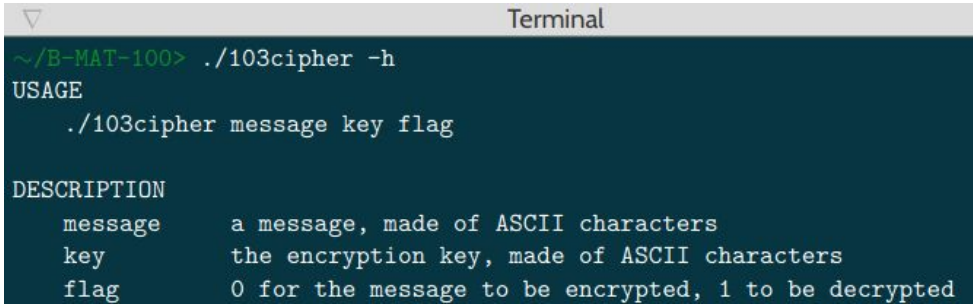
Python 3
(dictionary)

```
enum chess_piece {
    KNIGHT,
    BISHOP,
    PAWN,
    QUEEN,
    KING,
    ROOK,
    MAX_CHESS_PIECE
};

int piece_value[MAX_CHESS_PIECE] = {
    [KNIGHT] = 3,
    [BISHOP] = 3,
    [PAWN] = 1,
    [QUEEN] = 9,
    [KING] = 100,
    [ROOK] = 5
};

int main(void) {
    return piece_value[KNIGHT];
}
```

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, “Raw string literal” section

#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 → correct flag for printf/scanf

C99 §7.8.1

A SIMPLE PARSER

```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]

Uninitialized structure

Then we initialize each field

A SIMPLE PARSER

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

Everything is directly initialized, but not much readable (which false corresponds to which field ?)

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

Everything is directly initialized **and** it's readable, we explicitly write the name of each field and its value.

C99 §6.7.8/7 (designated initializer)

MACROS & PREDEFINED IDENTIFIERS

```c
#include <stdio.h>

int main(void) {
    printf(
        "File %s compiled on %s at %s\n",
        __FILE__,
        __DATE__,
        __TIME__
    );
    printf(
        "Function %s at line %d : %s\n",
        __func__,
        __LINE__,
        "Hello"
    );
}
```

__FILE__ (macro) :
→ string literal (in double quotes “”) containing file name

__DATE__* & __TIME__* (macros) :
→ hour and date where the file has been compiled

__func__ (const char []):
→ name of the function where we currently are

__LINE__ (macro) :
→ number of the line where we currently are

C89 §6.8.8 // C99 §6.10.8 (macros) C99 §6.4.2.2 (__func__ only)

* __DATE__, __TIME__ and locales

LOG FUNCTION

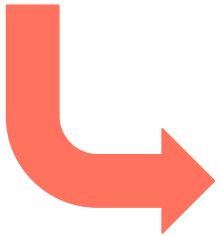
```
1   #include <stdio.h>
2
3   void log_msg(const char* msg) {
4       printf("LOG in function %s at line %d : %s\n", __func__, __LINE__, msg);
5   }
6
7   int main(void) {
8       log_msg("Hello");
9   }
```

We call the function and move there

LOG in function **log_msg** at line **4** : Hello

RATHER A LOG MACRO

```
1   #include <stdio.h>
2
3   #define log_msg(msg) printf("LOG in function %s at line %d : %s\n", __func__, __LINE__, msg);
4
5   int main(void) {
6       log_msg("Hello");
7   }
```



LOG in function **main** at line **6** : Hello

The macro has been pasted by the preprocessor, we don't move

LOG MACRO – A BIT OF VARIADIC

```c
#include <stdio.h>

#define log_msg(...)                                                    \
printf("LOG in function %s at line %d :\n", __func__, __LINE__);       \
printf(__VA_ARGS__);                                                    \
putchar('\n');

int main(int argc, char* argv[]) {
    log_msg("Hello I have %d argument(s)", argc);
}
```

Variadic macro, C99 §6.10.3/10

C99 §6.10.3/5, §6.10.3.1/2

./a.out helloworld
→ LOG in function **main** at line 9 : Hello I have 2 argument(s)

IS MEMCPY... UNDEFINED ?

How to move elements in an array ?

- Before : { 0, 1, 2, 3 }
- After : { 0, 0, 1, 2 }

Undefined behavior !

```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

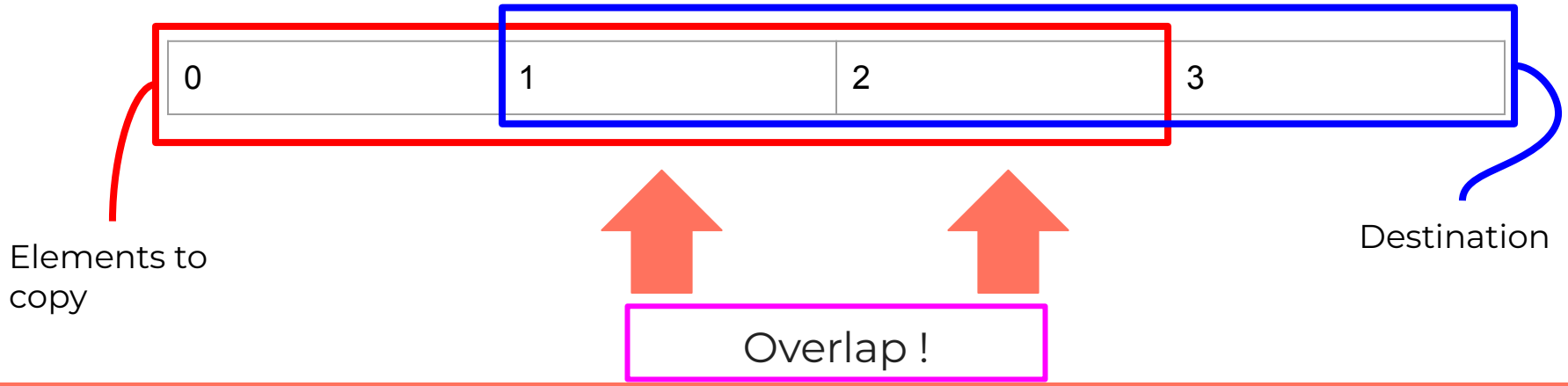
Behavior of memcpy on 6 different compilers

MEMMOVE TO THE RESCUE

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.



WHY MEMMOVE ?

Summary of C99 Rationale §7.21.2, about copy functions :

- A copy function must work **even if memory areas are overlapping**
- A copy function must **be fast** (and efficiently use hardware)
- Contradiction ⇒ handling overlap degrades performances
  - → **memcpy** for the speed (doesn't handle overlap)
  - → **memmove** for the feature (handles overlap)

RESTRICT POINTERS

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

A **restrict** pointer must be the **only one** to access its memory area !
→ Otherwise it's undefined behavior
C99 §6.7.3/7, §6.7.3.1

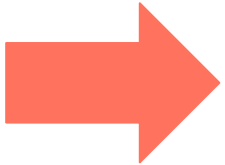
Goal: **optimization !**

To go further :

- StackOverflow
- Wikipedia

SIZEOF(STRUCT) AND 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) *

Then 1 + 4 = 8 ?

* According to the system

MEMORY ALIGNMENT

We expect our structure to take **5 bytes** :

Out of the structure

Address	0	1	2	3	4	5	6	7
Content	char	int	int	int	int	….	…	…

MEMORY ALIGNMENT

Actually, the structure takes **8 bytes** :

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

There are keywords to change/know the alignment.

int = 4 bytes, must* be 4 bytes-**aligned**.
⇒ Stored at an address being a **multiple of 4**

Address	0	1	2	3	4	5	6	7
Content	char	padding	padding	padding	int	int	int	int

3 padding bytes between the char and the int, to shift the int.
They're inserted by the compiler.

* For more details

BIT FIELDS

bit field, *bit-fields* and *bitfields* spellings exist.

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

n is a regular field.

m is a **bit field**, precisely 2 bits-wide.
m is a 2-bits **truncated** unsigned, instead of the usual 32 or 64 bits
m can only store values **from 0 to 3** *

Some restrictions are applied to bit fields, the main ones being :

- We cannot take their address with & ...

(C89 §6.5.2.1 footnote n°59, C99 §6.7.2.1, footnote n°103)

- Nor get their size with sizeof

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

* m is unsigned, so overflow works as it was a regular unsigned, it wraps around 0. If m was signed instead, then the overflow would be an undefined behavior.

C89 §6.5.2.1, §6.7.2.1

BIT FIELDS // <limits.h>

bit field, *bit-fields* and *bitfields* spellings exist.

```c
struct chess_coord {
    // 3 bits = 8 values (2^3 = 8)
    uint8_t rank : 3;
    uint8_t file : 3;
};
```

Low-level usage, an **amount of bits** to :

- Read (network protocols, binary decompression)
- Write (binary compression, specific embedded systems)
- **Needed** for the code to work

Specific application, some **data maximum value** is fixed and known :

- Here, coordinates in chess (maximum 8)
- **Optimization** (saves memory)

Minimum/maximum value of a 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
- ...

UNIONS

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

Structure :

- Contains **every** field
- Takes memory, but...
- We can read and write in every field

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

Union :

- Contains **only one** field at once
- Saves memory
- We can write in every field, but it overwrites the last written field
- We can read every field, but only reading the field we just wrote into will return a relevant value

union + struct + bit fields = float (*type-punning explicitly allowed in C but undefined in C++*)

UNIONS – (C)SFML CASE STUDY

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

Event type

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

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

Structure for each event

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

Event structure, with a type and an union

Here, the union contains **the unique** structure related to the event.

The original sfEvent is directly an union and not a struct, as sfEventType is the 1st field and each event structure contains that sfEventType as its 1st field.
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());
}
```

If x is fail()
→ #x is “fail()”
→ Useful for debugging

C89 §6.8.3.2
C99 §6.10.3.2

MACRO##MACRO

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

Functions for each command

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

{ “start”, cmd_start }

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

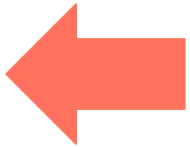
A command = a name and a function

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

C89 §6.8.3.3
C99 §6.10.3.3

C11 – GENERIC MACROS

```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));
}
```



Specialized algorithm according to floating point type

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

#include <tgmath.h>
C99 §7.22

C11 §6.5.1.1

Maths functions are usually unreadable... I mean optimized.

To go further

C23 – THE MODERNIZATION

Complete recap on cppreference

```c
#include <stdint.h>

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

Including a binary file

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

Bit-precise integers (no cppreference page yet)

constexpr

```c
// SIZE is a macro, copy-paste before compiling
// COMPILES !
#define SIZE 3
int array[SIZE] = { 0, 1, 2 };

// const indicates that SIZE can't b modified (is readonly)
// but const doesn't creates a constant
// SIZE isn't a constant, DOESN'T COMPILE !
const int SIZE = 3;
int array[SIZE] = { 0, 1, 2 };

// constexpr (as in C++) indicates that SIZE is
// a compilation constant
// COMPILES !
constexpr int SIZE = 3;
int array[SIZE] = { 0, 1, 2 };
```

THE QUIRKS

CLASSICS

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;}
```

Unreadable variables

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

Dubious indentation

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

Preprocessor abuse

```c
#include <stdio.h>

#define DISPLAY int main
#define ALL (void)
#define INTEGERS {
#define FROM for (int i =
#define TO ; i <=
#define PLEASE ; i++){ printf("%d\n", i); } }

DISPLAY ALL INTEGERS FROM 1 TO 10 PLEASE
```

DI/TRIGRAPHS

```
%: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));
}
```

Trigraphs, 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

[ ] { } # ##

Digraphs, C95 §6, C99 §6.4.6/3

Trigraphs are removed in C++17 and C23 !
To go further →N4210 (C++) N2940 (C)

DI/TRIGRAPHS

IBM Model M 1390572 keyboard without square brackets []
1986-1987? Wikipedia

Trigraphs, 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

[ ] { } # ##

Digraphs, C95 §6, C99 §6.4.6/3

Trigraphs removed in C++17 and C23 !
To go further →N4210 (C++) N2940 (C)

DI/TRIGRAPHS – TRAPS

Trigraphs = preprocessor

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

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

Digraphs = operators

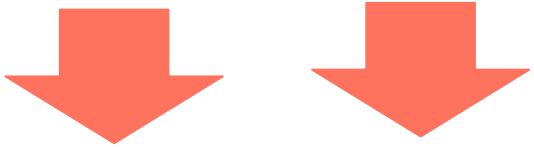
```c
int main(void) {
    printf( "What%:\n" );
}
```

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

DI/TRIGRAPHES – TRAPS

Trigraphs = **PREPROCESSOR** → replaced even in comments / strings

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



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



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

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

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

K&R FUNCTIONS

K&R Declaration / Prototype

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

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

Valid but obsolescent !
C89 §6.9.4, §6.9.5
C99 §6.11.6, §6.11.7
C99 §Introduction/2

K&R Definition

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

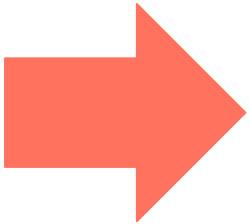
```
int main(int argc, char* argv[]) {
    return 0;
}
```

Removed in C23 ! N2432

COUNTING CARS...

```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;
}
```

It doesn't compile !



auto is a keyword !

Source : A question I asked about 'auto' on StackOverflow

AUTO ? – A BIT OF B...

B (1969) →C (1972)

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

```
int n = 4;
```

n is a variable stored in the stack, and has automatic storage.

auto exists in C for historical purposes and has been useless for a long time

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

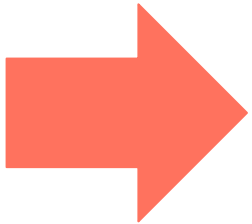
Learning B

To go further → A question I asked about 'auto' on StackOverflow

AUTO ? – A DIGRESSION IN C++ ?

C++11 : type inference

```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();
}
```

See on CppInsights

To go further:

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

AUTO ? – AND IN C ?

B*

C89→C17

auto is the same as in B

C23

auto is (almost*) the same as in C++

* B being only a transition language to C, it doesn't have an official logo.

* auto in C is derived from the GNU extension __auto_type, less powerful than auto keyword in C++, here are some differences, and here too

INT ? – WHAT FOR ?

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

- Compiles in C89 → implicit int
- Invalid since C99

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

INT ? – WHAT FOR ?

```
f();

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

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

f(void) declares a function :

- **Without any argument**
- Which returns an int

f() declares a function :

- With an **unknown number** of arguments
- Which returns an int

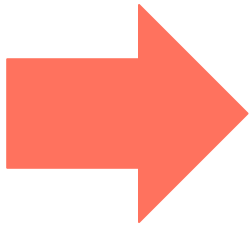
Even with **-std=c89**, GCC 14.2.0 and Clang 19.1 still emit warnings, as some of these features are nowadays deprecated and/or removed (but still perfectly valid in C89).

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

INT ? – WHAT FOR ?

- Compiles in C89 → implicit int
- Invalid since 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;
}
```

DECLARE ME 2 POINTERS !

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

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

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

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

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

Correct !

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

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

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

DECLARE ME 2 POINTERS !

Short answer: the pointer * binds/applies to the variable name, not to its type.
StackOverflow

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

Long answer :

C89 §6.5, C99 §6.7
A declaration is made up of...

- At least a **specifier** which is... :
  - A type (non pointer) and/or... // C89 §6.5.2, C99 §6.7.2
  - A storage class and/or... // C89 §6.5.1, C99 §6.7.1
  - A qualifier (const, ...) and/or... // C89 §6.5.3, C99 §6.7.3
  - A function specifier (inline) // since C99, C99 §6.7.4
- And potentially a **declarator** or a **list of** comma-separated **declarators**
  - A declarator : // C89 §6.5.4, C99 §6.7.5
    - An **identifier** (variable name) **and potentially...**
      - A **pointer** and/or
      - An **array**
      - etc..
  - Which could be initialized (= value) // C89 §6.5.7, C99 §6.7.8

int* ptr1, ptr2
→ type + pointer + identifier, identifier
Pointer + identifier = declarator
⇒ type + declarator, declarator
⇒ variable type*, variable type

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

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

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

A STRING IN SINGLE QUOTES ?

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

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

Multiple characters in single quotes → an **int** with an implementation-defined value

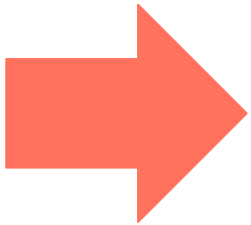
Usually, ‘ABC’ = 0x414243
We often have 4 bytes-wide **int** and little-endian

- ‘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

A STRING IN SINGLE QUOTES ?

```c
#include <stdio.h>

int main(void) {
    int arr[] = { 'lleh', '!o' };
    puts((char*)arr);
}
```



Displays **hello!**

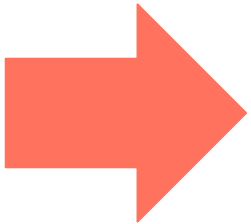
- ‘lleh’ → ‘hell’
- ‘!o’ → ‘o!\0\0’

⇒ Thus “hello!\0\0”

Same behavior for 6 different compilers

A STRING IN SINGLE QUOTES ?

```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;
    }
}
```



Using switch with multiple characters (up to 4 if an int is 4 bytes-wide).

Switch can only operate on integers (then no string) :

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

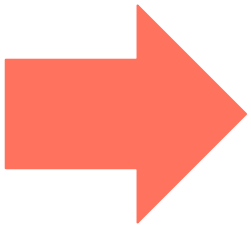
Tested on the same 6 compilers

With nested switches

A STRING IN SINGLE QUOTES ?

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

Program execution state



Old debug method, when one must read a memory dump, we make the enum value more readable.

Source (StackOverflow)

(NOT) RETURN 0

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

main() return code

```c
#include <stdio.h>

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

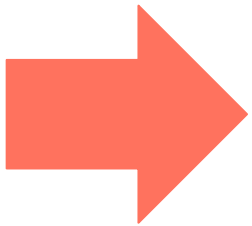
No return

* undefined, thus it may change between executions

ARRAY OR POINTER ?

Implicit conversion of arrays into pointers
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;
}
```

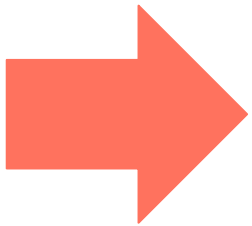
Implicit pointer

Same for “int array[]” !

A QUICK STROLL IN [BRACKETS]

[const] → qualifier on the **implicit pointer**
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;
}
```

Doesn't compile !

A QUICK STROLL IN [BRACKETS]

[static n] → indicates the expected minimum size of the array
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 :

- Array containing **at least** N elements
- Undefined behavior if NULL or less than N elements

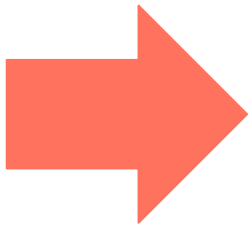
```c
#include <stdio.h>

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

Doesn't compile
→ static **only** allowed in function parameters !

AN UNSIZED... ARRAY ?

No size !

```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

AN UNSIZED... ARRAY ?

No size !

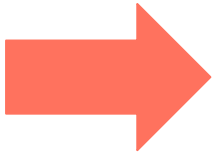
```c
#include <stdio.h>

int main(void) {
    int i[];

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

Isn't a global variable*
→ doesn't compile !

* Read about "file scope", C99 §6.9.2/1

AN UNSIZED... ARRAY ?

```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);
}
```

⇒ **Optimization** !

```c
#include <stddef.h>

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

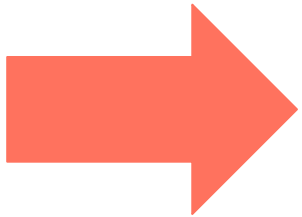
```c
int main(int argc, char* argv[]) {
    const size_t len = strlen(argv[0]);
    struct string* str = malloc(sizeof(struct string) + sizeof(char) * (len + 1));
    str->len = len;
    memcpy(str->str, argv[0], len);
}
```

To go further
To go further (2)
Pros and cons

C99 §6.7.2.1/16 : “Flexible Array Member”

AN UNSIZED... ARRAY ?

```c
#include <stddef.h>

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

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



Doesn't compile !
Cannot initialize a "flexible array member".

C99 §6.7.2.1/17, §6.7.2.1/20

NULL POINTERS

NULL isn't **necessarily** equal to 0 in binary...
- → C89 §7.1.6
- → C99 §7.17/3 ("implementation-defined")
- → Some ancient machines use nonzero **NULL**

But is must behave as if it was 0 :
- → Thus operators ==, != etc.. behave like they do with a "true" 0
- → 0 is a null pointer (C99, §6.3.2.3/3)
- → So **NULL** may be 0, 0x00, (void*)0, 0L....
- → if (NULL) → if (NULL != 0)

Then it's **standard and portable** to do :
- if (ptr) → if (ptr != NULL)
- if (!ptr) → if (!(ptr == NULL))

NULL macro is defined in headers :

Header	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>	Doesn't exist !	§7.24.1/3

POSIX requires NULL to be (void*)0

ANONYMOUS ARRAYS

```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 });
}
```

“Compound literal”

(const char*[]){“hello”, “world”}

- Anonymous variable
- Of type const char* []
- Containing 2 elements
- “hello” and “world”

C99 §Foreword/5
C99 §6.5.2.5

ANONYMOUS STRUCTURES

```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);
}
```

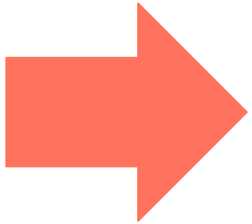
```c
int main(void) {
    print_person(&(struct person){
        .name = "Joe",
        .age = 40
    });
}
```

Compound literal = *lvalue*
→ We can take its address !

C99 §Foreword/5
C99 §6.5.2.5

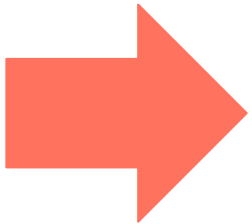
ANONYMOUS VARIABLES

```c
#include <stdio.h>

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



Also works for simple types (not arrays nor structures)

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



(int){0} is an *lvalue* (“a variable”), so we can assign a value to it

C99 §Foreword/5
C99 §6.5.2.5

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) is executed if x ≠ 0
if and for behave the same

See Slide “NULL POINTERS”

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");
    }
}
```

(char[]){0} is added in C99 → §6.5.2.5, “Compound literals”

The array has a static lifetime → 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
→ Displays “No argument”
→ Goes to “case 2”
→ Displays argv[1] even if it’s NULL
(%s expects a non-NULL char*)
→ Undefined behavior !

In practice, there’s no crash, as printf is often implemented to handle NULL with %s. “Real” output :

No argument.
Arg: '(null)'

Explanation of the undefined behavior on StackOverflow
C89 §7.1.7, §7.9.6.1
C99 §7.1.4/1, §7.19.6.1/8
(printf’s %s specifier)

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
→ Displays “No argument”
→ Goes to “case 2”
→ Displays argv[1] even if it’s NULL
(puts expects a non-NULL char*)
→ Undefined behavior !

We also have an undefined behavior with puts, which results in a crash (segfault), as puts usually doesn’t handle 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;
}
```

Duff's Device : an old technique abusing switch to improve performances
Coroutines with Duff's Device

SWITCH

```c
#include <stdio.h>

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

Displays “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");
    }
}
```

Doesn’t display anything !

BREAK AND CONTINUE ARE THE SAME

```c
#include <stdio.h>

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

while(0)
→ false condition
→ we don't re-enter in the loop
→ we exit the switch

Another question I asked about this on StackOverflow

CURLY BRACKETS ARE USELESS

```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… expects a *statement*, which may be an *expression-statement* which may just be a single semicolon ;

CODE GOLF* – PRINTING ARGV

In C89

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

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

```
main(argc, argv)
    /* int argc --> implicit int rule in C89 */
    char** argv;
{
    /* argv[argc] is NULL */
    while (argv[0]) {
        puts(argv[0]);
        argv++;
    }
}
```

C89 §5.1.2.2.1
C99 §5.1.2.2.1/2

* Code golf : challenge to make a code (very) short, using as few characters/bytes as possible.

INDEX[ARRAY]

```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

CLASSES... STORAGE CLASSES !

Storage class	Description
typedef	Type alias
extern	Symbol defined somewhere else
static	Alive from the beginning to the end of the program
auto	Automatic storage (in the stack)
register	(Potentially) stored in a register. Non-addressable (cannot get the address of a register variable)

C89 §6.5.1
C99 §6.7.1

Affects how a symbol is stored/linked.
Except typedef which is a storage class only for convenience
→ C89 §6.5.1 (Semantics, 1st paragraph), C99 §6.7.1/3

INT CONST VS CONST INT ?

C99 (simplified extract)

6.7 Declarations

Syntax

1 *declaration:*

declaration-specifiers *init-declarator-list*$_{opt}$ **;** → Pointer **and/or** '= value'

declaration-specifiers:

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

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

type-qualifier *declaration-specifiers*$_{opt}$ → Type qualifier: 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;
```

Valid !

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

C89 §6.5
C99 §6.7

INT CONST VS CONST INT ?

Backward compatibility !

→ Nothing changed for 34 years ! *

	C89	C99	C23
Allows a non-leading storage class	§6.5 (Syntax)	§6.7/1	§6.7.1/1
States this feature is obsolescent	§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;
```

Valid !

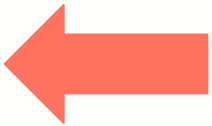
```
typedef int integer;
int typedef integer;
```

* This specific point hasn't changed.
Even if C evolves slower than other languages, C23 modernized the language a lot.

HOW TO CREATE AN ARRAY ?

```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));
}
```

OR

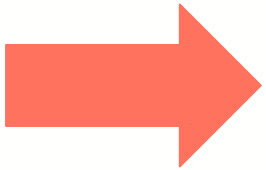


Fixed size, known at compile-time.
C89 §6.5.4.2 (Constraints)
C99 §6.7.5.2/1

The size is unknown at compile time (here, relies on an user input).
C89 §7.10.3.3
C99 §7.20.3.3

VLAs

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

#define BUFFER_SIZE 1024

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

    AND ?

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



```c
#include <stdio.h>

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

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

Unknown size at compile-time.

C99 §6.7.5.2/4

SIZEOF

```c
#include <stdio.h>

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

4 (may vary across different systems)

```c
#include <stdio.h>

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

???

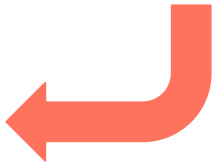
C99, §6.5.3.4/2

SIZEOF

```c
#include <stdio.h>

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

printf returns an 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 AND VLAs

```c
#include <stdio.h>

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

3 * 4 → 12 (according to the system)

```c
#include <stdio.h>

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

???

C99, §6.5.3.4/2

SIZEOF AND VLAs

```c
#include <stdio.h>

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

printf writes 3 characters → returns 3

sizeof(int[3])

3 * 4 → 12 (according to the system)

C99, §6.5.3.4/2

SIZEOF AND VLAs

sizeof(x) :

- Executes x if x is a **VLA**
- **Doesn't** execute x otherwise

```c
#include <stdio.h>

int main(void) {
    // not a VLA, int isn't executed
    printf("%zu\n", sizeof(int));

    // not a VLA, 4 isn't executed
    printf("%zu\n", sizeof(4));

    // not a VLA, printf(...) isn't executed
    printf("%zu\n", sizeof(printf("hello")));

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

C99, §6.5.3.4/2

CODE IN FUNCTION ARGUMENTS ??

```c
#include <stdio.h>

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

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

str is displayed and its length is returned.

tmp (a VLA) is ignored, as its only purpose was to execute printf

Don't forget that my_putstr takes an array argument → thus a pointer (Slide "ARRAY OR POINTER ?")

C99, §6.5.3.4/2

C11 AND VLAs

C11 made VLAs optional :

- §6.7.6.2/4
- §6.10.8.3 (__STDC_NO_VLA__)

```c
#include <stdio.h>

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

That macro expands to 1 if VLAs aren't supported, otherwise it's not even defined.

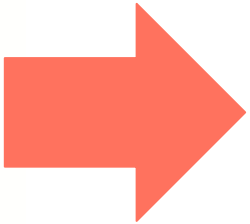
For instance, CompCert 3.12 (another C compiler) doesn't support VLAs :

- It defines __STDC_NO_VLA__
- It doesn't compile VLAs

In C23, VLAs as function parameters are mandatory again, but VLAs as local variables remain optional

A LITTLE VLA IN [BRACKETS]

```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);
}
```



Only in a function **declaration** (prototype):

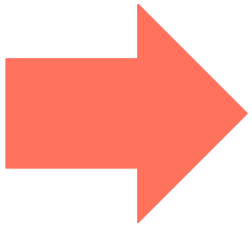
- int array[*] declares a VLA
- Whereas int array[] declares a regular array

C99, §6.7.5.2/4, §6.7.5.3/12

To go further

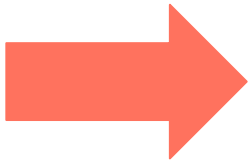
A LITTLE VLA IN [BRACKETS]

```c
#include <stdio.h>

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

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

    print_first(arr);
}
```



Doesn't compile !
→ int arr[*] is in a function **definition**, instead of a **declaration** !

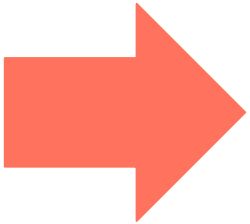
```c
#include <stdio.h>

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



Doesn't compile !
→ int arr[*] is in a function **body**, not in its **declaration** !

A LITTLE VLA IN [BRACKETS]

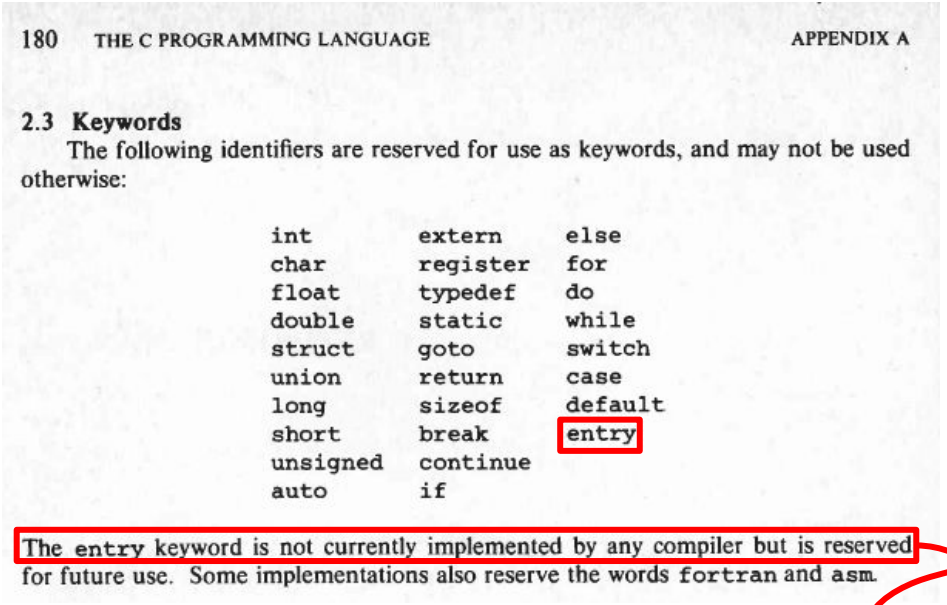
```c
#include <stdio.h>

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[*] is compatible with int[1]
→ at the end we'll have an int*
(see Slide "ARRAY OR POINTER ?")

SOME C-RELATED THINGS

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
```

In FORTRAN

A question I asked about this on Retrocomputing StackExchange

A StackOverflow question

K&R §A.2.3

#PRAGMA AND GCC

Removed since a long time !

```
#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), in file cccp.c at line 2985

#pragma is non-standard ! *

GCC :

- Runs a 1st game (nethack)
- Runs a 2nd game (rogue)
- Runs a 3rd game (hanoi)
- Displays an error message

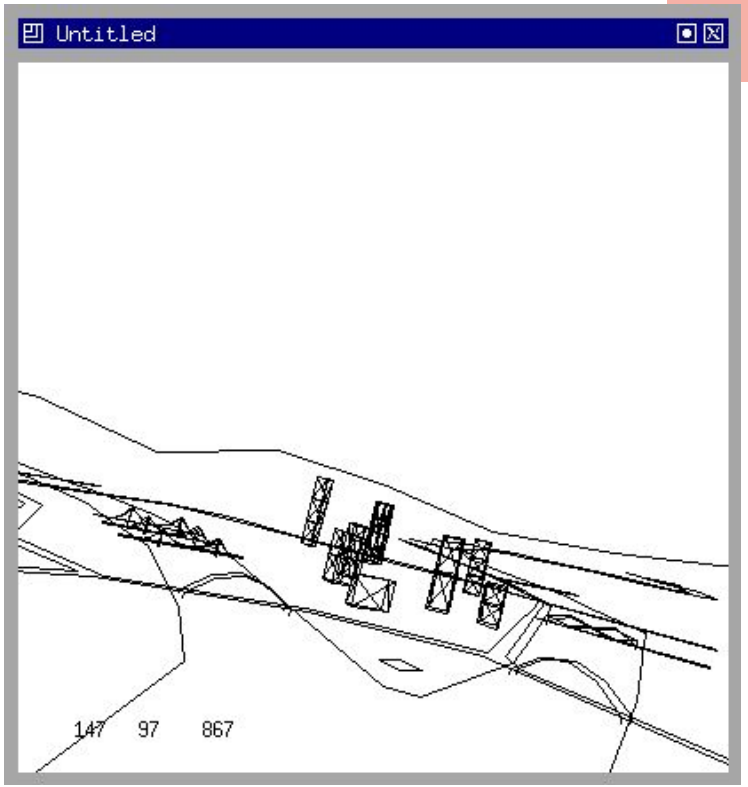
* #pragma (C89 §6.8.6, C99 §6.10.6) and _Pragma (added in C99, §6.10.9) are standard, but their arguments are not (with a few exceptions)

To go further

GCC 1.21 Source Code

```
#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



IOCCC official website / GitHub

Source

GOTOS IN AN ARRAY ?

```c
#include <stdio.h>

void even_or_odd(int n) {
    static void* label[] = { &&even, &&odd };

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

even:
    puts("even");
    return;

odd:
    puts("odd");
}

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

→ **performance** !
Avoids a function call.

Mostly used in interpreters and VMs (to read and then execute bytecode).

Compiler extension
→ non-standard !

At least supported by :
- GCC
- Clang

To go further
A "tokenizer" example
A coroutine example
GCC's documentation

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;
    }
}
```

Compiler extension
→ non-standard !

At least supported by :
- GCC
- Clang

GCC's documentation

MULTIPLYING DAMAGE – IF/ELSE

Level	1 ⇒ 5	6 ⇒ 10	11 ⇒ 15	15 ⇒ 20
Multiplier	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;
    }
}
```

MULTIPLYING DAMAGE – SWITCH...

Level	1 ⇒ 5	6 ⇒ 10	11 ⇒ 15	16 ⇒ 20
Multiplier	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;
    }
}
```

MULTIPLYING DAMAGE – ARRAY...

```
double damage_multiplier[] = {
    [1 ... 5] = 1.0,    // Level 1-5: Base damage
    [6 ... 10] = 1.5,   // Level 6-10: Damage x1.5
    [11 ... 15] = 2.0,  // Level 11-15: Damage x2
    [16 ... 20] = 2.5   // Level 16-20: Damage x2.5
};

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

Compiler extension
→ non-standard !

At least supported by :
- GCC
- Clang

GCC's documentation

RESOURCES

RESOURCES

Normes :

- K&R 1978
- C89 standard
- C95 amendment *
- C99 standard (C99 rationale)
- C11 standard
- C17 standard
- C23 standard (Annex M contains a changes history since C89)

IMPORTANT

shall = undefined behavior if constraint not fulfilled

- C89 §3, §3.16
- C99 §4/1, §4/2

Other resources:

- A pretty good course on C (but in French)
- argv animation's code (pastebin) (made with ManimGL)
- An unofficial C language website
- Sibling Rivalry : C and C++ → History of C89, C99 et C++
- History of C (cppreference)
- Online compiler (Compiler Explorer)
- Detailed C history // Another C history (with other details)
- Why do array indices begin at 0 and not 1 ?

* : aka AMD1 (AMD standing for amendment), it's not a standard, the amendment is included in C99.

QUOTING THE STANDARD – NOTATIONS

C99 excerpt

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, we write §6.6

2nd point of section 6.6, we write §6.6/2

C99 STANDARD SECTIONS

- “Foreword” : some information about ISO + changes since last standard (here C89)
- Introduction : Why does this document exist ? What does it (vaguely) contain ?
- Section 1 “Scope” : Which aspects of a C program are discussed or not ?
- Section 2 “Normative references” : links towards other relevant ISO standards
- Section 3 “Terms, definitions, and symbols” : important vocabulary (argument, bit...)
- Section 4 “Conformance” : General criteria to be met for the implementations
- Section 5 “Environment” : External constraints
    - How is a C file treated by the compiler ?
    - How is executed a C program ?
    - Do we execute a C program on an OS (“hosted”) or not (“freestanding”) ?
- Section 6 “Language” : Syntax
- Section 7 “Library” : The standard library and its headers, types and functions
- Various annexes...

C99 STANDARD ANNEXES

- Annex A “Language syntax summary” : Lists every “token” type of a C file (keywords, digits, operators, strings...)
- Annex B “Library summary” : Macros & function prototypes sorted by header
- Annex C “Sequence points” : Places in the code where all side effects must be resolved
  - Example → i++ introduces a side effect, we must evaluate i then increment it
  - ”Sequence Points” on StackOverflow
- Annex D “Universal character names for identifiers” : Allowed Unicode characters in identifiers, that is in the name of variables, fonctions, macros...
- Annex E “Implementation limits” : Minimum value of limits.h macros
- Annex F “IEC 60559 floating-point arithmetic” : Describes how C floating-point numbers behave, compared to IEC 60559 standard
- Annex G “IEC 60559-compatible complex arithmetic” : Like annex F, but for complex numbers

C99 STANDARD ANNEXES

- Annex H “Language independent arithmetic” : Describes arithmetic operations provided by C, compared to ISO/IEC 10967–1 standard (which lists many operations and their domain)
- Annex I “Common warnings” : Suggests some warnings for compilers → merely some **suggestions** and can absolutely be ignored
- Annex J “Portability issues” : Lists special behaviors (defined in C99 §3.4) :
    - Unspecified behavior → the standard lists many behaviors and leaves the choice to the implementation
    - Undefined behavior → the standard doesn’t say what must happen
    - Implementation-defined behavior → unspecified behavior, the implementation must document its choice
    - Locale-specific behavior
    - Usual compiler/environment extensions :
        - 3rd main argument for environment variables
          → int main(int argc, char* argv[], char* env[])
        - “asm” keyword to insert some assembly instructions

THANK YOU !

Slideshow on GitHub !

Many thanks to the r/C_Programming subreddit for the proofreading.