The Complete Reference

* Implemented by Dennis Ritchie on DEC PDP-11 - Unix OS (C-Result of - BCPL)
* BCPL - developed by Maorkin Richaords which Influenced Ken Thomson to write B.
* B led to C.
* C89 → C99 → C++

* mid level language (Combines high & low level language)
* Allows bit manipulation, bytes & addresses.
* portable - Software & OS independent.

(write - DOS, Run - Windows 2000)

---

* High level language - support data type
* C - datatypes are not strongly typed (∴ permits almost all type conversions)
* No runtime check - (array boundaries overrun - etc..) - resp. of programmer.
* C effects type conversion (Almost for all) - Automatic conversions too!
* C89 - 32 keywords, C99 - 37 keywords. (while high level lan - many more).
* C owes high hardware portability - as it doesn't has any file access or dynamic memory management statements / Console I/P & O/P.
* Instead C → provides extensive Std. library.

C - Structured language

* Building blocks - Functions (one can invoke another)
* But technically C - not a block structured (since a block-Structured language - permits procedure / func. Inside other proc/func.
* But C - doesn't allow function inside function!

* main() → must in every program (first func. to be executed)
* Top level of control - which can call other func as subroutines. (using local variables)
* Compartmentalize - other parts won't get affected! (Debug).
* C - allow to define & code functions individually - modular programming!

*

Area of circle

$3.14159265 \times r^2$

* Prototype of headers in #include <stdio.h> – preprocessor.

* Assembly – not structured – not easy to read, understand & debug!

Compilers vs Interpreters:

Compiled → C (Reads entire code → object code → (bin/machine code) – then execute!

Interpreted → Java (Read one line at a time – performs line by line)

Interpreted – slower than compiled program (Computer executes directly – one time cost) → Interpretation – incurs an overhead each time!

Form of a C program

By C99, (To manage 8086 family of microprocessors – Keywords (In several Compilers

asm	– ds	huge	pascal
cdecl	– es	Interrupt	– ss
– cs	far	near	

Keywords – C89

auto	continue	enum	if	short	switch	volatile
break	default	extern	int	signed	typedef	while
case	do	float	long	sizeof	union	– Bool
char	double	for	register	static	unsigned	– Imaginary
const	else	goto	return	struct	void	– Complex
						restrict
						inline

* Case sensitive (else not same as ELSE)

Source files

* Func def, global declaration, function – preprocessing directives → makesup a source file.

* Large C programs – several source files.

* C – supports modular programming! (organize program in many source files) to edit & compile them seperately.

eg: Suffix.c

problem: several source files: declare global var, func & macros → in many files

Solution: write all info (above) in a file (header)

#include ___.h to include it simply!

* Compiler translates – parse them into tokens – Smallest semantic unit. (variable name & operators).

* Any no. of white space allowed b/w two tokens – format source file (easy)!

* No line break / indentation rules – make it human readable (add lines)

But – Preprocessor must be strictly as
#include ___.h (no spaces)

Library & linking:

* C – No I/P O/P keywords – Need from standard external library.

* Standard → portable (else: not!)

C → combines our code with already written std. library source code!

"Linking"

Functions in library: Relocatable format: memory addresses for variables machine code ins. have not been absolutely assigned. (only keep offset info).

Seperate Compilation

*Large program – more compile time – many files

*Compile only modified files & its dependencies alone! (Save time)

Create → Compile → Link (Ide: do this for use)
.c

---

* Compiler only accepts std. text file as I/P – not word, etc..

C's memory map

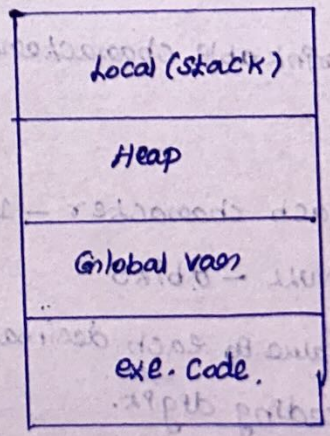
4 regions.

1 → holds the prog. exe code

2 → Global variable (memory)

3 → Stack

4 → Heap.



* Stack also saves current state of CPU

* Heap – dynamic memory allocation!

C vs C++

* C++ compilers compile C

Source code : Human readable i/p to compilers

object code : machine code (o/p to linkers)

Linkers : Links libraries etc..

Comments – // (or) /* */

↓ single line ↓ multiple line.

```
int open (const char *name, int mode, ... /* int permissions */ );
```

→ optional parameter!

Comment using conditional preprocessor directive

```
#if 0
    const double PI = 3.14159265358;
    area = PI * r * r;
endif
```

---

* A to Z
* a to z
* 0 to 9
* ! " # % & ' ( ) * + , - . / : ; < = > ? [\ ] ^ _ { | }

Space Characters:

* space
* horizontal tab
* vertical tab
* new line
* form feed.

non printable characters (4): * NULL, * alert, * backspace & Carriage return

\0 \a \b \r

* each character – 1 byte
* NULL – 0 bits
* value of each decimal digit after 0 is greater by one than that of the preceding digit.

---

wide size & multibyte characters

* C → developed in English speaking environment – where dominant set was 7-bit ASCII Code. (∴ 8-bits – became common with enc. encoding)

Realworld: more different symbols - hard to code in one byte. - multibyte character encoding schemes - non latin alphabets, non alphabetic chinese, Korean writing systems.

* wide char → same bit width used for every char in a char set.
* multi char (byte) → several bytes

Now: C has mechanisms to manipulate & convert these schemes

---

* wchar_t → wide char type (stddef.h) → large enough to represent any element of the given implementation's extended char sets.

* Although C doesn't require support for Unicode - UTF-16, 32 are used. (wchar_t → Atleast 16 bit - 32 bits wide)

eg: wchar_t wc = '\x3b1';

└→ hexadecimal, (lowercase alpha)

Better unicode support (C11) → char16_t, char32_t) uchar.h (unsigned int types) in header.

char16_t → defined by macro __STDC_UTF_16__

char32_t → macro. __STDC__UTF_32__

---

multibyte char

* each char sets → coded as a sequence of one / more bytes

'No two symbols - coded as same' → only \0 has all 0.

multibyte: varying no. of bytes. → independent of system architecture

widechar: same size → dependent on system's byte order (big-endian / little-endian)

Conversion

* wchar_t ⟷ multibyte character
* wctomb() → wide character to multibyte

wctomb (string pointer, wide char pointer) → returns no. of bytes required!

```
wchar_t wc = L'\x3B1'  → α
char mbStr[10] = "";
int nBytes = 0;
nBytes = wctomb( mbStr, wc);   (nBytes = 2)
                               mbStr = "\xCE\xB1".
```

char16_t ⟷ char32_t

C16rtomb()

'Universal character names'

universal char. names → use regardless of encoding.

\uXXXX (or) \u XXXXXXXX

Hexadecimal

\uXXXX (or) \u0000XXXX → Same

Identifiers

* Names of variables, functions, macros, structures, objects.

* A-Z, a-z (case sensitive), _, 0-9 (first char must not be a digit), universal char names rep. letters & digits of other languages.

'C has 44 keywords'

Predefined identifier: __func__ → use in any function to access a string constant containing the name of a function.

↓

'useful in debugging'!

---

* Readable, meaningful, short: Identifiers

Identifiers → ordinary
→ Label names
  → Tags (struct, union, enum)
  → member of struct / union.

File scope: → Global (entire file) – its valid (accessible from anywhere)!

Block Scope: only inside {...} block. (Need to declare first before using)!

'Scope: generally after its declaration'

exception: 'type name, tag, enum const → effect immediately'

eg: declare prototype – use – define at the end!

Compiling

* newline char, trigraphs – replaced (with single char they rep.)
* \ any → deleted!

That's why! ↑ 'Every source file, if not completely empty- must end with a new line char'

Source file → Tokens → obj → machine code! (binary)!

Token: (Keyword)/Identifier/Constant/String literal/Symbol

printf ("Hello, World.\n");

Tokens:

printf
(
"Hello, World.\n"
)
;

'If appending – makes token invalid → it won't be appended'

a+++b → Invalid!

a++ + b → valid! (white space removal – append)

'Token' stage!

Datatypes

Type: how much space, how encoded, signed/unsigned, operations!

* Basic → std & extended
  → Real & Complex floating point types
* Enum
* void type
* Derived → pointer
  → Array
  → structure
  → Union!

* Basic & Enum types ⟶ Arithmetic types
* Arithmetic & Pointer type ⟶ Scalar types.
* Array, Structure ⟶ Aggregate types
* function type ⟶ return value.

Foundational data types: char, int, float, double, valueless (void)

char - 1 byte
int - word size of machine.
} 7 basic data types.

C99 ⟶ _Bool, _Complex, _Imaginary

char → ASCII characters
int → integers
float, double → floating point numbers.

void → no returning value (generic pointers).

Functions → signed
→ unsigned
→ short
→ long.

char → signed
→ unsigned.

long → long long.

---

char → 2^8 → -127 to 127 (signed)

unsigned char → 2^8 → 0 to 255

int $2^{16/32}$ → -32,767 to 32,767

int $2^{16/32}$ → 0 to 65,535

long int → 2^{32} → -2,147,483,647 to 2,147,483,647

long long int → 2^{64}

'Ref: Pg:19'

---

* Variables: named location in memory - to hold a value.

Declare → Inside Functions
→ outside of all functions
→ In prototype.

Local variable: Inside functions (or) blocks - alone meaningful!

'auto' → keyword used to declare local variables.

* By default: All non-global variable → assumed to be auto (local variables)

```
if (x==1) {
    char s[80];
}
```

→ created while executing if & destroyed when it's done.

"If inner local variable-name same as outer name - outer one is hided"

example:

```
{
int x = 10;
if (x==10)
{
  int x = 99;
}
printf("%d", x); ⟶ 10
}
```

Note: C89 → must declare all variable at the start of the block!
Later versions: no need!

---

* entry → created → exit → destroyed
* Retain values of local variables: static. → "makes a copy in heap"

"formal parameters"

* function will be written with some assumed variables - formal parameters.
* (Behave like local variables)

Global variables

* Risky - since anywhere - it can be modified. - Avoid unnecessary.
* makes function dependable.
* Error prone!

Scopes → Block
→ function
→ prototype
→ File.

Type qualifiers → Const
→ Volatile
→ Restrict (C99)

Const: → may not be changed by program. (Initialization must) – Place in ROM.

"Many functions in std. library uses const" → eg: Strlen

size_t Strlen (const char *Str)

Ensure: Strlen doesn't modify the content! (No need)

Debug: make const & run to see whether program modifies anything!

Volatile

* Tells the compiler – variable's value may be changed in ways not explicitly specified by the program.

eg: global variable's address – passed to OS clock routine – hold the system time → contents may be altered without proper assignment.

* why?: most C compilers assume – unchanging variable content if it doesn't occur on the left side eg an assignments.

* volatile prevents this!

eg: const volatile char *port = (const volatile char *) 0x30;

Storage class specifiers → extern
→ static
→ Register
→ Auto

Complex floating point types (C99)

* math calc → involving Complex numbers – C99

"Complex.h" → Csin()
→ Ccos()
→ Ctan()

In C11 → Support for Complex numbers → optional

Include macro: __STDC_NO_COMPLEX__ → No need for headers.

* $x + iy$ → (x, y real), i - imaginary ($i^2 = -1$)

* float _ Complex
  double _ Complex
  long double _ Complex

Declare

```
double Complex z = 1.0 + 2.0 * I;
```

(or)

CMPLX
CMPXF
CMPLXL
→ float
→ double

eg: CMPLX (1.0, 2.0) → Same as

Enum

Enumerations are integer types - define in a program. The definition will be

enum [identifier] {enumeration_list}

eg:

enum Color { black, red, green, yellow, blue, white = 7, gray }; — Enumeration constants. - different element - may have same value.

Tag. (optional)

Implicit → By position (0, 1, 2, 3, 4, 7, 8)

Explicit → Initialize.

Pg: 97 (C in a nutshell)

define Variables

enum Color bgColor = blue;

enum Color { a, b, c, d = 0, e, f };
0 1 2 0 1 2 (Again from the prev value)

Void type

* No value is available - we can't declare variables / constants with this type.

* function declaration → No return
* No i/p arguments

eg: void person (const char *)

FILE *tmpfile (void);

Expression: "No value"

(fopen (filename, "r") == NULL) ⟶ void expression

Pointers to void:

void * ⟶ represents address of an object (type: unknown)!

* use: multipurpose pointer.

* more space – strictest than others (float, int)
4 4 bytes

* stddef.h ⟶ has these values (int, float etc..)

– Alignas(4) short var; ⟶ provides strictest alignment.

stdalign.h

Float ⟶ for floating numbers

Types

"\n" → int

a+1 → int

a+1.0 → double

"A literal" ⟶ char *

abort() → void

Sqrt(2.0) → double

z/Sqrt(2.0) → double_complex

&& → AND (logical)

|| → OR

?: → Conditional operator

, → comma operator

Precedence & Associativity

"Take care"

1. Postfix/prefix → left to right

[], (), •, → , ++ , --

(type name) {list}

2. Unary operators — Right to left

++ --

! ~ + - * &

sizeof _Alignof

3. Cast : (type name) → Right to left
4. mul op * / % — Left to right
5. Add op + - — "
6. Shift << , >> — "
7. Relational < <= > >= — left to right
8. Equality == , != — "
9. Bitwise AND : & — "
10. Bitwise exOR : ^ — "
11. Bitwise OR : | — "
12. Logical AND : && — "
13. " OR : || — "
14. ? : — Right to left
15. Assignment

=, += , -= , *=

/= , %= , &= , ^=

|= , <<= , >>=

— "

16. Comma operator — left to right

---

* Increment & decrement operators → higher precedence!

Arithmetic:

+ , - , * , / , % , +(unary), -(unary)

x*y, x/y, x%y, x+y, x-y, +x, -x

* modulo operand → Requires int

Pointer arithmetic

dptr = dptr + 1; → Advance to second element.

dptr = 2+dptr; → Advance to 4th element.

(default: increased by sizeof (int/double/..)

dptr++;
dptr += 1;
++dptr; } Same effect!

& → Address of (Pointer to x) - &x

* → Indirection operator (*p → object/function p points to)

[ ] → Subscripting x[y]

. → Structure / union member designator

-> (p -> y) . - structure/union member designated by reference.

x->m = (*x).m

Storage class specifiers

* "Tells how to store the subsequent variable."
* Storage specifier type var_name;

typedef - storage class for syntactic convenience.

* extern
* static
* Register
* Auto

Extern

C Linkage → External (Global var)
→ Internal (static)
→ None (Local variables)

* Global variables have external linkage - meaning available to all files of a program. File scope objects declared as static - have internal linkage - only known within the file - they are declared.

Local variables - no linkage - (∴ known only within their block)

* extern → specify object is declared with external linkage.

* declaration - declares name & type of an object

* Definition - causes storage to be allocated. (Same object may have many declaration but only one definition.

---

extern → Tells the compiler - Initialized somewhere!

```
#include <stdio.h>
int main()
{
 extern int a, b;
 printf ("%d %d", a, b);
 return 0;
}
int a=10, b=20;
```

→ "Here extern is must" - since a, b are only initialized after declaration.

---

* Extern - declare a variable without defining it.

If a variable (extern) - not in current block - compiler checks all { } enclosing blocks, then global variables - (matched) → assumes this is the one referenced.

"object can have multiple declarations but only one definition"

* declare all global variables in one file - with extern → use anywhere!

Static

* variables declared as static variables - permanent variables within their file/func.

* Not outside their func/file - maintain values b/w calls (when function over - their value won't expire - like global variables).

"Static local variables"

* compiler create permanent storage for it (like global variable)

Difference:

* Static local variable known only to the block.

* Static global /global → everywhere

why important? Stand alone functions: preserve values b/w calls.

* using global - have side effects!

make sure: no conflict b/w static local variable & global variable

eg:

```
void sample (int i)
{
static int a = 0;
a = a+i;
printf ("%d", a);
}
```

```
int main()
{
int a = 10;
for (int i=0; i<a; i++)
    sample (i)
```

1 3 6 10 15 21 28 36 45 55.

'preserved'

Static Global variables

* Instructs Compiler: create a global variable only known to the file-declared!
* Static global variable → Internal linkage. (other files won't know it)
* A replacement for static local!

'Static: permits variable - only to the func - that need them' 'Avoids side effects' 'hide portions - from other portions'

Register variables

* Applied only to type int / char / pointer types.
* Asks: keep in registers - rather in memory.
* operations done by registers - much faster (as don't need to acquire them from memory).
* practice: char & int - stored in registers while array, which can't be stored in a register - still receive treatment (acquire) from the Compiler.

Registers variables: only to local var & function formal parameters

* Can't able to get address of register variable using &

(Not addressable - Registers)

eg:

```
int sample (register int a, register int b)
{
  register int temp;
  temp = a + b;
  return temp
}
```

Constants (literals)

* Fixed value - don't alter

wide character constant : precede with an L

```
wchar_t wc;
wc = L 'A';
```

0x → hex

0 → oct

```
int hex = 0x80;
int oct = 012;
```

'a' → char → single quotes

"a" → string → double quotes

\n → New line

\b → Backspace (1 char deleted)

\f → form feed (page break) - pointer - eject current page.

\r → Carriage return (return to particular character - pointer) - left margin

\t → Horizontal tab

\" → double quote

\' → single quote

\\ → Backslash

\v → vertical tab

\a → alert (produces alert)

\? → question mark

\N → octal constant

\xN → Hexa decimal Constant.

\xA → Alternate form of \n

\xD → Carriage return

123 → 7B (Hex)

173 (oct)

printf ("%d %x %o", 123, 123, 123);

%x , %o

hex oct

---

operators:

* lvalue (object) = rvalue (expression)

Implicit Conversion

signed char → −127 to 127

char → 2^8

low type → high type ✓ → No problem

High type → Low type → Messy happens.

» x=y=z=10;

» x = x+10;

» x += 10

» x *= 10;

» x -= 10;

/ → Any remainder will be truncated!

% → Remainder of int division!

x = x+1; → ++x / x++

x = x-1; → --x / x--

Difference:

x = 10;
y = ++x;
x = 11, y = 10

x = 10;
y = x++;
x = 11, y = 11

Precedance

Highest ++ --

− (unary)

* / %

Lowest + −

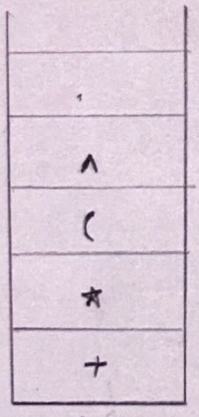
] same level precedence (left to right)

Infix to postfix

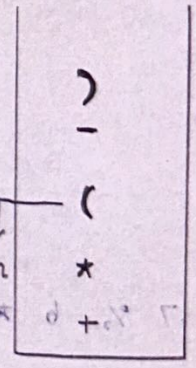
1. scan left to right
2. Operand → o/p it
3. Else
   1. precedence of scanned > precedence of operator in stack → push
   2. Else: pop all the operators from the stack which are ≥ to the precedence of scanned operator & push the scanned one. If you encounter paranthesis while popping stop there - then push the scanned one.
4. If the scanned one is an '(' → push it
5. If " ')' → pop the stack & o/p until '(' - discard '(' ')'
6. Repeat
7. print the o/p
8. pop & o/p from the stack until its not empty.

---

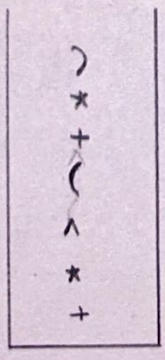
a+b*(c^d-e)^(f+g*h)-i



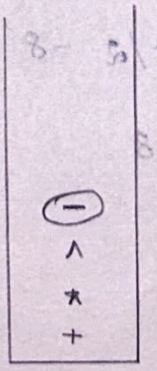
1



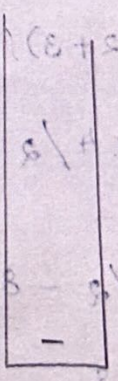
2



3



4



5

BODMAS
Braces
orders

execute:

10/2 * 5 — Left to right

(10/2) * 5 = 25.

() [] → . ++ --	left to right
! ~ ++ -- + - * * & (type) size_of	Right to left
* / %	left to right
+ -	left to right
<< >>	left to right
< <= >>=	left to right
== !=	left to right
&	//
^	//

Postfix precedence greater than prefix

a = fun() + fun1();

"undefined behaviour"

"Compiler dependent"

which fun/fun1 evaluated first?

3 * 4 % 5 / 2 = 12 % 5 / 2 = 2/2 = 1

3 * (4 % 5)/2 = 3 * 2 = 6

3 * 4 + 5 * 6 = 12 + 30 = 42

1) Parentheses
2) * % /
3) + -

17 % 6 * 4 / 2 + (2+3) * 4/2 - 8

17 % 6 * 4/2 + 5 * 4/2 - 8

5 * 4/2 + 5 * 4/2 - 8

10 + 5 * 4/2 - 8

10 + 20/2 - 8

10 + 10 - 8 = 12

```
int i = 5;
int var = sizeof(i++);
printf("%d %d", i, var);
```

Bracket `i++`

5 4

'Not evaluated i++'

C 99

array type (sizeof) → evaluated

otherwise: not evaluated

```
a = 1;      c = ++a || b++;
b = 1;      d = b-- && --a;
```

--, ++ (postfix) — R←L

++, -- (prefix) — R←L

&& ||

```
c = ++a || 2
  = 2 || 1   (after completion ↑)
  = 1
```

a = 2, b = 2 → After completion of expression.

```
d = b-- && --a;
  = 2 && 1
  = 1
```

(B/w &&, || → so no need for Associativity)

a = 1, b = 1 → after expression

'wrong'

why?: With || (whenever we encounter True - after that won't be implemented)

```
c = ++a || b++;
c = (2) T || (not simple)
```

a = 2, b = 1

```
d = b-- && --a;
d = 1 && 1
```

d = 1

a = 1, b = 0 → after implem

1 0 1 1

T || anything = T

100 + 200 / 10 − 3 * 10

100 + 20 − 30

120 − 30 = 90

5 + 2 * 10/1 − 3 + ([++4] − 5 * 2 − 1)

5 + 2 * 10/1 − 3 + (5 − 10 − 1)

5 + 20/1 − 3 + (−6)

5 + 20 − 3 − 6

25 − 9

16

x = 2;

y = ++x * ++x;

= 3 * 4

= 12

3

x = 4

x = 4

y = 12

x = 4 % 5 = = 4 % 2 + 3;

x = 4 = = 3

x = 0

a = ++y || ++x

b = --y && ++x

Printf ("%d", ++(a * b + 1));

↓

Compile error.

35 > 25 > 15

1 > 15

0

a = a+++a → 11

taken as

a = a ++ + a

Note!

* operand can't be a constant
* Always a variable (Increment o/p) can't be a expression!

25 − 15 > 35 < = 0

10 > 35 < = 0

→

0 < = 0

1

a = Printf ("Hello") + Printf ("Hi");

a → 7

a = a + ++a

= a + 6

= 6 + 6 = 12

a = 4, b = 5

b = a++ + a--

b = 4 + 5

b = 9

* a = 4, b = 5

*

a = 5

a = 4 → After evaluation!

a = a++ + a

= 5 + 5

a = 10

a = 10

a = 11

After end.

a = ++b + b--

a = 9 + 8

a = 18

First evaluate!

a = 4, b = 8

① a = 4, b = 9

② a = 4, b = 8 → After done

printf ("%d\n", a++); → 4 (print a = 4)
.... ++a); → 5. (print a = 4)

---

b = a++ + a++ + a++

= 1 + 2 + 2

b = 5 | a = 4

---

a = a++ + a--

a = 1 + 2

a = 3 → a = 2 After assigned — ∴ writing a — no need!

---

b = a++ + a++ + a++

= 1 + 2 + 3

b = 6

---

b = (++a); a → 2, b → 2 | b = a++ → b = 1, a = 2 | b = a++ + a++ → b = 1 + 2 → b = 3 | a = 3, b = 3

---

b = ++a + ++a

b = 3 + 3

b = 6, a = 3

2 → 3

'Case with pre increment'

---

b = ++a + a++

b = 3 + 2 = 5

2

2 ++a 3 + 3 a++ 2 (assign)

'pre increment: Assigned last'

b = (2 ++a 3 + 3 a++ 2 (assign)) + 4 ++a 4;

'Take as a combination eg two'

3 + 2 + 4 = 9

a = 4
b = 9

b = ++a + a++ + ++a;

b = a++ + ++a + ++a;

b = (a++ + ++a) + ++a;
(2, 3, 3, 1)

= (4) + ++a
4 (assign)

= 8

→ Refer this alone!

b = (a++ + ++a) + a++;
(2, 3, 4, 1, 3, 3)

= 4 + 3

b = 7

a = 4.

youtube.com/watch?v=XKPPLdkZIYs

++a, a, a++ → Wrong
(2, 2, 2, 3)

Printf("%d %d %d", ++a, a, a++)

← order.

3 2
++a, a, a++
3 → 3 1

∴ Now a = 3

Handle post/pre increment first.

a=0, b=0

→ a = b++ (1, 0) + b++ (2, 1)

a=1, b=2 ✓

→ a = ++b (1, 2) + ++b (2, 2) = 4

a=4, b=2 ✗

→ a = b++ (1, 0) + ++b (2, 2) = 2

a=2, b=2 ✓

a = ++b (1, 2) + b++ (2, 1) = 3

a=3, b=2 ✗

---

a=0, b=0

a = (b++ (1, 0) + b++ (2, 1)) + b++ (3)

= (1) + b++ (2)

a = = 3, b=3 ✓

---

a = (b++ (1, 0) + b++ (2, 1)) + ++b (3, 3)

+ → left to right associativity.

a=4, b=3 ✓

---

a = (b++ (1, 0) + ++b (2, 2)) b++ (3, 2)

= (2) + (2) (3)

a=4, b=3

---

a = (b++ (1, 0) + ++b (2, 2)) + ++b (3, 3)

a=5, b=3

$$a = \begin{pmatrix} 1 \\ ++b \\ 2 \end{pmatrix} \begin{pmatrix} 2 \\ b++ \\ 1 \end{pmatrix} \begin{matrix} 3 \\ ++b \\ 3 \end{matrix}$$

$a = 6, b = 3$

$$a = \underset{2}{\underset{1}{++b}} + \underset{2}{\underset{2}{++b}} \quad \underset{2}{\underset{3}{b++}} = 6. \qquad b = 3.$$

$$a = \begin{pmatrix} 1 \\ ++b \\ 2 \end{pmatrix} \begin{matrix} 2 \\ ++b \\ 2 \end{matrix} \begin{matrix} 3 \\ ++b \\ 3 \end{matrix}$$

$a = 7, b = 3.$

---

① $a = \underset{1}{b++} + \underset{2}{b++};$

$a = 0 + 1$

$\boxed{a = 1}$

② $a = ++b + ++b;$

$1 + 2$

$\boxed{a = 3}, b = 2$

$b = 2, a = 1.$

③ $a = \underset{1}{b++} + \underset{2}{++b};$

$0 \quad 2 = 2$

$(a = 2, b = 2)$

④ $a = \underset{1}{++b} + b++;$

$a = 1 + 1$

$a = 2, b = 2$

---

① $a = \underset{1}{b++} + \underset{2}{b++} + \underset{3}{b++}$

$0 \quad 1 \quad 2 = 3$

$a = 3, b = 3$

② $a = \underset{1}{b++} + \underset{2}{b++} + \underset{3}{++b};$

$0 \quad 1 \quad 3$

$a = 4, b = 3$

③ $a = \underset{1}{b++} + \underset{2}{++b} + \underset{3}{b++};$

$0 \quad 2 \quad 2 = 4$

$a = 4, b = 3.$

④ $a = \underset{1}{b++} + \underset{2}{++b} + \underset{3}{++b};$

$0 \quad 2 \quad 3 = 5$

$a = 5, b = 3$

⑤ $a = \underset{1}{++b} + \underset{2}{b++} + \underset{3}{++b};$

$1 \quad 1 \quad 3 = 5$

$a = 5, b = 3.$

6) `a = ++b + ++b + b++;` (1, 2, 2 = 5) — a=5, b=3

7) `a = ++b + ++b + ++b;` (1, 2, 3 = 6) — a=6, b=3.

8) `a = ++b + b++ + b++;` (1, 1, 2 = 4) — a=4, b=3

Relational & logical

>> && (logical)
|| (logical)
! (logical)

Relational

>, >=, <=, <,
==, !=

Precedence:

highest ↑
!
> >= < <=
== !=
&&
||
lowest

Bitwise operators

& - AND
| - OR
^ - NOR
~ - NOT
>> - Right shift
<< - left shift.

1 1 0 0 0 0 0 1 ⟶ 193
0 1 1 1 1 1 1 1 ⟶ 127
0 1 0 0 0 0 0 1 ⟶ 65.

int → 4 bytes (using bitwise: 10 bits enough! (1000 coordinates))

② Any number power of two: $(x \& (x-1)) == 0$ → power of 2 or not!

Then change MSB to 1, others to zero!

③ Swap: $x = x \wedge y$, $y = x \wedge y$, $x = x \wedge y$.

Single integer doesn't appear twice:

* exor every number with all elements of array
* exor a number with itself = 0.

Count zero: job done!

---

* Bitwise operations: faster!
* $(x << 3 + x) \rightarrow x * 9$
* Limited memory: Bits to the rescue.
* odd/even → $!(x \& 1)$ → True (even), False (odd)

```
0 1 1 1
0 0 0 1
0 0 0 1 → True
```

! True = false

'odd'

* Change to upper case → 'a' & '_'

(regardless of current casing)

* Lower case → 'A' | ' '

---

1 1 0 0 0 0 0 1 → A with parity

(&) 0 1 1 1 1 1 1 1

0 1 0 0 0 0 0 1 → A without parity.

1 0 0 0 0 0 0 0

(|) 0 0 0 0 0 0 1 1

1 0 0 0 0 0 1 1

0 1 1 1 1 1 1 1

(^) 0 1 1 1 1 0 0 0

0 0 0 0 0 1 1 1

Shift – Not rotate!

* Left shift 1 – same as multiply by 2

(zeros are brought in on the other end) – Not rotate (won't come to other end – shifted digits)

* Right shift 1 – divide by 2;

★ 2's complement (negative)

★ $7 << 1$: 00000111 (left shift 1) → 00001110

8 4 2 = 14. (mul by 2)

★ $16 >> 1$: 00010000 → 00001000 = 8 (div by 2)

① ②'s Complement – NOT then add by 1.

13 → 1.

1101	0110	0011	0001	0000
0001	0001	0001	0001	0001
1110	0000	0010	0010	0001
'carry' 1	'carry' 0	'carry' 1	'carry' 1	'carry' 0 → stop.

'Left & right shift can't be used for → (-)ve numbers.

n th bit (Set/Not).

101 → 2

0011 & 0001 → 0001 'Set'

1011 → 2 (2nd bit!)

0010 & 0001 → 0000 'Not set'

1111111 → 6

2 → 4
3 → 8
4 → 16
5 → 32

$2^5 = 32$

$\log_2 2^n = \log_2(\text{ans})$

$n \log 2 = \log(\text{ans})$

$n = \log_2(\text{ans})$

∴ (n << (bit number)) & 1

MSB → set / not (Find MSB digit, Right shift & 1).

* get n th bit → same as set/not
* MSB set/not → (Right shift by MSB pos) & 1
* LSB → (number) & 1 → Answer!

---

set

1 0 1 1 0 0
0 0 0 0 1 0 (OR)
1 0 1 1 1 0

done

Set n th bit: (1 << (n-1)) | num

Unset

unset (2nd bit)

1 0 1 1 1 0
~ 0 0 0 0 1 0 (n-1) times left shift 1

Toggle - exOR

1 0 1 1 1 0
(&) 1 1 1 1 0 1
1 0 1 1 0 0

'unset'

(exOR ^) 0 1 1 0
0 1 0 0
0 0 1 0

'Toggle'

---

1 0 1 1 1 = 7

2^0
2^1
2^2

1+2+4+16 = 23

171

(17 x 10)
9 x 10 + bit

1	1	1	0	1

1+0(2) 1+2 1+6 0+17 1+28 = 29 → Not correct
(bit)

Correct

From Left