* As size may greater than the sum of sizes of its members - good to use `sizeof`

* dynamically allocate memory - should use a statement like

```
struct s *p;
p = malloc (sizeof (struct s));
```

* Sizeof → compile time operation - (So all the size of any variable is known at compile time) → meaningful for unions (size = largest member size).

```
union u {
    char ch;
    int i;
    double f;
} u_var;
```

while writing code - we don't need to guess size

write code using sizeof

» sizeof (u_var) = 8 [union - must be large as its largest element].

---

which - not working

```
struct {
    int a;
    :
} var;

struct var variable;
```

→ Error

'no tag'

Solution!

```
typedef struct {
    :
} var;

var variable;
```

Perfect way!

```
struct tag {
    :
};

struct tag var;
```

typedef

* define new datatype - actually not creating a new data type - just defining a new name for an existing type -

* help: make machine dependable programs - portable.

* If you define - your own type name for each machine dependent programs - data type used by your program → then only typedef statements have to be changed - when compiling for new environment.

Adv: * Self documenting code - allow descriptive names.

typedef type newname;

typedef float balance;

float → valid data type

balance → new name.

* Not a replacement - Just an Alias.

* Recognize balance as float (alias)

```
balance over_due;
float
```

```
typedef balance overdraft;
```

alias for balance

* Use:

* make more readable - easier to port to a new machine

* Note: Alias - not creating a new physical type.

I/O console

* C language doesn't define any keywords that perform I/O
* Done through library functions: <stdio.h> — Both Console & file I/O func.
* Std. C doesn't define any functions: perform various screen control operations (cursor positioning), display graphics → they vary among machines
* doesn't define any functions – write a window/dialog box under windows
* Console I/O functions performs TTY based operations – O/P.
* most compilers – have their libraries screen control, graphics func – apply to specific environment.
* Just: (For portability): C doesn't have any graphics/IO functions

Keyboard: I/P, O/P: screen (more)

* operate on std I/P & std O/P. – Further: redirected to devices
* stdin, stdout → doesn't need to be operate on the console.

In addition, C++ includes I/O operators – not supported by C.

Read & write characters

* int getchar(void); → waits until a key is pressed
* int putchar(int c); → display char at cursor's position.

* int → Convertable to char (low order byte)
* getchar() → returns EOF (macro from <stdio.h> often equal to -1)
* putchar() → returns character written / EOF – in case of error.

Program: get char, putchar until (.)

Problem with getchar()

* For many compilers, getchar() → implemented in such a way that it buffers I/P until Enter is pressed.

"Line buffered I/P."

* getchar() → takes 1st char – remaining char in queue – which is annoying in interactive environments.

* The remaining i/p (buffered) will be used by upcoming part).

* plan accordingly! – program may not behave as expected!

Alternatives

* getchar() – I need only one char, remaining char need to be ignored – 'failed'!

* getche(), getch → conio.h → doesn't use any buffer (immediately returns after 1st char)

> * 'Most modern Compilers – don't have conio.h'

Let's create one!

* Use getc, clean buffer → getch/getche.()

Note: don't use getch/getche – instead use scanf/getchar/getc

↳ remove buffer!

Strings

* gets(), puts() → read/write string.

  * Stores the data at the address – returns pointer
  * Null termination – placed at the end.
  * gets can't return a carriage return.

<CR> → Control character | used to reset the device's position to the beginning of the line of text!

'\r'	'\b', '\n'

>> char *gets (char *str)

Caution: "Don't use"

* No boundary checking - possible + users may enter more characters
* must avoid in commercial code - use fgets() instead.
* fgets prevents array overrun.

```
int puts (const char *str);
```

* puts - recognises \t, \n like printf()
* call to puts() → requires less overhead than printf();

overhead: Computation time, memory, bandwidth/other resources

why: puts() → only o/p string - no formatting, no numbers, no conversions.

(used when no need for formatting) - fast!

* puts → returns non negative value if successful

EOF - failed.

eg: puts ("hello")

---

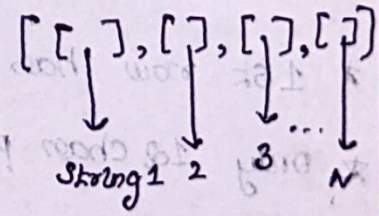
* getchar()
* getc()

] use buffer clear - write code

* putchar()
* fgets(), gets(), puts()

"simple dictionary"

"get word - give meaning"



* str[] → array

char *str[][40]

* str[][40] → str[] has pointer element pointing to size eg 40.

↳ "wrong"

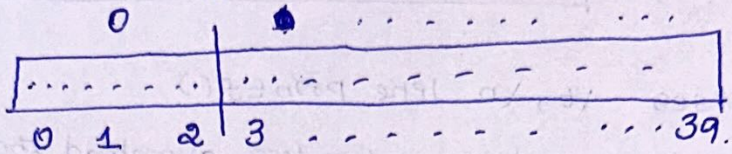
int * a[10];

array of 10 – having int pointers.

char * arr[ ][40] = {. . . . . };

Our case: 12 words.

* array of 10 having each 40 elements.



---

char * arr [ ] [40] = { "atlas", "book of maps", "car", "vehicle",

. . . . . };

* 6 words, 6 meaning – Totally 12 words.

* arr [ ] [40] → array of some rows

each row has 40 elements

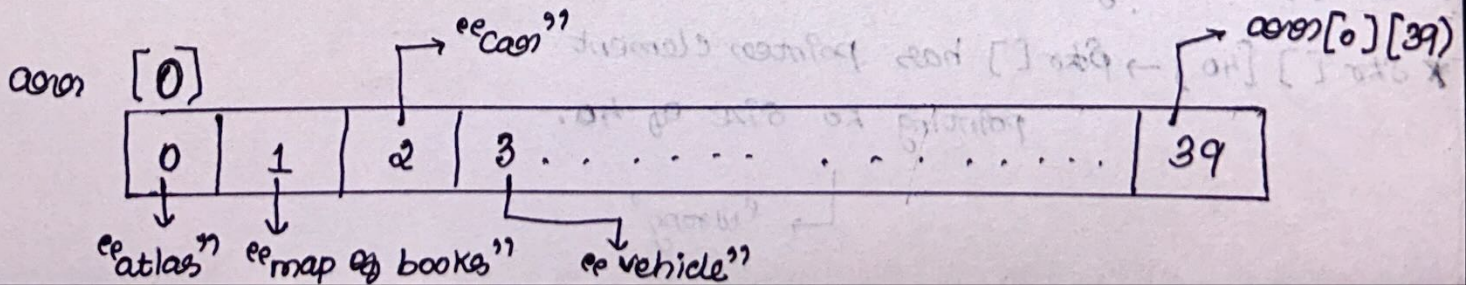
char(*arr [ ] [40] → array of some length rows

each row has 40 char pointers

→ {"atlas", ... }; → Stores string somewhere & returns char pointer!

... { 12 char pointers }

* 1st row has 40 char pointer capacity
* only 12 char pointers.
* So only 1 row enough.



No. rows = $\frac{\text{total size}}{\text{size of 1st row}}$

10 rows

10 cols.	

10 rows, (10 col for each row)

∴ Total size = 100

Size of [row 1] = 10

No. of rows = $\frac{100}{10}$

= 1

formatted i/p, o/p

printf()

int printf (const char * Control_String, ...)

* Returns number of characters - that will be printed on the screen
* Format specifiers - begins with % sign.

"Same number of arguments as format specifiers"

* matched - left to right.

%a → hexadecimal (C99), %c - char, %i → signed decimal int

%A → Cap " (C99), %d - decimal, %e - Scientific

Pg: 213.

%p, %n → no. of char written (Argument - Pointer) to an int.

%e/%E → double arg in scientific notation

%s → string

%u → unsigned

Scientific notation: x.ddddd E +/- yy

↳ %E → Uppercase E

%e → smaller case e. | %.G, %.g

printf ("%.g", f); → Scientific notation!

%.e → 1.000000e+00 ... 1.000000e+08

%g → 1 10 100 1000 10000 100000 1e+06

1e+07 1e+08

hexadecimal → %.x , octal → %.0

★ Display address %p

%.n Specifier

★ %.n → doesn't tells printf what to print, just loads, numbers of characters, that have been printed to the output. So we need to give an int address (variable) to store.

Printf →

```
printf ("this %n is a test \n", &count);
printf ("%d", count);
return 0;
```

Scanf → scanf ("%d-%n", &a, &b);

o/p : 12345

a = 12345

b = 5 (no. of digits)

Format specifier – width

%.f → 10.12304

%10f → 10.123040

%012 → 0001 0.123040

1) 10

%05 → pad with five zeros (less than) → 5 digits must

default padding : spaces

%05 → pad a number less than 5 → with 0's

%.0f → Totally 10 digits (including decimal point & after decimal point)

usually: %.f → 6 decimal places

%.f → 10.120400

%10f (10 places) → _10.120400 → 0 padded.

space

%012f → 12 digits must pad with zeros. (default: space).

%8d → 8 digits must pad with space.

Precision Specifier

* width: including decimals
* precision: only decimal places.

%10.4 → 10 chars wide, 4 decimal places.

* Precision specifier → maximum field length.
* %.5.7s → Atleast string of five with not exceeding seven characters long.

"long - truncated at the end"

printf("%7.9s\n", "Helloworld"); → Helloworl

min - 7chars, max: 9chars

Justifying outputs

* By default: all o/p is right justified - field width larger than data printed, data will be placed on the right edge of the field.

we can force it: left justified

↳ default: padded before digits (spaces) right too! justify

```
printf("..... \n");
printf("right - justified : %8d\n", 100);
printf("left - justified : %-8d\n", 100);
```

o/p

```
.....
right - justified :      100
left - justified : 100
```

%ld → long int → 220: pg

%hu → short unsigned int.

* and

* g, G, E, e → with # ensures - will be a decimal point even - if no decimal dig.
* x/X → with # causes - printed with 0x prefix
* o → with # → leading zero.

* other than these # - can't be applied to others'

* - placeholder - precision & width - in specifier

printf ("% * . * f", 10, 4, 1234.34);

Scanf

* general purpose - console i/p routine - read all built in types & automatically convert numbers into the proper internal format. It is much like the.

int scanf (const char * control string, ...);

* returns - no. of items scanned
* Failed : EOF

classification by scanf:

* format specifiers (preceded by %)
* white space characters
* Non-white-space characters

Numbers:

%d, %i - int
%f, %e, %g
%a - float

%X - hexadecimal] upper/lower case.
%o - octal

%a, %c, %d, %i, %e, f, g, o, s, x, p, n, u,

% [ ] → reads set of characters

%u → unsigned int (default: int)

scanf() → buffers.

Strings:

* %s → read chars until a white space [including space, newline]

"this is a test" → "this" → alone read.

Input address: %p

%n specifies → gets no. of chars counted

Scanset

1) scanf() → can process a scanset – get i/p as long as they are the characters defined by the scanset.

%[xyz] → read only xyz

* Read until: characters – not in characters set (scanset)
* Read until: end of array – defined
* Null terminated at the end.

scanf("%d%[abcdefg]%s", &i, str, str2)

i23abcdtye

i = 123

str = abcd

str2 = tye. (remaining: %s)

only one characters set? → str

Inverted set → except this

[^abc] → except abc

Range

[A-Z] → A to Z.

"Both upper & lower"

[A-Za-z]

"case sensitive"

% [a-z A-z]
↳ include space also.

Discard white spaces

* scanf() → stops at white spaces (and returns what it read)
* \t, \n; formfeed, vertical tab.
* One white-space character in the control string causes scanf() to read, but not store, any number (inc. zero) of white-space chars up to the first non-white space character.

Non-white space

* %d, %d → read an int, read & discard a comma
* %% → discard a % sign after reading.

Scanf: *pass address

Int → &count
%s → str [∵ array itself address].

Format modifiers

* max length modifier: scanf("%20s", str); lf → double.

∴ Can use used with %f, e, g, d, i, o, u, x, n

1 modifier

* used with c and s format codes – as long as compiler has wide-char features.

Suppress Input

* Tell scanf – Read a field – Don't assign it (By *)

scanf("%d%*c%d", &x, &y);

10, 10
↳ ignores comma

'C - File System'

* C - originally implemented for Unix OS – Unix like I/O system.
* C I/O provides abstraction b/w the programmer & device.

> Abstraction is called : Stream
> Device : File

Streams: C file systems work with a variety of devices – including terminals, disk drives, tape drives – each device different, buffered, file system transforms each into a logical device called a stream. (All streams behave similarly) – device independent, same function – can write – disk, write other devices like Console.

Stream → Text (sequence of char), certain char transformation can occur (based on environment using)
Stream → Binary

eg: Transformation \n → Converted to carriage return / linefeed.
Transformation: number of char read may not be same as no. of char written.

Binary stream: bytes – no translation (no. of bytes received = written) – null bytes may be appended to pad the info.

In C File → disk file → Anything
File → Terminal → Anything

* Associate stream with file using open
* Not all files have same capabilities (disk file: support random access, pointers can't)

> 'All files are not same but streams are'

When closed: files associated from a specific stream are written to the external device. (Flushing the stream): guarantees: no info is accidentally lost in the disk buffer. All files are closed by default when main() ends.

> Files not closed: when program ends abnormally. (Crash, abort())

I/o system automatically converts raw i/p to easily managed stream.

Filesystem

<stdio.h> → prototypes for I/o functions (defines three types)
- → size_t
- → fpos_t
- → FILE

* size_t → some variety of unsigned int
* <stdio.h> → several macros (NULL, EOF etc...)
* EOF often −1 (returned value when an i/p func tries to read past the end of the file.

FOPEN_MAX → defines int that can be opened at any one time

fseek() → random access on a file.

File pointer

* Common thread writes C I/o system. File pointer is a pointer to a structure of type FILE.

* points to info about various things of a file (name, status, current position)
* Read/write files → "use pointers"

```
FILE *fp;
```

Opening a file:

→ FILE *fopen (const char *filename, const char *mode);

feof()	fopen()	fgets()
ferror()	fclose()	fputs()
rewind()	putc()	fseek() → Seeks a specified byte
remove()	fputc()	ftell() → Returns current file position
fflush()	getc()	fprintf()
	fgetc()	fscanf()

open

```
FILE *fp
fp = fopen("test.txt", "w");
```

r - open, read	rb - open, read binary
w - create, write	wb - create, write
a - append	

(pg: 234)

r+b = rb+

r - read
w - write
a - Append to a text file
rb - open a binary file for reading
wb - write "
ab - append "
r+ - read/write
w+ - create/ read/ write

* fopen → NULL pointer if failed to open

In text mode: newline → \n, \r proceeds Carriage return

Not happens in binary mode.

FOPEN_MAX → Check Compiler (Atleast 8)

* fclose() → closes a stream - opened by fopen
* flushes - writes buffers & does a OS level close on the file.
* Failure to close a stream:
  * Losing data
  * destroyed files
  * Intermitten error
* fclose() → frees the file control block associated with stream Available for reuse.

∴ Limit (no. of files open at a time): close, reuse!

Eof → any error.

fclose() → fail only
* Disc removed prematurely
* Not enough space.

write a character

*putc(), fputc()

↳ Usually implemented with a MACRO?

Why two? : Support older versions.

* putc() → writes characters to the opened file

```
int putc(int ch, FILE *fp);
```

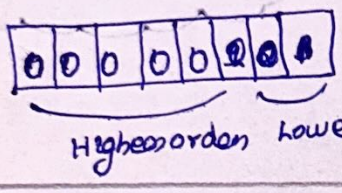
* Lower order byte is return
* Failed → returns EOF

Read a character

getc(), fgetc()

```
int getc(FILE *fp);
```

* character is contained in low-order byte (unless Error: high order: 0)



* Returns EOF → if end of file is reached.

"r" → don't create (file absent : EOF)

"w" → create (file absent : create).

fopen()
getc()
putc()
fclose()

get keyboard characters, write in file (until - dollar sign)

Simple text reader

textRead.c name (get from cmd)

Read

* getc() → returns EOF at the end (testing the value of getc() may not be the best way to determine - end of the file -
  * Text, Binary - Supported by C I/O
  * may read -1 in binary (end!)
  * getc() → also returns EOF when it is failed

'-1' - doesn't denote what happened

int feof (FILE *fp);

true → if end is reached

Otherwise → zero!

while (!feof(fp)) ch = getc(fp);

* No i/p argument
* file can't be opened — Source / destination

Copy from Source to destination

Strings – fputs(), fgets()

* They read/write strings

int fputs (const char *str, FILE *fp);

char *fgets (char *str, int length, FILE *fp);

* fputs → failed → EOF
* fgets() → reads string from the stream until : newline char / length − 1 (specified). – newline : part of the string (unlike gets())
*

Note: fgets() → also takes \n

Exercise: Read, display using fgets, fputs

Rewind()

* Reset all the file position to the beginning the specified file.

void rewind (FILE *fp);

ferror()

* determines whether a file operation has produced an error

int ferror (FILE *fp);

Erase files

remove() → erases specified file.

int remove (const char * file name);

* Returns: If successful - else non-zero value.

*. ASK - can I delete - yes - delete

Flushing a stream: flush contents of an o/p stream

int fflush (FILE *fp);

* writes contents of any buffered data to fp.
* fflush() ⟶ without fp ⟶ flush all files
* fflush() ⟶ zero (success), otherwise EOF.

fread(), fwrite()

* only 1 byte - any data

size_t fread (void *buffer, size_t num_bytes, size_t count, FILE *fp);

size_t fwrite (const void *buffer, size_t num_bytes, size_t count, FILE *fp);

* buffer - pointer to a reign of memory - receive data from the file (fread())
* buffer - fwrite() - info to be written to the file
* Count - how many items written / read - each item - being num_bytes bytes in length.
* fp - pointer to a prev. opened stream.
* fread returns number of items read, fwrite ⟶ no. of items written.
* This value will equal count unless an error occurs.

↓

Binary data file!

Mailing list: (structure!) - update!

* write large amount of data using fread() & fwrite()
* Enhance mailing list program - addresses need to be stored in a file.

```
struct struct_type {
    float balance;
    char name[80];
} cust;
```

```
fwrite(&cust, sizeof(struct struct_type), 1, fp);
```

* Note: I can't directly write a structure file : using fgets...
* Note: But I can write in binary - fwrite, fread.

* Aim: Just load files data : structures [Already stored]
* fread → to read

fread(&addr_list[i], sizeof(struct_addr), 1, fp) → 1 (success)
↳ else: unsuccess!

* fSeek() - Perform random read & write using fseek - seks file position indicator.

int fseek(FILE *fp, long int numbytes, int origin);
↓ no. of bytes from origin ↓ Macros

Beginning of File → SEEK_SET
Current Position → SEEK_CUR
End of file → SEEK_END

Zero - Successful
Non-Zero - Unsuccessful

* Seeks to & display the specified byte in specified file.

"Byte Search"

Say: 0010 1100
Search with all bytes

Search an int: (Not like that)

int fseek(FILE *fp, int offset, int whence/origin);

fseek(fp, 3, SEEK_SET);
↳ returns pointer pointing 3 bytes from starting.

File

HELLO WORLD
↑ ↑ → returned (3 points ahead).

Current position of a file

ftell()

* long int ftell (FILE *fp)
* Failure → returns −1.

* Many times: need to use random access on binary files – text files – character translations (encoding) performed – so there may not be a direct correspondence (need to decode – else useless) – with .txt file.

* we majorly use ftell(), then using ftell() returned value to fSeek()'

txt file can be opened as a binary file

---

fprintf, fscanf

```
int fprintf (FILE *fp, const char * control_string, ...);
int fscanf (FILE *fp, const char * control_string, ...);
```

Read from file:

```
fscanf (fp, "%s %d", s, &t);
```

write in terminal:

```
fprintf (stdin, "%d %s", t, s);
fscanf (stdin, "%s %d", s, &t);  → from keyboard.
```

Job!

'Read input – write in file'

* fprintf(), fscanf() → easiest way to write & read assorted data to desk files. [Not most efficient].
* formatted ASCII is written, overhead is incurred → (excess time/ memory/ computation) with each call.

for speed & size of file: use fread(), fwrite()

* As low efficient - don't use fscanf, fprintf (many others out there)

putchar()
getchar()
fgets()

Console : stdin/out

Redirect the standard streams

TEST > OUTPUT

* execute TEST, write o/p of TEST in OUTPUT.

TEST < INPUT > OUTPUT

* direct stdin (INPUT) to TEST
* write output to (OUTPUT)
* redirect stream by using freopen()

```
FILE *freopen(const char *filename, const char *mode,
              FILE *stream);
```

* filename → pointer associated with the stream pointed to by stream. (opened by fopen())
* freopen() → another method (fopen()) → success: returns stream
  failure: NULL

Preprocessor & Comments

* Can include various instructions to the compiler - preprocessor directive - expand scope!

Preprocessor

#define	#error	#ifndef	#line
#elif	#else	#include	#pragma
#endif	#ifdef	#if	#undef

All pre processor directives → # sign

1 line - 1 Preprocessor directive

1. #include <stdio.h>
2. #include <stdlib.h>

#define

* defines identifier & character sequence [used for identifiers]
* Identifier (alias) – macro name – Replacement process – macro replacement

#define macro-name char-sequence	No Semicolon (note)

* Any no. of spaces b/w identifier – but Char seq terminated by next line.

```
#define LEFT 1
#define RIGHT 0
```

→ can use LEFT & RIGHT in all functions of that .c file.

Use macros name to define others

```
#define ONE 1
#define TWO ONE+ONE
#define THREE ONE+TWO
```

eg:

```
#define EMS "standard error on input \n"
:
printf (EMS);
```

Quotes must (else: No effect) Just error!

Longer than one line!

* put a backslash

```
#define LONG_STRING "this is a very \
long string"
```

* Convention: Uppercase – So easy to identify as macro
* Practice: All #define – put in a separate file.

include that file as header file

* Macros – used to define 'magic numbers' – may have a programs – defines array & has several routines that access that array. Instead of constants, let's define – then recompile!

#define MAX_SIZE 100

In array:

float balance [MAX_SIZE]

for (i=0; i < MAX_Size; i++)

Functions like macros

#define ABS(a) (a) < 0 ? -(a) : (a)

* powerful: macro name can have arguments.
* This form: function like macro!

Abs (-1) ⟶ -1

* a must be enclosed with parentheses

If no parantheses:

Abs(a) a < 0 ? -a : a

Abs(10-20) 10-20 < 0 ? -10-20 : 10-20

(our need:) 10-20 < 0 ? -(10-20) : 10-20

Result: Wrong result

Use:

* Execution speed high – no function overhead.
* C99 ⟶ Inline Code!

#define ABS(a) (a) > 0 ? "positive" : "negative"

(Returns string (or assigns string to ABS)

#error:

* #error directive forces the compiler to stop Compilation.
* Used for debugging!

#error error message ⟶ ee ""

#include

* Tells compiler to read another source file

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

* Read & Compile - library functions.

C99 → 15 levels of nesting available.

* < > → Searched for in a manner defined by the creator (compiler).
* quotes → Another implementation defined manner (search current directory) - If filename not found → In CD → do < > search.

Not necessary (No rule).

* Convention - use < > → headers
* ee "" → including files at hand.

Conditional Compilation - 'Commercial Software house - customize program'

#if, #else, #elif, #endif

* Allows conditionally include portions of the code.

```
#if Constant-expression
    Statement sequence
#endif
```

if, else] Normal: will be compiles - executed according to condition.

#if → Boolean at the Compilation! pg: 267

```
#if MAX > 99
    printf("ee..");
#endif
```

```
#if MAX > 99
#else
#endif
```

```
#if
#elif
#elif
:
#endif
```

→ c99 → 63 levels of nesting allowed

Note: endif → must!

#ifdef and #ifndef

```
#ifdef macro-name
    statement-sequence
#endif
```

Another method of conditional compilation!

#ifndef → Not defined!

#undef: Removes previously defined definition of the macro

```
#define LEN 100
#define WIDTH 100
#undef LEN ——→ Now LEN is not defined!
```

use: localize definition scope!

Using defined

#ifdef → determine whether a macro is defined / not!

* defined → compile time operator

`defined macro_name` → used with #if

```
#if defined MYFILE   /   #ifdef MYFILE
```

#line → `__LINE__` / `__FILE__`] predefined identifiers in a file.

* `__FILE__` identifier is a string that has the name source file being compiled.

```
#line number "filename"
```

* number → any +ve int becomes new value of `__LINE`

Pg: 271

→ #Pragma → pre defined implementation directive – allows various instruction to be given to the compiler. – program execution tracing.

and ## preprocessor operators

* used with #define statements.

* stringize operator – turns argument in to a quote string

#define mkStr(s) #s

printf(mkStr(I like C)); ⟶ printf("I like C");

>> ## Concatenates two strings

```
#include <stdio.h>
#define concat(a,b) a## b
printf("%d", concat(x,y)); ⟶ printf("%d", xy);
```

xy=10

Predefined MACROS: (5 built-in)

1) __LINE__] #line [current line number, filename]
2) __FILE__]
3) __DATE__ → month/day/year string
4) __TIME__ → hour:min:sec string
5) __STDC__ → 1 (Standard c) else not zero.

__STDC_HOSTED__ → 1 (OS is present else 0)

__STDC_VERSION__ → 199901 (version)

/* */ // → Anywhere but don't bifurcate

* Keywords / identifiers.

* C99 features added

* Compound literals
* Flexible array structure members.
* Designated initializers
* inline
* restrict
* _Bool
* _Complex
* _Imaginary
* Variable length arrays
* Complex arithmetic, //
* Long long int; preprocessor
* Intermixed code & data
* for (int i)
  decl inside for

* Removed: implicit int (not int is assumed) — No implicit function definition
* Pg: 281.

Restrict — qualified pointers

* restrict type qualify — only one direct pointer
* second pointer based on first [optimize certain routines]

→ void *memcpy (void *str1, const void *str2, size_t size);

→ object pointed to by str1 & str2 → overlap [memcpy →

ensures — no overlap (not same object).

Inline

* inline — optimize calls to the function (function's code — expanded in line) rather than called → (request to the compiler).

supported by C++.

```
inline int max (int a, int b)
{
    return a>b ? a : b
}
```

```
printf ("%d", max(x,y));
```

Execution (expanded)

```
printf ("%d", x>y ? x:y);
```

* Runs faster, avoid overhead

_Bool, _Complex, _Imaginary datatypes.

long long int → $-(2^{63}-1)$ to $(2^{63}-1)$

unsigned → $2^{64}-1$

Array enhancement

* variable length array – but only local array can be done so! (Pg: 285).

Type qualifier in array declaration

int f(char str[static 80]) → Array contains atleast 80 no. of elements. (guaranteed).

* volatile
* restrict
* Const.

Interspersed code & declaration

```
int i;
i = 10;
int j;
j = i;
```

declare anywhere!

preprocessor changes

#define MyMax(...) max(__VA_ARGS)

↳ determines where arg must be substituted

MyMax(a,b) → max(a,b);

#define Compare(Compfunc, ...) Compfunc (__VA_ARGS__)

```
Compare(strcmp, "one", "two");
strcmp("one", "two");
```

* more macros, pragmas

Compound literal – array, structure, union

```
double *fp = (double []) {1.0, 2.0, 3.0};
```

pointer to a double → points to first 3 element array of double!

Flexible:

```
int a[10] = { [0] = 100, [3] = 200 };

struct mystruct {
    int a;
    int b;
    int c;
} ob = { .c = 30, .a = 10 };
```

New libraries

__func__ → name (as a string) of the function!

* Increased translation limit
* No implicit int
* No implicit function declarations — restriction on return!
* Extented int types

C standard library [compiler, linker]

* link libraries, headers.
* linker → combine (links) — various piece of object code
  → load objects (seperate compilation, single linking)

```
int count;
void display(void);
int main()
{
    display();
}
```

```
#include<stdio.h>
extern int count;
void display(void)
{
    printf("%d", count);
}
```

Substitute placeholder for display()

Relocatable (vs) Absolute:

* modern environment — o/p of a linker - relocatable (run in any available memory reign large enough to hold it — not fixed)!

→ overlay linker: create overlays — store in a disk file, execute whenever needed.

<stdio.h>

* dynamic linking - remains in a seperate file! (windows)
* C library - not contained in a dynamic library
* In windows API - stored in DLLs.

Library file (.lib) object file:

* object file - entirely becomes part of .exe file (whether code used / not)
* library - only necessary cases.

→ 25 headers

3 files:

LINK F1 F2 F3 LIBC (std. library name)

I/o Functions

1) clearerr → resets error flag associated with the stream
2) fclose()
3) feof()
4) ferror() → check for error!
5) fflush()
6) fgetc()
7) fgetpos() → file position indicator
8) fgets()
9) fopen() → 318
10) fprintf()
11) fputc()
12) fputs()
13) fread()
14) freopen()
15) fscanf()

* fseek()
* fsetpos()
* ftell() - current value of file position.
* fwrite()
* getc()
* getchar()
* gets()
* remove() → erases file
* rename() → change name of the file
* rewind() → pointer to starting line
* scanf() - 343 (scanset)
* perror → maps value of global variable errno
* printf()
* putc()
* putchar()
* puts()

Format modifiers by C99

* setbuf → (if null - buffer will be turned off) - 345
* setvbuf → programmer can specify buffer!

* snprintf() → like sprintf
* sprintf() → o/p is put in to the array pointed to by buf instead of stdout
* sscanf()
* tmpfile() – open temporary binary file (read/write)
* tmpnam() → unique file name.
* ungetc() → returns low order bytes
* vprintf, vfprintf, vsprintf, vsnprintf → argument list pointer.
* vscanf, vfscanf, vsscanf.

String & character functions – 354.

isalnum()
isalpha()
isblank() – space, horizontal tab
iscntrl() → Control character
isdigit()
isgraph() – printable char other than space.
islower()
isprint()
ispunct() → punctuation char (all printing char neither alpha numeric – nor space)
isspace() – white space, \t, \v, \r, \b,
isupper()
isxdigit → hexadecimal,
memchr() → Searches array pointed to by buffer for the first occurence of ch.
memcmp() → Compares!
memcpy()
memmove()

memset() – Copy lower-order byte
Strcat() → 368
Strchr() → first occurence of low order byte!
Strcmp()
Strcoll()
Strcpy() → 372
Strcspn()
Strerror() → String value asso. with errnum
Strlen()
Strncat()
Strncmp()
Strncpy()
Strpbrk()
Strrchr()
Strspn()
Strstr()
tolower()
toupper()

Math: macro — 384

INFINITY

int isinf()	int isgreater()	int islessequal(a,b)
int isinf()	int isgreaterequal()	int islessgreater()
int isnan()	int isless()	int isunordered()

Pg: 386

acos(), acosh(), asin(), asinh(), atan(), atanh(), atan2(),
cbrt(), ceil(), fabs(), cos, cosh(), erf → error function, exp2, exp,
fabs(), floor, fdim, log, log10, log2, round(), trunc(), nearbyint()

Time, date & localization functions

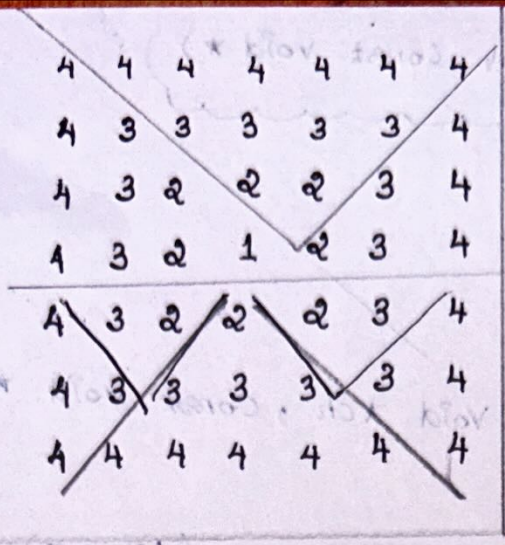
*asctime()	mktime()
* clock()	Setlocale
* ctime()	strftime
* difftime()	time
* gmtime	
* localeconv	
* localtime	

Dynamic allocation

→ calloc() → void *calloc (size_t num, size_t size);
→ Allocates & initializes with 0 (null pointer: not enough space)
→ free()
→ malloc() → void *malloc (size_t size);
→ void *realloc (void *ptr, size_t size);

Utility functions

abort() – abnormal termination
abs()
assert() – check condition, abort!
atexit → normal program termination.



	-3	-2	-1	0	+1	+2	+3
+3	4	4	4	4	4	4	4
+2	3	2	2	2			
+1							
0							
-1							
-2							
-3							

```
for(i=n-1; i>=1-n; i--){
    for (j=1-n; j<=n-1; j++){
        min = abs(i) > abs(j) ? i : j;
        min = abs(min) - 1;
        printf("%d", min+2);
    }
    printf("\n");
}
return 0;
```

---

* atof → string to double (number must be at the beginning) 100.12abcd
* atoi → string to int
* atol → string to long int
* atoll → long long int
* bsearch → binary search

```
void *bsearch (const void *key, const void *base,
               size_t num, size_t size,
               int (*compare) (const void *, const void *));
```

* Sorted array.
  * key – any character
  * base – string / array
  * size of array/string
  * size of character key
  * search function: comp!

* Key not found – NULL pointer

int (*compare) (const void *, const void *)

arg1, 2 → same → 0

arg 1 > arg 2 → >0

arg 1 < arg2 → <0

```
int comp (const void *ch, const void * ch1)
{
  return (*((char *) ch) - *((char*) ch1))
}
```

---

div(int numerator, int denominator);

exit() → immediate, normal termination

getenv() → return environment info.

qsort()

```
void qsort (void *buf, size_t num, size_t size,
            int (*compare) (const void *, const void *));
```

---

rand() → zero to 32,767

strtod → string to double

strof → string to float

strol → string to long.

strtold → string to long double

strtoll → " long long

strtoul → unsigned long

strtoull → " " long

Widecharacter functions: <wctype.h> — equivalent widetype to char type.

pg: 472

p = strdup("hello") → p → "hello\0"

Library features

* Complex, Imaginary pg: 484.
* <Std bool.h>
* <StdInt.h>

Sorting & Searching: → Exchange
→ Selection
→ Insertion

Exchange: Cards: exchange continuously – until ordered.

Selection: Spread the cards – select lowest value – take it out – Repeat.

Insertion: hold all the cards – Present that card in its position!
(correct position)

---

*