return

* Returns to the place whose ft is called (from).
* Any no. of return statements possible - first one - exits function & return.

Return value

* Except void - all type has return value
* In C89 - if return not used - garbage will be returned!
* In C99 - must specify return value. - when it reaches end of function ({ }) → returns garbage. → should be avoided.

"As long as a function is non-void return type we can use it in expression"

```
x = power(y);
if (max(x,y) > 100)
```
↳ expression.

---

functions (3 types)

→ Computations (perform operations) — also - pure function
eg: sin(), sqrt()

→ manipulate files & return a values - indicates success / failure of that manipulation.
eg: fclose() → closes file (return 0 - Success, EOF - failed)

→ No return value (explicitly) — function: strictly procedural & procedures no value.
eg: exit() → terminates a program.

return type → void

By returning (declaring) as void - Keep it from being used in an expression.

(prevent accidental misuse)

* printf() → returns number of characters written!
* If no assignment statement: Discarded the returned value!

Enter the number: 5 4 3 1 = 18

x = printf("%d"); → 18

Returning pointers

→ pointers: neither int – nor unsigned int – memory address of a certain type of data.

→ pointer arithmetic: is relative to the base type.

eg: incrementing int pointer → (4 bytes) – next int address

∴ length of diff. data types may differ

↓

Compiler must know what data pointing to.

So need to declaring returning type

although we say: all pointers are same.

Problem: returning int * when char * needed – creates problems.

need to return a generic pointer → void *

Convertible as per required (since void * has no type available)

eg: (string, char) – parameters

match: Return that address

No match: NULL address.

void function – explicitly return – don't return values

» early versions: no void type (so for void also – type: int) – just don't return.

main function

* Returns an int to the calling process, which is generally the OS
* Equivalent to exit() = returning from main.
* If no return from main → not known (undefined) – what is returned.

Automatically return 0 – mostly

Recursion: – Function call itself – recursive.

* Recursion: process of defining something in terms of itself (Circular definition)

  * For each call – new set of local variables – stack
  * Recursive call doesn't make new copy of the function – only values that it is operating.
  * As each recursive call return – old local var, parameters removed from the stack.

> "Telescope out & back"

* slower than iterative pgm. – Stack overrun – Crash.

Advantage → clearer, simple version of several algorithms
→ quicksort (difficult: Iterative).
→ helpful problems related with AI

Recursion → Base case
→ Recursive case

> Base case: To force convergence

Prototypes

* modern – functions must be declared before they are used. (prototype)
* C89 – added – use – strongly encouraged (but in C++ must)

> prototypes: enable compiler to provide stronger type checking.
> eg: As in Pascal.

* If prototypes given – compilers can find & report – any questionable type conversions b/w arguments used to call a function & the type of its parameters.

> "Compiler – catch differences b/w no. of arguments used to call a func & parameters defined in a function"

type func_name (type parm_name 1, ... type parm_name N);

eg: void swr_it (int *p);

definition also serves as prototype - If definition occurs prior to the functions first use in the program.

```
void f (int a, int b)
{
:
}
```

→ Serves as prototype (definition).

'No seperate prototype required'.

» In practice: Seperate prototypes are used!

In C++ → prototypes must

main() → only function that doesn't requires any prototype
Reason: It is the first one - begins → program-run

C (Important difference) C++

int f();

No parameter info given - it may have parameters and also may not have! → 'old style function declaration'

int f();

Empty parameter list means absence of parameters.

↓

No parameters - use void

eg: float f (void)

This tells C - no parameters.

↓

use of this in C++ - allowed - but redundant.

* function prototype - helps to trap bug - more debugging.
* mismatched arguments.

'convention - Just use prototypes'

Old style function declarations

* Before creation of function prototypes: still a need to convey: compiler in advance about return type of a function. — so proper code could be generated. (size - different).

↳ Accomplish using function declaration without any parameter declaration. (Prior).

Not in practice: But older codes may have it.

---

eg: double div(); → // old style.

```
double div (double num, double denom)
{
  return num/denom;
}
```

---

old style: tells - return type double

compiler — can generate code for calls to div()

eg: form: type_specifier function_name();

* Parameters: not listed in type declaration.
* But this style outmoded - shouldn't be used - incompatible with c++.

Considered as void?

Standard library function prototypes

* Any std. func lib using — must be prototyped → 'header'

<stdio.h> → definitions
→ prototypes.] for library function.

Variable length parameter lists.

* Specify a function with variable number of parameters.

eg: printf()

end declaration with three periods

» int func (int a, int b, ...);

Also used by function's definition also!

Rule

"variable number of parameters" – must have atleast one actual parameter.

eg: int func (...) // Illegal – error!

Implicit int rule

* In absence of any explicit type specifier – type int (assumed). – C89
* In C99 - eliminated! (not supported by C++)

old types (don't use now)

```
int f (void) {
}
```

old time ⟶

```
f(void) {
:
}
```

∴ default int (old times).

old style vs modern function style

modern form (C++)

```
float f(int a, int b, char ch)
{
:
}
```

"Recommended"

oldstyle

```
float f (a, b, ch)
int a, b;
char ch;
{ ... }
```

old style

- parameter list
- Actual parameter declaration.

```
type func_name(parm1...parmN)
type parm1;
type parm2;
:
type parm N;
```

Note: First just declare. Then type as mentioned below.

Inline Keyword

* Tell the compiler to optimize calls to the function - meaning function's code will be expanded inline, rather than called,
"only a request to the compiler - can be ignored"

→ Inline specifier - also supported by C++. ↳ place that code there (instead of calling)

Structures, Unions, Enumerations & typedef

Custom data type (5 types)

→ Structure (group variables under a single name - aggregate / compound / conglomerate data type

→ Union (same piece of memory - to be defined as two/more diff. types of variables)

→ bit-field - special type of structure / union element - allows easy access to individual bits.

→ Enumeration - list of named int constants.

→ typedef → define a new name for an existing type.

Structure

* Collection of variables - referenced under one name -
* Convenient - relate info together.
* Forms a template - that can be used to create structure objects (structre)

variables that make up structures → members / elements / fields

* Usually members of a structure are logically related.

eg: name, address → mail

declare:

```
struct addr
{
  char name[30], street[40]; city[20], state[3];
  unsigned long int zip; };
```

```
struct addr
{
    :
};
```

→ Terminated by semicolon: Structure declaration is a statement.

structure tag: addr → identifies particular data structure & its type specifier.

→ At this point: nothing is created! only form of the data – defined

→ declare a structure – defining an aggregate type, not a variable.

To declare variable

```
struct addr addr_info;
```

* declares variable addr_info of type addr
* Automatically allocates enough memory.

we can create one/more objects

```
struct addr {        → optional.
    char name[30];
    char street[40];
    :
} addr_info, binfo, cinfo;   → variables.
```

Note: each variable has its own copies of structure members.

---

what we need: variable tag. (Pg: 175

```
struct {
    char name[30];
    :
} addr_info;
```

either tag/ variable can be omitted but not both!

Access members

```
addr_info.zip = 12345;
```

object name — member name.

fgets(addr_info.name, 100, stdin); → gets string for a member.

```
struct {
    char name[30];
}
```

Assignment

```
int main(void) {
    struct {
        int a;
        int b;
    } x, y;
    x.a = 10;
    x.b = 11;
    y = x
```

→ assign one struct to another (same type)

```
printf(y.a) → 10
y.b → 11.
```

"Same type - using single assignment"

Arrays of structures

* .

Problems with fgets(), scanf

fgets stops → end of line
→ n-1 char are read
↳ \n read.

```
typedef struct
{
    char name[30];
    char roll[30];
    char dep[10];
    int year;
    char dob[30];
} student.
```

```
int main() {
    student stu1;
    fgets(stu1.name, 100, stdin);
    fgets(stu1.roll, 100, stdin);
    fgets(stu1.dep, 100, stdin);
    scanf("%d", &stu1.year); → read
    fgets(stu1.dob, 100, stdin); → skipped.
```

fine!

hari
iece
ece
2

done : Problem: dob not read!

Reason: Scanf reads an integer & leaves a newline character in the buffer. fgets() only reads newline character - further (not read)!

Solution

① scanf ("%d ", &—)
↳ space

② scanf ("%d\n", &—)
↳ \n

③ scanf ("%d", —);
getchar();] preferred!

Another problem

```
struct {
    char *name;
    char *dep;
    ;
} var;
```

fgets (name, 100, stdin);

now: name has not been initialized (no address - where to copy?)

Solution:

① Initialize, then use
② use char name [30]
char dep [30].

Array of structures

→ struct addr addr_list [100];

"creates 100 structures"

addr_list [2].zip
addr_list [1].zip

addr_list [1].name [0] = 'x';

Student [0].name [0] = 'y';

"Develop single mailing list program"

```
struct addr{
    char name[30];
    char street[40];
    char city[20];
    char state[3];
    unsigned long int zip;
} addr_list[MAX];
```

```
void init_list(void)
{
  register int t;
  for(t=0; t<MAX; ++t)
      addr_list[t].name[0] = '\0';
}
```

```
int menu_select(void)
{
  char s[80];
  int c;
  printf("1. Name \n 2. Delete\n
          3. List \n 4. Quit");
  do {
      printf("\n Enter your choice: ");
      gets(s);
      c = atoi(s);
  } while(c<0 || c>4);
  return c;
}
```

```
main()
    int main(void);
{
  char choice;
  init_list(); /* initialize struct array*/
  for(;;){
    choice = menu_select();
    switch(choice){
      case 1: enter();
          break;
      case 2: delete();
          break;
      case 3: list();
          break;
      case 4: exit(0);
    }
  }
  return 0;
}
```

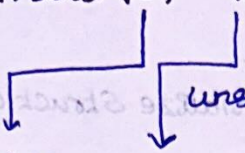
```
void enter(void)
{ int slot;
  char s[80];
  slot = find_free();
  if(slot==-1){
    printf("List Full");
    return;
  }
  printf("Enter name");
  gets(addr_list[slot].name);
  printf("Enter Street: ");
  gets(addr_list[slot].street);
```

- city
- state

zip - need to format

"mailing list.c"

```
fgets (s, 100, stdin);
strtoul (s, '\0', 10); → str to ul (unsigned long)!
```



own case: Global (Struct) — use directly without creating objects!

```
int find_free (void) {
    register int t;
    for (t=0; addr_list [t].name [0] && t<MAX; ++t);
    if (t==MAX) return -1;
    return t;
```

```
void delete (void)
{
  register int slot;
  char s[80];
  printf ("Enter record #: ");
  gets (s);
  slot = atoi (s);
  if (slot >=0 && slot < MAX)
      addr_list [slot].name [0] = '\0';
```

```
void list (void)
{
  register int t;
  for (t=0; t<MAX; ++t) {
    if (addr_list [t].name [0]) {
          printf ("%s \n", addr_list [t].name);
                                  .street);
                                  .city);
                                  .state);
                                  .zip);
    }
  }
  printf ("\n \n");
}
```

* mailing list - say (we can store 100 mails)

→ free space - enter a mail (place)

→ sent to deliver - free that space.

→ list() → list all mails

→ exit() → from program!

Analyze

fgets (array, 40, stdin) → hello
fgets (array, 40, stdin) → hi

] so new one stands! old one - not!

atoi → O/P (int)

---

name, street, door number, city, state, ZIP

Passing structures to functions

Pass a member → passing the value (Nothing different)

Say:

```
Struct
  {
   int a = 10;
  } b;
```

fun (b.a) (same as) fun (10)

"No difference"

---

* Its pass a struct - creates no difference.

```
Struct food
{
 char x;
  int y;
  float z;
  char a[10];
} mike;
```

```
func (mike.x);
func2 (mike.y);
func3 (mike.s);
fun4 (mike.a);
func (mike.s[2]);
```

Say: passed address.

```
func (&mike.x)
func2 (&mike.y);
```

& → precede structure name

* s → already specifies address!

* So (&) → not necessary.

```
func2 (mike.s);
```

Passing Entire Structures to Functions

* pass by value - doesn't affect actual structure passed as the arguments.
* type of the arg - must match parameter.

```
void f1(struct struct_type parm)
{
....
}
```

```
struct struct_type arg;
f1 (arg);
```

Error

```
struct struct_type{
int a,b;
char ch;
}
```

```
struct struct_type2 {
int a,b;
char ch;
};
```

Call

```
struct struct_type arg = {1,2,'a'};
f1(arg);
```

declare

```
void f1 (struct struct_type2
parm)
{
}
```

error: why: eventhough similar variable in both struct_type & struct_type2.

Struct - user defined type

* like int → struct_type as a type
* like that → struct_type2 another type

) different - error!

Structure pointer

Structure pointers:
* C allows to use pointers to structures.
* But some special aspects are there!

declare:

>> struct addr *addr_pointer; ⟶ without typedef

>> addr *addr_pointer ⟶ with typedef

use structure pointers

Uses → Pass by reference
→ create linked list, other dynamic DS rely on dynamic allocation.

why useful:

* local variable – stack (stored)
* structure (5 pts) → No problem/overhead
* array – overhead in stack. (since taking copy)
* solution: pass by reference – pointer! (only address is passed - very fast)

Advantages → modify directly
→ low storage, high speed.

eg:
```
struct bal {
float balance;
char name[80];
} person;
```

`struct bal *p;` → declare

Initialize

`p = &person;` [Same data type]

Access:

p → balance (dereference p access balance).

(*p).balance — Same as — p → balance.

→ Arrow operator

* Replacing (.) operator used after dereferencing a struct pointer.

display hours, minutes, seconds

```
#include <stdio.h>
#define DELAY 128000
struct my_time {
```

```
int hours, minutes, seconds;
};
void display (struct my_time *t)
{
  printf ("%02d:", t->hours);
  printf ("%02d:", t->minutes);
  printf ("%02d\n", t->seconds);
}
```

```
void delay (void)
{
  long int t;
  /* change this as needed */
  for (t=1; t < DELAY; ++t);
}
```

'Based on computer' — too fast — more loops!

```
void update (struct my_time *t)
{
  t->seconds++;
  if (t->seconds == 60) {
      t->seconds = 0;
      t->minutes++;
  }
  if (t->minutes == 60){
      t->minutes = 0;
      t->hours++;
  }
  if (t->hours == 24)
      t->hours = 0;
  delay();
}
```

```
int main (void)
{
  struct my_time systime;
  systime.hours = 0;
  systime.minutes = 0;
  systime.seconds = 0;
  for (;;) {
      update (&systime);
      display (&systime);
  }
  return 0;
}
```

'even running — clock'

Some problems

```
typedef struct {
   int a, b, c;
} vars;
void printvars (vars *num)
{
  printf ("%d %d %d\n", num->a, num->b, num->c);
}
```

```
int main() {
   vars num = {1, 2, 3}, *num1;
   num1 = &num;
   num1->c = 10;
   printvars (num1);
}
```

o/p: 1 2 10

Problem: Above programs - runs correctly!

Say:

```
int main() {
    Vaan *num;
    Vaan.a = 1;
    Vaan.b = 2;
    Vaan.c = 3;
    num = &Vaan;
    printVaan(num);
}
```

→ Looks like error!

why like int → Vaan → data type.
So we can't use it like a struct.

Rule: Can't use type name!

Use like this: (works)

```
struct myVaan {
    int a;
    int b;
    int c;
} Vaan;

int main() {
    struct myVaan *num;
    num = &Vaan;
    num → a = 1;
    num → b = 2;
    num → c = 3;
    printVaan(num);
}
```

Can't use

```
typedef struct myVaan {
    :
} Vaan;

int main() {
    myVaan *num;
    num = &Vaan;
    :
}
```

→ using typedef making Vaan also a type

① So don't use typedef

② As myVaan is type now! Can't use myVaan as struct object (use Vaan instead)!

Simple!

```
struct tag
{
    :
} Vaan;

int main() {
    struct tag num;
    num = &Vaan
}.
```

better

```
struct
{
:
} var;
```

```
main() {
    struct var ___ ;
    :
}
```

Arrays & Structures within structures.

* member - also can be arrays, structures (Aggregate types).

```
* struct x {
      int a[10][10];
      float b;
  } y;
```

y.a[3][7] → Access.

"A structure can be member of another structure"

"Nested structure"

```
struct emp {
    struct addr address;
    float wage;
} workers;
```

```
struct addr {
    char name[40];
    char street[30];
    :
} addr;
```

"Nested structure"

```
workers.address.zip = 93456;
```

members of each structures - referenced from outermost to innermost.

C89 → 15 levels possible
C99 → 63 levels possible

→ very rare!

Unions

* memory location shared by two/more variables.

union tag {
⋮
} union_variables

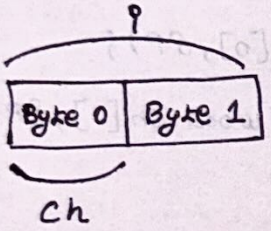
union u_type {
int i; → doesn't create any
char ch; variables
};

To do so

① union u_type myvar;

② union u_type {
⋮
} myvar;

Note: both i and ch share the same memory location (i → 4 bytes, ch → 2 bytes)



what happens: Compiler allocates memory to store largest data type!

eg: char, int → allocates 4 bytes.

Can use → dot
→ arrow } operators like structures.

cnvt.i = 10; → value

void func (union u_type *un) → Reference.
{
un → i = 10;
}

---

use

* Frequently - used when specialized type conversions - needed.
* Because data held in union - can be interpreted in fundamentally different ways.

eg: use union to manipulate the bytes having a double in order to alter its precision / perform.

unusual type of rounding.

eg: nonstandard type conversions

1) writing a short integer in a disk file.

(no function - to write a short int to a file).

Although: we can write any data to a file using fwrite() → it involves excessive overhead for such simple operation.

But easily done using union

```
union pw {
    short int i;
    char ch [2]
};
```

→ use pw

```
putw (short int num, FILE *fp)
{
    union pw word;
    word.i = num;
    putc (word.ch[0], fp);
    return (putc(word.ch[1], fp));
}
```

```
int main (void)
{
  FILE *fp;
  fp = fopen ("test.tmp", "wb+");
  if (fp == NULL) {
      printf ("can't open file.\n");
      exit(1);
  }
  putw (1025, fp);
  fclose (fp);
  return 0;
}
```

putc() → write each byte in the integer to a disk file one byte at a time.

```
union {
    int a;
    int b;
} var;
```

```
main()
{
    struct var var;
    var.a = 1;
    var.b = 11;
    Print (var.a, var.b)
```

→ Both 11

"overwrite" → Not! write seperately!

```
var.a = 11;
var.b = 12;
var.c = 'a';
```

```
printf (var.a, var.b, var.c);
```

97

```
var.a = 11;
var.b = 12;
var.c = 'a';
var.d = 111.002;
```

−996432413 −996432413 (?) (Symbol) 111.002000.

---

* Note: Union - used for writing non-standard type conversions (byte by byte) → give short, write using ch[0], [1]

Bit fields

* unlike - many languages → C - inbuilt feature - "Bit field" - allows to access a single bit.

* storage limited - store several boolean var in one byte.

* certain devices transmit status info encoded - into 1/more bits but within a byte

* certain encryption routines: need to access bits within bytes.

* Though these can achieved by bitwise operators - a bit field can be more structured (possibly efficient).

* bit field must be a member of structure / union

* defines how long, (in bits) - the field is to be.

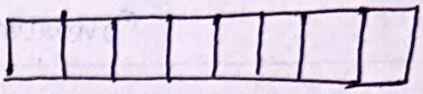
```
type name : length;
```

int, signed, unsigned (C99 - Bool)

Allowed bit-field types.

bit-field → often used when analyzing i/p from a hardware device.

eg: status of a serial comm. adapter might return byte status like



Bit	meaning
0	change in clear to send line
1	change in data-set-ready
2	Trailing edge detected
3	change in receive line
4	clear to send
5	Data-set-ready
6	Telephone ringing
7	Received signal.

```
struct status_type {
    unsigned delta_cts : 1;
    unsigned delta_dsr : 1;
    unsigned tr_edge   : 1;
    unsigned delta_rec : 1;
    unsigned cts       : 1;
    unsigned dsr       : 1;
    unsigned ring      : 1;
    unsigned rec_line  : 1;
} status;
```

→ only unsigned / signed / bool / int alone

type name : length;

(In bits)

```
status = get_port_status();
if (status.cts) printf("clear to send");
if (status.dsr) printf("data ready");
```

Assign a value

```
status.ring = 0;
```

Don't need to name each bit field

- Just reach the bits you want - skip others - if only cts, dsr bits'

```
struct status_type {
    unsigned :  4;
    unsigned cts: 1;
    unsigned dsr: 1;
} status;
```

bits after dsr - don't need to be specified - if they are not going to be used.

Normal to mix bit fields & variables

```
struct emp {
    struct addr address;
    float pay;
    unsigned lay_off: 1;      /* lay off on active */
    unsigned hourly : 1;      // hourly wage / not
    unsigned deductions: 3;   // IRS deduction - how much
                                 (next 3 bits).
```

* employee record → uses only 1 byte to hold 3 pieces of info.
* employee's status, salary, no. of deductions.

- without - bit field takes - 3 bytes'

Disadvantages: (Restriction)

* we can't take address of a bit-field
* Can't be arrayed.
* Can't know - (from machine to machine) - whether - the fields will run from left to right; → implies any code using bit fields may have some machine dependencies.
* other restrictions - may be imposed by various specific implementations.

* use of bit field: specify size (in bits) of structure/union members
* Idea: use memory efficiently (when we know: when group of fields never exceed a limit / within a small range).

```
struct date {
    unsigned int d;
    unsigned int m;
    unsigned int y;
};
```

```
struct date dt = {31, 12, 2014};
```

dt.d → 31
dt.m → 12
dt.y → 2014

12 bytes.

* 1 to 31 ($2^5 = 32$) → 5 bits enough
* 1 to 12 ($2^4 = 16$) → 4 bits enough

```
struct date {
    int d : 5;
    int m : 4;
    int y;
};
```

Now:

31/12/2014

o/p: −1/−4/2014

why?

00000	0011
1	1
00001	0100
−1	−4

31 → 11111 (MSB − 1 → took as negative)

12 → 1100 (MSB − 1 → took it as neg)

'solution take 1 bit extra'

Facts

1. A special unnamed field of size 0 is used to force alignment on next boundary.

eg:

```
struct test1 {
    unsigned int x : 5;
    unsigned int y : 8;
};
```

→ 4 bytes

reason by: 4 bytes?

Force alignment

```
struct test2 {
    unsigned int x : 5;
    unsigned int   : 0;
    unsigned int y : 8;
};
```

8 bytes

5 bits → 4 bytes (align)
8 bits → 4 bytes

2) pointers won't work - as they may not start from a byte boundary.

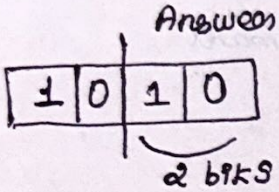
"we are dealing with bits"

3) Implementation defined to assign an out of range value to a bit field member.

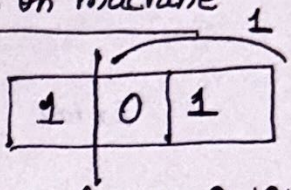
```
struct test {
    unsigned int x:2;
    "      y:2;
};
```

```
struct test t;
t.x = 5 (3 bits)
But limit 2 bits
```

Conversion happens - depends on machine



mydevice



In C++

* we can have static members in a structure/class → But bit fields can't be static

"error"

```
struct {
    static unsigned int x:5;
};
int main() { }
```

Array of bit fields – not allowed

```
struct test {
    unsigned int x[10]:5;
};
int main()
{
}
```

→ error

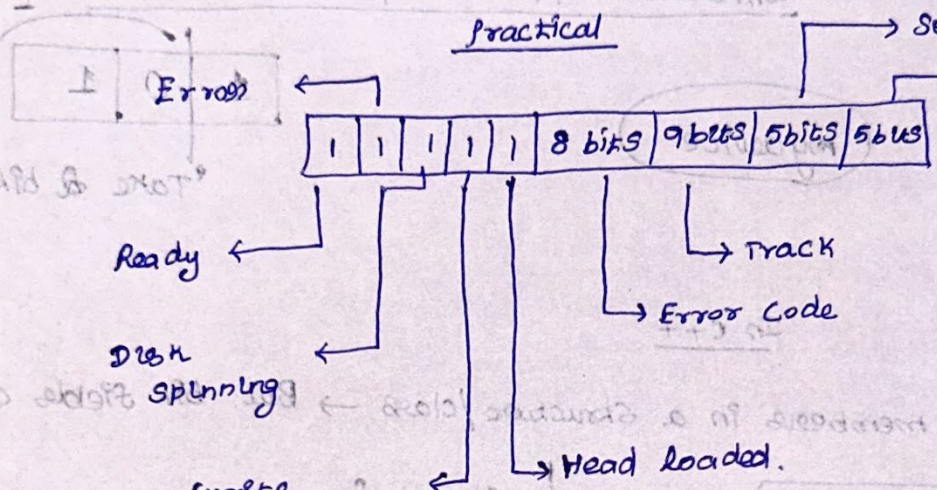
Applications: Limited storage! – go for bit field.

Device transmit status / info encoded in to bits – most efficient.

Encryption routines – need to access bits with in byte.

* only n lower bits will be assigned to an n bit number → can't take more than 16 (4 bits long)! – (Say!)
* Bit fields → converted to int for computation.
* Allowed to mix with member (normal).
* unsigned → important.

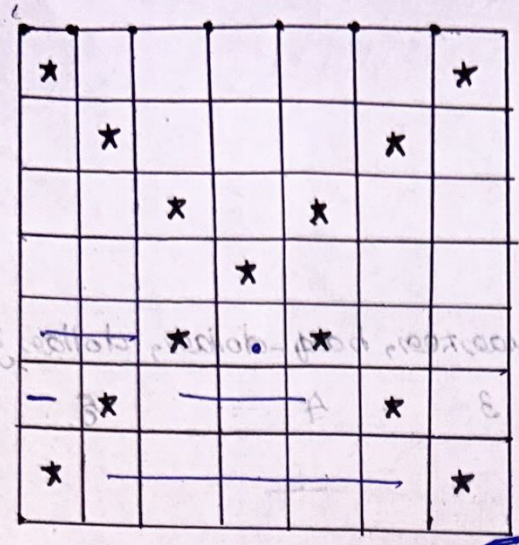
Practical

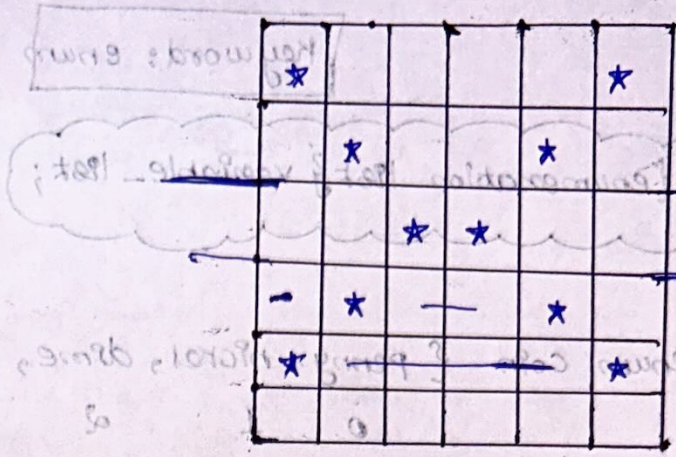



```
Struct {
unsigned ready : 1;
unsigned error_occured : 1;
unsigned disk_spinning : 1;
unsigned write_protection : 1;
unsigned head_loaded : 1;
unsigned error_code : 8;
unsigned track : 9;
unsigned sector : 5;
unsigned command : 5;
};
```

* microprocessor, MCU
* Low memory

"use it efficiently"





2, 4.

p =

q == 9 , q+9 == n-1

1, 3, 5

b/2 = 2 (but rev : 2)

③ a/2 = 3

a/2 = 3

0 to 3 (excluding)

0 to 2 (excluding)

Space: 2, 1, 0 | 1, 3, 5

space : 1, 0.

2, 4

Start from:

0 upto: 2*(1)+1

0 to 1
0 to 3
0 to 5

0 to 0 → 0
1 to 3 → 2
1 to 5 → 4.

1 2 4 5 6 7

1 2 3 4 5 6 7

temp = index.	temp = index
index = ele	index = ele
ele = temp	ele = temp

enum - Enumeration

* Set of int Constant - Common
* eg: enumeration of the Coins used in US

penny, nickel, dime, quarter, half dollar, dollar.

* enum — defined like Struct.

Keyword: enum

enum tag {enumeration list} variable_list;

eg: enum coin { penny, nickel, dime, quarter, half_dollar, dollar };
0 1 2 3 4 5

```
int main()
{
  enum coin money;
}
```

Valid statements

```
money = dime;
if (money == quarter) printf("a quarter\n");
```

What is enum?

* Each symbols stands for an int value – can be used anywhere. like an integer.
* Each symbol → given value one greater than the variable preceding

eg.

```
printf("%d", penny, dime);
```
(penny → 0, dime → 2)

* we can also initialize enum

```
enum coin { penny, nickel, dime, quarter = 100, half_dollar, dollar };
```
penny → 0, nickel → 1, dime → 2, quarter → 100, half_dollar → 101, dollar → 102

"Common errors"

* Symbols can be I/P & O/P directly. → wrong!

```
/* won't work */
money = dollar;
printf("%s", money);
```

dollar (a variable) – not a string.

'Name for an integer' – not a string!

`strcpy(money, "dime");` → also wrong!

∴ money → holding ('enum type')!

→ Can't expect to automatically converted to that symbol 'string'.

```
switch (money) {
    case penny: printf("penny\n");
    break;
    case nickel: printf("nickel\n");
        break;
        ⋮
}
```

use

```
enum decimal {... };
int main()
{
 enum decimal deci;
}
```

'Tag – important'

won't work

```
enum {... }decimal;
int main()
{
 enum decimal deci;
}
```

work

```
typedef enum {... }decimal;
int main()
{
}
```

(or)

```
char name[][20] = {"penny", "nickel", "dime", "quarter", "half_dollar",
                   "dollar"};
```

```
printf("%s", name[money]);
```

'use: define compiler's symbol table'

Important difference b/w c & C++

* Related to type names – 'structures, unions & enumerations.

```
struct MyStruct {
    int a;
    int b;
};

struct MyStruct Obj;
```

C prog

```
MyStruct Obj;
```

C++

* In C++, keyword struct → not necessary (we can use tag alone).
* In C, a structure's name doesn't define a complete type name. That's why C requires this as a tag!
* In C++, a structure's name is a complete type name — however it is perfectly legal to use C style (writing struct). → In C++

---

* Same applies for enum, union & struct (C++ don't need keyword)
* But C++ to C (look after this)

> * C++ supports C
> * C need not to support C++

Portability

* sizeof() char – 1
  int – 4
  double – 8

→ Size of structure – greater than or equal to the sum of its members.

```
struct S {
    char ch;
    int i;
    double d;
} S_var;
```

sizeof(S_var) → atleast

1+4+8 = 13 bytes.