C APTITUDE

Pointer

Declaration of pointer

Type	Declaration	Meaning
Pointer to int	int *p;	stores address of int
Pointer to pointer	int **p;	stores address of pointer
Constant pointer	int *const p;	pointer can't change
Pointer to const	const int *p;	data can't change
Both constant	const int *const p;	neither can change
Void pointer	void *p;	can hold any address
Pointer to array	int (*p)[3];	points to entire array
Array of pointers	int *p[3];	array of pointer variables
Function pointer	int (*f)(int,int);	points to function
Pointer to struct	struct st *p;	points to structure
Function returning pointer	int* f();	returns address of int

Size of the pointer doesn't depend on the type.

Example

Int *ptr_x;

sizeof(ptr_x) -> depend on the machine (8 0r 4)

sizeof(*ptr_x)-> depend on the type(here its int so 4)

Array and Pointer

int arr[5] = {1, 2, 3, 4, 5};

int *b = arr;

ptr1 = var; → Points to the **first element**

ptr1++;

→ Moves by **4 bytes** (size of int).

ptr2 = &var; → Points to the **entire array**

ptr2++;

→ Moves by **12 bytes** (size of whole array: 3 * sizeof(int)).

==========

🔥 C APTITUDE + CONCEPT REVISION NOTES (2025 EDITION)

==========

Author: ChatGPT + You (Deep-dive learning sessions)

🧩 1️⃣ FUNDAMENTALS — TYPES, VARIABLES, STORAGE CLASSES

⚙️ Data Types

char → 1 byte

int → 4 bytes

float → 4 bytes

double→ 8 bytes

⚙️ Storage Classes

Class	Lifetime	Scope	Default	Notes
auto	block	local	garbage	default local
register	block	local	garbage	stored in CPU register
static	program	local/global	0	retains value
extern	program	global	0	links to external var

Tips:

- extern only declares, not defines.
- static persists between calls.

🧩 2️⃣ POINTERS & ARRAYS

Pointer basics:

int a=10; int *p=&a; printf("%d", *p); // 10

Pointer arithmetic:

p+1 → next element

p2 - p1 → element gap

(char*)p2 - (char*)p1 → byte gap

Example:

```
int arr[]={10,20,30,40,50};
int *p1=arr, *p2=arr+4;
printf("%d", p2-p1); //4
printf("%d",(char*)p2-(char*)p1); //16 bytes
```

Pre/Post increment:

*ptr++ → use then move

*++ptr → move then use

(*ptr)++ → increment value

Array-pointer equivalence:
arr[i] == *(arr+i) == i[arr]

🧩 3️⃣ OPERATORS & EXPRESSIONS

Comma operator:
(a,b) → evaluates both, returns last

Ternary:
cond ? expr1 : expr2

Undefined behavior:
++i + ++i;
printf("%d %d", i, i++);
x = ++i + ++i + ++i;

sizeof:
Compile-time, no side effects
sizeof(++a + b); → doesn't increment a

🧩 4️⃣ STRINGS & POINTER TRICKS

---

char *p="hello"; // points to literal
char s[]="hello"; // array copy (modifiable)

Pointer arithmetic:
"Geeks for Geeks"+6 → "for Geeks"

String skip:
"abcdef"+2 → "cdef"

🧩 5️⃣ FUNCTIONS & BEHAVIOR

Static inside function:
static int x=5; retains value.

Extern:
extern int var; → global link

printf():
returns number of chars printed.

scanf():
returns successful input count.

🧩 6️⃣ CONTROL STATEMENTS

Empty for loop:
for(k=0.0;k<3.0;k++); → prints 3.0

Switch:
Fallthrough without break.

Goto:
Label must be in same function.

🧩 7️⃣ STRUCT, UNION, ENUM

Structure:
sum of all members + padding

Union:
size of largest member

Enum:
auto-increment unless specified

🧩 8️⃣ POINTER ADVANCED EXAMPLES

Function pointer movement:

```
void test(char c[]){
 c=c+2; c--; printf("%c",*c);
}
```

```
char ch[]={'p','o','u','r'};
test(ch); // o

(buf+1)[5] == buf[6]
```

🧩 9️⃣ SPECIAL CASES

Octal: 010 → 8 decimal
Hex: 0xA → 10 decimal

printf("%d",(1,2,3)); → 3

```
register int i;
scanf("%d",&i); // invalid
```

sizeof with expression: no side effect.

🧩 🔟 QUICK RECAP CHEATS

Pointers:
*(p+n)==p[n]==n[p]
Strings:
"hello"+2=="llo"
sizeof:
Type-based, no evaluation
Static:
Retains value
Extern:
Global link
Union:
Largest member size
printf():
Returns printed char count
scanf():
Returns success count
Comma:
Returns rightmost value
Undefined:
Never modify same var twice

⚡ MEMORY BOOST FORMULA SHEET

---

1. sizeof doesn’t execute ++
2. "string"+n → substring
3. i[arr]==arr[i]
4. printf returns char count
5. scanf returns input count
6. static persists between calls
7. extern links to global var
8. pointer arithmetic → element-based
9. struct adds padding
10. union largest member defines size
11. ++ and -- never mix in one line
12. switch fallthrough without break

✅ FINAL TAKEAWAYS

✓ sizeof is compile-time, safe.
✓ Don’t modify same variable twice.
✓ Use %p for pointers.
✓ Array decays to pointer in function.
✓ String literals are read-only.
✓ printf/scanf return useful values.
✓ static persists; extern connects.
✓ Comma returns last expression.
✓ Keep one increment per line.
✓ Understand pointer arithmetic deeply.

🧠 Practice Output Predicting Questions

---

1. printf("%s",6+"Geeks for Geeks"); → "for Geeks"
2. printf("%c",*p++); → first char then move
3. printf("%d",sizeof(++a+b)); → sizeof(float)=4
4. switch(2) { case 2: printf("B"); } → B
5. union with int[34] → 136 bytes (if int=4)
6. printf("%d",(1,2,3)); → 3
7. printf("%x",-1<<1); → fffffffe

1️⃣ Loop decrement and print pattern

Code:

for (n = 7; n != 0; n--) printf("n = %d", n--);

Trick: post-decrement happens twice.

Output: 7, 5, 3, 1

2️⃣ Bitwise left shift with negative number

---

printf("%x", -1 << 1);

Trick: -1 = 0xFFFFFFFF (32-bit); shift → 0xFFFFFFFE.

Output: fffffffe

3️⃣ Macro without parentheses

#define prod(a,b) a*b

printf("%d", prod(x+2,y-1));

Trick: expands to x+2*y-1 → wrong precedence.

Output: 10

Fix: #define prod(a,b) ((a)*(b))

4️⃣ Static variable recursion

---

static int i=5; if(--i){main(); printf("%d ",i);}

Trick: Static persists across calls.

Output: 0 0 0 0

5️⃣ scanf return value

---

scanf("%d",&x);

printf("%d", scanf("%d",&x));

Trick: scanf returns number of successful reads.

Output: 1

6️⃣ Unsigned overflow

---

unsigned int i=65000; while(i++!=0); printf("%u",i);

Trick: Wrap-around → next value = 1.

Output: 1

7️⃣ signed char overflow

---

signed char i=0; for(;i>=0;i++); printf("%d",i);

Range: -128 to 127.

Output: -128

8️⃣ Function call argument order

---

f(i++,i++,i++);

Order unspecified → undefined behavior.

Outputs: 10 11 12 13 OR 12 11 10 13.

9️⃣ Calling convention (pascal vs cdecl)

Pascal → Left to Right, Cdecl → Right to Left

Outputs:

10 11 12 13

12 11 10 13

🔟 Pointer arithmetic increment

++*ptr++;

Trick: Increment value, then move pointer.

Output: modified string ("hffltgpshfflt")

1️⃣1️⃣ Label inside another function

---

goto inside_foo; // invalid across functions

Trick: Labels local to same function.

1️⃣2️⃣ printf field width

---

printf("%*d",x,p);

Trick: Dynamic width; x=5 → " 10"

Output: 10

1️⃣3️⃣ Pointer vs array address

---

if(&p == (char*)&arr) printf("Same");

Output: Not same

1 4 sizeof pointer vs array

---

char arr[]={1,2,3}; char *p=arr;

sizeof(p)=8, sizeof(arr)=3

Output: 8 3

1 5 Shadowed global variable

---

int x=0; int f(){return x;} int g(){int x=1;return f();}

Trick: f() sees global variable.

Output: 0

1 6 Shift operators in printf

---

printf("%d %d %d",c,c<<=2,c>>=2);

Trick: modifies same var → undefined.

Possible: 5 20 5 OR 4 4 1

1 7 Array decay in sizeof

Inside func, array decays → pointer.

Output: 8 (64-bit), not 3.

1 8 Pointer vs pointer-to-const

---

void fun(const char **p); fun(argv);

Trick: char** → const char** invalid.

Fix: fun(char * const *p)

1 9 Pointer arithmetic simplification

---

*a+1-*a+3 = 4 (cancel same terms)

Output: 4

2 0 String pointer equivalence

---

str[i] == *(str+i) == *(i+str) == i[str]

Output:

gggg

eeee

kkkk

ssss ...

2 1 signed vs unsigned wrap-up

---

signed overflow → undefined

unsigned overflow → wraps to 0

2 2 sizeof in summary

---

sizeof(pointer) = 4/8 bytes

sizeof(array) = N * sizeof(type)

Golden Rules

---

1. Never modify/read same variable in one expression.
2. Use parentheses in macros.
3. Arrays decay to pointers (except in sizeof or &).
4. Unsigned wraps; signed overflows undefined.
5. Function argument order unspecified.
6. Static persists; locals reset.
7. Pointer size fixed, array size dynamic.
8. goto valid only inside same function.
9. const correctness: char** != const char**.
10. i[arr] == arr[i].

📘 POINTER INCREMENT — FULL EXPLANATION

1️⃣ Pointer Basics

---

int a[] = {10, 20, 30};
int *p = a; // p → &a[0]

p → address of a[0]
*p → value at p
p+1 → address of next element
*(p+1) → value of next element

2️⃣ Pointer Increment vs Value Increment

Expression	Effect
p++	Move pointer to next element (address changes)
++p	Move pointer to next element (address changes)
(*p)++	Increment value pointed by p (data changes)
++(*p)	Increment value pointed by p (data changes)
*++p	Move pointer first, then dereference
*p++	Dereference first, then move pointer

3️⃣ Example

```
int arr[] = {10, 20, 30};
int *p = arr;

printf("%d ", *p++);   // 10
printf("%d ", *++p);   // 30
printf("%d ", (*p)++); // 30
printf("%d ", ++(*p)); // 32
```

Output: 10 30 30 32
Final arr = {10, 20, 32}

4️⃣ When Address Moves vs Value Changes

Expression	Address Moves	Value Changes
p++	✅ Yes	❌ No
++p	✅ Yes	❌ No
*p++	✅ Yes	❌ No
*++p	✅ Yes	❌ No
(*p)++	❌ No	✅ Yes
++(*p)	❌ No	✅ Yes

5️⃣ Mnemonic Rule

If ++ is **inside parentheses**, it affects **value**.

If ++ is **outside parentheses**, it affects **pointer**.

6️⃣ Shortcut Table

Expression	Step Order	Moves Pointer	Changes Value	Prints
*p++	use → move	✅	❌	old value
*++p	move → use	✅	❌	next value
(*p)++	use → inc	❌	✅	old value
++(*p)	inc → use	❌	✅	new value

7️⃣ Practice Example

```
int arr[] = {5,10,15};
int *p = arr;
printf("%d ", *p++);    // 5
printf("%d ", (*p)++);  // 10
printf("%d ", *++p);    // 15
```

Output: 5 10 15

Final arr: {5, 11, 15}

✅ **Summary**

- ++p or p++ → moves pointer (address)
- (*p)++ or ++(*p) → modifies value (data)
- *p++ → use old value, then move
- *++p → move first, then use new value
- Never combine multiple ++ on same variable

Mnemonic:

> “Inside parentheses → Value changes.

> Outside parentheses → Address moves.”