NEC Registration Examination

Computer Engineering (ACtE)

Chapter 3: Programming Language and Its Applications (ACtE03)

Computer Engineering (ACtE)
Electronics and Communication
Telecommunication
Software Engineering
Information Technology

Full Operator Tables • All Inheritance Types • Worked Programs with Output • Diagrams • Practice MCQs

COMPLETE NOTES WITH SOLVED MCQ AND PYQS

EASY EXPLANATION



3.1 Introduction to C Programming

C Tokens — the Six Building Blocks

A **token** is the smallest individual unit the compiler recognises. Every C program is built entirely out of these six token types:

Token Type	Description	Examples
Keywords	Reserved words with fixed meaning (32 in ANSI C); cannot be used as names	`int, float, if, while, return, struct`
Identifiers	Names given by the programmer to variables/functions. Must start with a letter or _	`sum, total_marks, _count, main`
Constants	Fixed values that do not change during execution	`10, 3.14, 'A', "hello"`
Strings	Sequence of characters ending with the null character	`"Nepal", "NEC 2026"`
Operators	Symbols that perform operations on operands	`+, -, *, /, %, &&, ++`
Special Symbols	Punctuation with special meaning	`{ } [ ] ( ) ; , #`

Complete Operator Reference — Classified by Type

Operators are classified both by **what they do** (arithmetic, relational...) and by **how many operands they take** (**unary** = 1 operand, **binary** = 2 operands, **ternary** = 3 operands).

1. Arithmetic Operators

Operator	Meaning	Example (a=10, b=3)	Result
+	Addition	`a + b`	13
–	Subtraction	`a - b`	7
*	Multiplication	`a * b`	30
/	Division (integer division if both are int)	`a / b`	3 (not 3.33)
%	Modulus — remainder (integers only)	`a % b`	1

Important: the `%` operator **cannot be used with float/double** operands. And `10/3` gives `3` (integer division), whereas `10.0/3` gives `3.33`.

2. Relational Operators

Compare two values and return `1 (true)` or `0 (false)`. All are binary.

Operator	Meaning	Example (a=10, b=3)	Result
<	Less than	`a < b`	`0 (false)`
>	Greater than	`a > b`	`1 (true)`
<=	Less than or equal to	`a <= b`	0
>=	Greater than or equal to	`a >= b`	1
==	Equal to (comparison)	`a == b`	0
!=	Not equal to	`a != b`	1

Classic exam trap: `=` is the **assignment** operator while `==` is the **comparison** operator. Writing `if(a = 5)` assigns 5 to a and is always true!

3. Logical Operators

Operator	Name	Type	Meaning
&&	Logical AND	Binary	True only if BOTH operands are true
||	Logical OR	Binary	True if AT LEAST ONE operand is true
!	Logical NOT	Unary	Reverses the truth value

Short-circuit evaluation: in A && B, if A is false, B is **never evaluated** (result is already false). In A || B, if A is true, B is never evaluated.

4. Bitwise Operators

Operate on individual **bits** of integer operands.

Operator	Name	Type	Example (a=12 i.e. 1100, b=10 i.e. 1010)
&	Bitwise AND	Binary	a & b = 1000 = 8
|	Bitwise OR	Binary	a | b = 1110 = 14
^	Bitwise XOR	Binary	a ^ b = 0110 = 6
~	Bitwise NOT (1's complement)	Unary	~a = -13
<<	Left shift (multiply by 2^n)	Binary	a << 1 = 24
>>	Right shift (divide by 2^n)	Binary	a >> 1 = 6

5. Assignment Operators

Operator	Equivalent to	Example (a=10)	Result
=	Simple assignment	a = 5	a becomes 5
+=	a = a + x	a += 5	15
-=	a = a – x	a -= 5	5
*=	a = a * x	a *= 5	50
/=	a = a / x	a /= 5	2
%=	a = a % x	a %= 3	1

6. Increment / Decrement Operators (Unary)

Form	Name	Behaviour	Example (x=5)
++x	Pre-increment	Increment FIRST, then use the value	y = ++x → x=6, y=6
x++	Post-increment	Use the value FIRST, then increment	y = x++ → y=5, x=6
--x	Pre-decrement	Decrement first, then use	y = --x → x=4, y=4
x--	Post-decrement	Use first, then decrement	y = x-- → y=5, x=4

Program 1: The classic increment trap

```
#include <stdio.h>
int main() {
    int a = 5;
    int b = a++ + ++a;   // a++ uses 5 (a becomes 6); ++a makes a=7, uses 7
    printf("a = %d, b = %d\n", a, b);
    return 0;
}
```

▶Output

```
a = 7, b = 12
```

7. Other Important Operators

Operator	Name	Type	Purpose / Example
?:	Conditional (Ternary)	Ternary	max = (a>b) ? a : b;
sizeof	Sizeof	Unary	sizeof(int) → 4 (bytes)
&	Address-of	Unary	&x gives the memory address of x
*	Dereference / Indirection	Unary	*p gives the value stored at address p
.	Member access (dot)	Binary	s.name for a structure variable
->	Member access via pointer	Binary	p->name when p is a pointer to struct
,	Comma operator	Binary	Evaluates left to right, returns rightmost value

Operator	Name	Type	Purpose / Example
(type)	Type cast	Unary	(float)10/3 → 3.33

8. Operator Precedence and Associativity (high → low)

Precedence	Operators	Associativity
1 (highest)	() [] . ->	Left to Right
2	! ~ ++ -- + - (unary), (type), *, &, sizeof	Right to Left
3	* / %	Left to Right
4	+ -	Left to Right
5	<< >>	Left to Right
6	< <= > >=	Left to Right
7	== !=	Left to Right
8	& then ^ then |	Left to Right
9	&& then ||	Left to Right
10	?: (ternary)	Right to Left
11 (lowest)	= += -= *= /= %= , (comma)	Right to Left

Easy Explanation

Memorize precedence with the phrase 'PUMA-RE-BLAC': Parentheses → Unary → Multiplicative (/%)→ Additive (+-) → Relational → Equality → Bitwise → Logical → Assignment → Comma. Note that unary operators and assignment are the two groups that associate RIGHT to LEFT — everything else is left to right.*

Formatted and Unformatted Input/Output

Category	Functions	Notes
Formatted	printf(), scanf()	Use format specifiers; can handle any data type
Unformatted (char)	getchar(), putchar()	Single character input/output
Unformatted (string)	gets(), puts()	Whole line; gets() is unsafe (no bounds check)

Specifier	Data Type	Specifier	Data Type
%d / %i	int	%f	float
%ld	long int	%lf	double
%c	char	%s	string (char array)
%u	unsigned int	%p	pointer/address
%x / %o	hexadecimal / octal	%e	scientific notation

Control Statements

A. Selection (Branching) Statements

- **if / if-else:** executes a block based on a condition.
- **else-if ladder:** tests multiple conditions in sequence; first true branch runs.
- **switch-case:** selects among many constant cases. Requires `break` to prevent **fall-through**. Works only with **int / char** (not float or string).

Program 2: switch-case

```
// switch demonstrating fall-through danger
int grade = 2;
switch (grade) {
    case 1: printf("First\n");   break;
    case 2: printf("Second\n");  break;   // without break, case 3 also runs
    case 3: printf("Third\n");   break;
    default: printf("Invalid\n");
```

```
}
```

▶Output

```
Second
```

B. Looping (Iteration) Statements

Loop	Condition checked	Minimum executions	Best used when
`for`	Before body (entry-controlled)	0	Number of iterations is known
`while`	Before body (entry-controlled)	0	Iterations depend on a condition
`do-while`	AFTER body (exit-controlled)	1 (always runs once)	Body must run at least once (e.g. menus)

Program 3: do-while executes at least once

```
// do-while always executes at least once, even though the condition is false
int i = 10;
do {
    printf("Executed once, i = %d\n", i);
    i++;
} while (i < 5);
```

▶Output

```
Executed once, i = 10
```

- **break:** exits the loop/switch **immediately**.
- **continue:** skips the rest of the current iteration and jumps to the **next iteration**.
- **goto:** unconditional jump to a label — available but discouraged (creates 'spaghetti code').

User-defined and Recursive Functions

A function has three parts: **declaration (prototype)** → **definition** → **call**. Functions promote reusability and modularity.

Program 4: Recursive factorial

```
#include <stdio.h>
int factorial(int n);            // 1. Declaration (prototype)

int main() {
    printf("5! = %d\n", factorial(5));   // 3. Function call
    return 0;
}

int factorial(int n) {           // 2. Definition
    if (n == 0 || n == 1)
        return 1;                // BASE CASE – stops the recursion
    return n * factorial(n - 1); // RECURSIVE CASE – moves toward base
}
```

▶Output

```
5! = 120
```

How the recursion unwinds: `factorial(5) = 5×factorial(4) = 5×4×factorial(3) = 5×4×3×factorial(2) = 5×4×3×2×factorial(1) = 5×4×3×2×1 = 120`

Easy Explanation

Every recursive function needs BOTH: (1) a base case that stops the recursion, and (2) a recursive case that moves TOWARD the base case. Missing either causes infinite recursion, which exhausts the call stack — a 'stack overflow'. Recursion uses more memory than a loop because each pending call keeps its own stack frame.

Arrays and String Manipulation

- **1-D array:** `int a[5];` — indices run from **0 to n−1** (so `a[5]` is out of bounds!).
- **2-D array:** `int a[3][4];` — 3 rows, 4 columns, stored in **row-major order** in C.

Address of a[i][j] = Base + (i × Ncols + j) × size_of_element

Strings in C are character arrays terminated by the null character `'\0'`. So `"NEC"` actually occupies **4 bytes** — N, E, C and \0.

Function	Purpose	Example
`strlen(s)`	Length excluding '\0'	`strlen("NEC") → 3`
`strcpy(d,s)`	Copy string s into d	`strcpy(d,"Nepal")`
`strcat(d,s)`	Append s to the end of d	`strcat("Kath","mandu")`
`strcmp(a,b)`	Compare; 0 if identical	`strcmp("a","a") → 0`
`strrev(s)`	Reverse the string	`strrev("abc") → "cba"`

Program 5: Sum and average of an array

```
#include <stdio.h>
int main() {
    int a[5] = {10, 20, 30, 40, 50};
    int sum = 0;
    for (int i = 0; i < 5; i++)
        sum += a[i];
    printf("Sum = %d, Average = %.2f\n", sum, sum / 5.0);
    return 0;
}
```

▶Output

```
Sum = 150, Average = 30.00
```

Probable MCQs: Introduction to C

1. int a=5; int b = a++ + ++a; What is b?

a) 10
b) 11
c) 12
d) 13

✓Answer: c) 12

Why: a++ uses 5 (a→6); ++a makes a=7 and uses 7. So b = 5+7 = 12.

2. Which loop is guaranteed to execute at least once?

a) for
b) while
c) do-while
d) None

✓**Answer: c) do-while**

Why: do-while checks the condition AFTER executing the body.

3. The modulus operator % can be applied to:

a) float only

b) double only

c) Integers only

d) Any data type

✓**Answer: c) Integers only**

Why: % requires integer operands; using it with float/double is a compile error.

4. In C, 2-D arrays are stored in:

a) Column-major order

b) Row-major order

c) Random order

d) Reverse order

✓**Answer: b) Row-major order**

Why: C uses row-major ordering — one complete row is stored before the next.

5. Which operator group has RIGHT to LEFT associativity?

a) * / %

b) + -

c) Assignment (=, +=)

d) Relational (<, >)

✓**Answer: c) Assignment (=, +=)**

Why: Assignment and unary operators associate right to left; most others are left to right.

6. The string "NEC" occupies how many bytes in memory?

a) 3

b) 4

c) 5

d) 6

✓**Answer: b) 4**

Why: 3 characters plus the terminating null character '\0' = 4 bytes.

7. A recursive function MUST have a:

a) Loop

b) Base case

c) Global variable

d) Pointer

✓**Answer: b) Base case**

Why: Without a base case the recursion never ends, causing stack overflow.

8. In A && B, if A is false then B is:

a) Always evaluated

b) Never evaluated (short-circuit)

c) Evaluated twice

d) Converted to true

✓**Answer: b) Never evaluated (short-circuit)**

Why: Logical AND short-circuits: once A is false the result is known, so B is skipped.

3.2 Pointers, Structure and Data Files in C

Pointer Fundamentals

A **pointer** is a variable that stores the **memory address** of another variable. Two operators are central:

Operator	Name	Meaning
&	Address-of	Gives the address of a variable: &x
*	Dereference / Indirection	Gives the value AT an address: *p

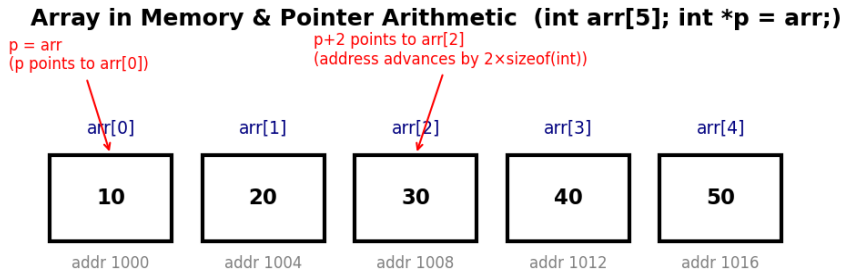
Program 6: Basic pointer usage

```
#include <stdio.h>
int main() {
    int x = 25;
    int *p = &x;        // p stores the ADDRESS of x
    printf("Value of x      = %d\n", x);
    printf("Value via *p    = %d\n", *p);
    *p = 50;            // writing through the pointer changes x itself
    printf("x after *p = 50 : %d\n", x);
    return 0;
}
```

▶Output

```
Value of x      = 25
Value via *p    = 25
x after *p = 50 : 50
```

Pointer Arithmetic and the Pointer–Array Relationship



p + 1 → address advances by 1 × sizeof(data type)

arr ≡ &arr[0] and arr[i] ≡ *(arr + i)

So for `int *p` (4-byte int) holding address 2000, `p+1` is **2004** — not 2001. For a `double *` (8 bytes) it would be 2008.

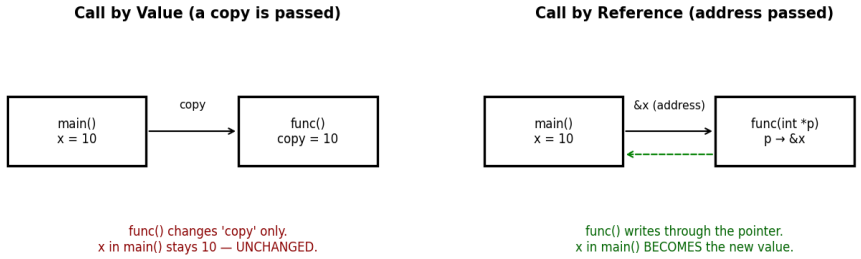
Program 7: Array traversal using pointers

```
int arr[5] = {10, 20, 30, 40, 50};
int *p = arr;                   // array name decays to &arr[0]
printf("%d\n", *(p + 2));       // same as arr[2]
printf("%d\n", *(arr + 3));     // same as arr[3]
```

▶Output

```
30
40
```

Passing Pointers to Functions (Call by Value vs Call by Reference)



Program 8: Swapping two numbers — value vs reference

```c
#include <stdio.h>
void swapWrong(int a, int b) {     // call by VALUE — swaps copies only
    int t = a; a = b; b = t;
}
void swapRight(int *a, int *b) {   // call by REFERENCE — swaps originals
    int t = *a; *a = *b; *b = t;
}
int main() {
    int x = 10, y = 20;
    swapWrong(x, y);
    printf("After swapWrong : x=%d y=%d\n", x, y);
    swapRight(&x, &y);
    printf("After swapRight : x=%d y=%d\n", x, y);
    return 0;
}
```

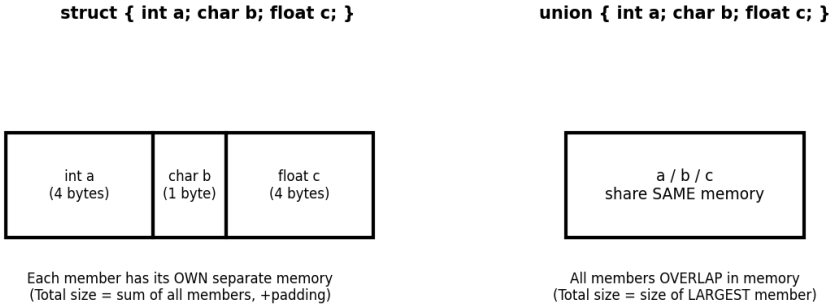
▶Output

```
After swapWrong : x=10 y=20
After swapRight : x=20 y=10
```

🔍 Easy Explanation

C is ALWAYS call-by-value — but when you pass an address, the copied value IS an address, so the function can reach back into the caller's memory. That is how 'call by reference' is simulated in C.

Structure vs Union — Detailed Comparison



Feature	struct	union
Keyword	`struct`	`union`
Memory allocation	Each member gets its OWN memory	All members SHARE the same memory
Total size	≥ sum of all member sizes (plus padding)	= size of the LARGEST member
Members usable at a time	ALL members simultaneously	Only ONE member holds a valid value
Writing to one member	Does not affect others	OVERWRITES the other members
Typical use	Grouping related data (a record)	Saving memory when only one field is needed at a time

Program 9: struct vs union memory size

```
#include <stdio.h>
struct S { int a; char b; float c; };
union  U { int a; char b; float c; };
int main() {
    printf("sizeof(struct) = %lu\n", sizeof(struct S));  // 4+1+4 (+padding)
    printf("sizeof(union)  = %lu\n", sizeof(union U));   // size of largest = 4
    return 0;
}
```

▶Output

```
sizeof(struct) = 12
sizeof(union)  = 4
```

Array of Structures, Nested Access and Pointers to Structures

Program 10: Array of structures and the -> operator

```
#include <stdio.h>
struct Student {
    char name[20];
    int  roll;
    float marks;
};
int main() {
    struct Student s[2] = { {"Ram", 1, 78.5}, {"Sita", 2, 85.0} };
    struct Student *p = &s[0];

    printf("%s scored %.1f\n", s[0].name, s[0].marks);  // dot with variable
    printf("%s scored %.1f\n", p->name, p->marks);      // arrow with pointer
    return 0;
}
```

▶Output

```
Ram scored 78.5
Ram scored 78.5
```

Rule to remember: use `.` when you have a structure **variable**; use `->` when you have a **pointer** to a structure. `p->name` is shorthand for `(*p).name`.

File Handling in C

Files give **permanent (non-volatile) storage** so data survives after the program ends. Everything goes through a **file pointer**: `FILE *fp;`

Mode	Meaning	If file does not exist	If file exists
`"r"`	Read only	Returns NULL (error)	Opens for reading
`"w"`	Write only	Creates new file	ERASES all existing content
`"a"`	Append	Creates new file	Adds to the END, keeps content

Mode	Meaning	If file does not exist	If file exists
"r+"	Read and write	Returns NULL	Opens without erasing
"w+"	Read and write	Creates new file	Erases existing content
"a+"	Read and append	Creates new file	Appends, keeps content

Function	Purpose
fopen() / fclose()	Open / close a file
fprintf() / fscanf()	Formatted write / read
fputc() / fgetc()	Single character write / read
fputs() / fgets()	String write / read
fwrite() / fread()	Block (binary) write / read
fseek() / ftell() / rewind()	Random access: move / report / reset file position
feof()	Tests for end-of-file

Program 11: Writing to and reading from a file

```
#include <stdio.h>
int main() {
    FILE *fp;
    char line[100];

    fp = fopen("data.txt", "w");          // create/overwrite
    if (fp == NULL) { printf("Error!\n"); return 1; }
    fprintf(fp, "NEC Exam 2026\n");
    fclose(fp);                            // ALWAYS close

    fp = fopen("data.txt", "r");          // reopen for reading
    fgets(line, 100, fp);
    printf("Read from file: %s", line);
    fclose(fp);
    return 0;
}
```

▶Output

```
Read from file: NEC Exam 2026
```

Access Type	How it works	Speed for a specific record
Sequential access	Reads records one after another from the start	Slow for records near the end
Random access	Jumps straight to a position using fseek()	Fast — direct positioning

Easy Explanation

Two file-handling habits that show up as 'find the bug' MCQs: (1) always check whether fopen() returned NULL before using the file, and (2) always call fclose() — otherwise buffered data may never be written to disk.

Probable MCQs: Pointers, Structures and Files

1. If int *p holds address 2000 and int is 4 bytes, p+1 points to:

a) 2001
b) 2002
c) 2004
d) 2000
✓Answer: c) 2004

Why: Pointer arithmetic scales by the size of the pointed-to type: 2000 + 1×4 = 2004.

2. In a union, the allocated memory equals the size of the:

a) First member
b) Sum of all members
c) Largest member
d) Smallest member
✓Answer: c) Largest member
Why: All union members overlap in one shared space sized for the largest member.

3. Which operator accesses a structure member through a POINTER?

a) .
b) ->
c) &
d) *
✓Answer: b) ->
*Why: p->member is shorthand for (*p).member.*

4. Opening an existing file in "w" mode will:

a) Append to the end
b) Erase all existing content
c) Return NULL
d) Open it read-only
✓Answer: b) Erase all existing content
Why: Mode "w" truncates the file to zero length; use "a" to preserve content.

5. Which function moves the file pointer for random access?

a) fopen()
b) fseek()
c) fprintf()
d) fclose()
✓Answer: b) fseek()
Why: fseek() repositions the file pointer to any offset, enabling direct record access.

6. An array name used in an expression decays into:

a) A constant pointer to its first element
b) An integer
c) A structure
d) A function pointer
✓Answer: a) A constant pointer to its first element
*Why: arr is equivalent to &arr[0], and arr[i] equals *(arr+i).*

7. To let a function modify the caller's variable in C, you must pass:

a) The variable by value
b) The variable's address (pointer)
c) A copy of the variable
d) A constant
✓Answer: b) The variable's address (pointer)
Why: C is call-by-value; passing the address lets the function write back through the pointer.

8. Which statement about struct is TRUE?

a) All members share one memory location

b) Each member has its own separate memory

c) Only one member can hold a value at a time

d) Size equals the largest member

✓Answer: b) Each member has its own separate memory

Why: Those other options describe a union, not a struct.

3.3 C++ Language Constructs with Objects and Classes

Namespace

A **namespace** prevents naming collisions by grouping identifiers under a named scope. The standard library lives in namespace `std`.

Program 12: Namespaces avoid name clashes

```
#include <iostream>
namespace Nepal { int code = 977; }
namespace India  { int code = 91;  }

int main() {
    std::cout << Nepal::code << " " << India::code << std::endl;
    return 0;
}
```

▶Output

```
977 91
```

Function Overloading

Several functions may share the **same name** provided their **parameter lists differ** (in number or type). The compiler chooses the right one at **compile time** — this is **static / early binding** and is a form of **compile-time polymorphism**.

Important: functions **cannot** be overloaded on return type alone.

Program 13: Function overloading

```
int  add(int a, int b)        { return a + b; }
double add(double a, double b) { return a + b; }
int  add(int a, int b, int c) { return a + b + c; }

cout << add(2, 3)        << endl;   // calls version 1
cout << add(2.5, 3.5)    << endl;   // calls version 2
cout << add(1, 2, 3)     << endl;   // calls version 3
```

▶Output

```
5
6
6
```

Inline Functions and Default Arguments

- **Inline function:** the keyword `inline` **requests** the compiler to substitute the function body directly at the call site, avoiding call overhead. Best for **small, frequently called** functions; the compiler may refuse for large or recursive functions.
- **Default argument:** a parameter given a default value, used when the caller omits it. Defaults must be the **rightmost** parameters.

Program 14: Inline function and default arguments

```
inline int square(int x) { return x * x; }

void greet(string name = "Guest", string city = "Kathmandu") {
    cout << "Hello " << name << " from " << city << endl;
```

```
}

greet();                    // both defaults used
greet("Ram");               // city default used
greet("Sita", "Pokhara");   // no defaults used
```

▶Output

```
Hello Guest from Kathmandu
Hello Ram from Kathmandu
Hello Sita from Pokhara
```

Pass and Return by Reference

A **reference (&)** is an **alias** for an existing variable. Unlike a pointer it **cannot be null**, **must be initialised** when declared, and **cannot be reassigned** to refer to something else.

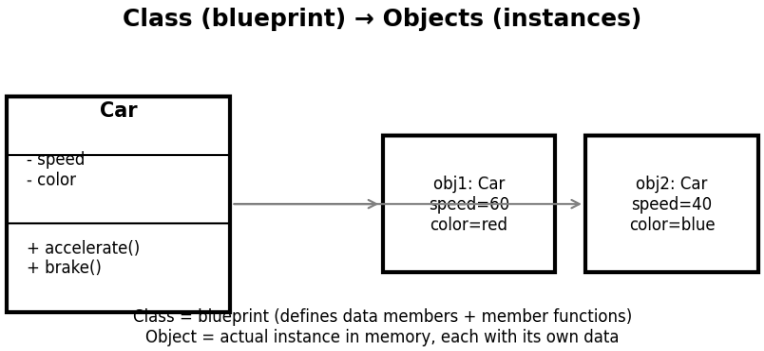
Program 15: Pass by reference in C++

```
void increment(int &x) {   // x is an ALIAS of the caller's variable
    x = x + 1;
}
int main() {
    int a = 10;
    increment(a);
    cout << a << endl;     // the original a was modified
    return 0;
}
```

▶Output

```
11
```

Classes, Objects and Access Specifiers



Specifier	Accessible within the class	Accessible in derived classes	Accessible outside
`private`	Yes	No	No
`protected`	Yes	Yes	No
`public`	Yes	Yes	Yes

Default access: members of a `class` are **private** by default, while members of a `struct` in C++ are **public** by default — a favourite MCQ distinction.

Constructors and Destructor

Type	Description	Example signature
Default constructor	No parameters; called automatically on object creation	`Student()`
Parameterized constructor	Takes arguments to initialise members	`Student(int r, float m)`
Copy constructor	Creates a new object as a copy of an existing one	`Student(const Student &s)`
Destructor	Frees resources; no return type, no arguments, cannot be overloaded	`~Student()`

Key properties of a constructor: same name as the class, **no return type (not even void)**, called **automatically**, **can be overloaded**.

Program 16: All three constructor types plus destructor

```
#include <iostream>
using namespace std;
class Student {
private:
    int roll;
    float marks;
public:
    Student() { roll = 0; marks = 0; cout << "Default ctor\n"; }      // default
    Student(int r, float m) { roll = r; marks = m;                     // parameterized
                              cout << "Param ctor\n"; }
    Student(const Student &s) { roll = s.roll; marks = s.marks;        // copy
                                cout << "Copy ctor\n"; }
    void show() const { cout << "Roll " << roll << ", Marks " << marks << endl; }
    ~Student() { cout << "Destructor for roll " << roll << endl; }
};

int main() {
    Student s1(5, 82.5);
    Student s2 = s1;        // copy constructor invoked
    s2.show();
    return 0;
}
```

▶Output

```
Param ctor
Copy ctor
Roll 5, Marks 82.5
Destructor for roll 5
Destructor for roll 5
```

Easy Explanation

Destructors run in REVERSE order of construction (last created is destroyed first — a stack/LIFO behaviour). That is why s2 is destroyed before s1 above.

Dynamic Memory Allocation for Objects

Purpose	C++ operator	Matching deallocation
Single object	`Student *p = new Student();`	`delete p;`
Array of objects	`Student *p = new Student[10];`	`delete[] p;`

Critical rule: `new` must pair with `delete`, and `new[]` must pair with `delete[]`. Mismatching them causes undefined behaviour; forgetting delete entirely causes a **memory leak**.

this Pointer, static Members and const Members

- **this pointer:** an implicit pointer available inside every **non-static** member function, pointing to the **object that invoked the function**. Useful when a parameter shadows a member name: `this->roll = roll;`
- **static data member:** only **ONE copy shared by all objects** of the class; exists even if no object is created. Must be defined outside the class.
- **static member function:** can access **only static members** (it has **no this pointer**); callable without an object as `ClassName::func()`.
- **const member function:** declared `void show() const;` — promises **not to modify** any data member.

Program 17: static data member counting objects

```
class Counter {
    static int count;          // ONE copy shared by all objects
public:
    Counter() { count++; }
    static int getCount() { return count; }   // no object needed
};
int Counter::count = 0;        // definition outside the class

int main() {
    Counter a, b, c;
    cout << "Objects created: " << Counter::getCount() << endl;
    return 0;
}
```

▶Output

```
Objects created: 3
```

Friend Function and Friend Class

A **friend** is granted access to the **private and protected** members of a class even though it is **not a member** of that class. Declared inside the class using the `friend` keyword.

Property	Behaviour
Is it a class member?	NO — it is an external function/class given special access
Called using an object?	No — called like a normal function, takes the object as an argument
Is it inherited?	NO — friendship is not inherited
Is it mutual?	NO — if A is a friend of B, B is not automatically a friend of A
Common use	Operator overloading needing access to two different classes

Program 18: Friend function accessing private data

```
class Box {
    int width;                      // private
public:
    Box(int w) { width = w; }
    friend void printWidth(Box b);   // friend declaration
};

void printWidth(Box b) {             // NOT a member, yet reads private data
    cout << "Width = " << b.width << endl;
}

int main() { Box b(15); printWidth(b); return 0; }
```

▶Output

```
Width = 15
```

Easy Explanation

Think of a friend as a trusted outsider given a house key — they can enter (access private members) but they do not live there (are not a class member), the key is not passed to their children (not inherited), and giving them a key does not give you a key to their house (not mutual).

Probable MCQs: C++ Classes and Objects

1. Function overloading is an example of which binding?

a) Dynamic (late) binding
b) Static (compile-time) binding
c) Runtime binding
d) No binding
✓Answer: b) Static (compile-time) binding
Why: The compiler selects the correct overload at compile time from the argument types.

2. Functions CANNOT be overloaded based on differences in only the:

a) Number of parameters
b) Type of parameters
c) Return type
d) Order of parameter types
✓Answer: c) Return type
Why: Return type alone is not part of the overload signature.

3. A destructor:

a) Can be overloaded
b) Takes arguments
c) Has no return type and cannot be overloaded
d) Must be called manually
✓Answer: c) Has no return type and cannot be overloaded
Why: ~ClassName() takes no arguments, returns nothing and is unique per class.

4. A static data member of a class is:

a) Created for every object
b) Shared by all objects of the class
c) Destroyed with each object
d) Only usable in main()
✓Answer: b) Shared by all objects of the class
Why: Exactly one copy exists, shared across all instances.

5. A static member function can directly access:

a) Only static members
b) Both static and non-static members
c) Only private non-static members
d) The this pointer
✓Answer: a) Only static members
Why: Static functions have no this pointer, so they can only touch static members.

6. A friend function of a class can access:

a) Only public members

b) Only protected members

c) Private and protected members too

d) No members

✓Answer: c) Private and protected members too

Why: Friendship grants full access to private and protected members.

7. Friendship in C++ is:

a) Inherited by derived classes

b) Mutual between the two classes

c) Neither inherited nor mutual

d) Always transitive

✓Answer: c) Neither inherited nor mutual

Why: Friendship is neither inherited nor mutual nor transitive.

8. In a C++ class, members are by default:

a) public

b) private

c) protected

d) static

✓Answer: b) private

Why: class members default to private; struct members default to public.

9. The 'this' pointer points to:

a) The base class

b) The calling object itself

c) A static member

d) The next object in memory

✓Answer: b) The calling object itself

Why: 'this' is an implicit pointer to the object invoking the member function.

3.4 Features of Object-Oriented Programming

The Four Pillars of OOP (quick recap)

Pillar	Meaning
Encapsulation	Bundling data + methods together and hiding internal details (via private members)
Abstraction	Exposing only essential features, hiding implementation complexity
Inheritance	Deriving a new class from an existing one, reusing its members
Polymorphism	One interface, many forms (overloading = compile-time; virtual functions = runtime)

Operator Overloading

Gives an **additional meaning** to an existing operator for **user-defined types**. It does **not** change the operator's meaning for built-in types, its **precedence**, or its **number of operands**.

Syntax: ReturnType operator<symbol>(parameters) { ... }

Category	Operands	Examples	Member function parameters
Unary	1	`+, -, ++, --, !, ~`	Zero (operand is the calling object)
Binary	2	`+, -, *, /, ==, <, >, +=`	One (the right-hand operand)

Operators that CANNOT be overloaded (memorize this list):

Operator	Name
`::`	Scope resolution operator
`.`	Member access (dot) operator
`.*`	Pointer-to-member access
`?:`	Conditional (ternary) operator
`sizeof`	Sizeof operator

Program 19: Unary and binary operator overloading

```
#include <iostream>
using namespace std;
class Complex {
    int real, imag;
public:
    Complex(int r = 0, int i = 0) { real = r; imag = i; }

    Complex operator+(const Complex &c) {     // BINARY: one parameter
        return Complex(real + c.real, imag + c.imag);
    }
    Complex operator-() {                      // UNARY: no parameter
        return Complex(-real, -imag);
    }
    void show() { cout << real << " + " << imag << "i" << endl; }
};

int main() {
```

```
    Complex c1(3, 4), c2(1, 2);
    Complex c3 = c1 + c2;      // invokes operator+
    Complex c4 = -c1;          // invokes operator-
    c3.show();
    c4.show();
    return 0;
}
```

▶Output

```
4 + 6i
-3 + -4i
```

Data Conversion

Conversion	Mechanism used	Where it is written
Basic type → Class	Single-argument constructor	In the destination class
Class → Basic type	`Conversion operator function, e.g. operator int()`	In the source class
Class → Class	Constructor in destination OR conversion operator in source	Either class

Inheritance — Detailed Study

Inheritance lets a **derived class** (child/subclass) reuse the members of a **base class** (parent/superclass), modelling an **IS-A** relationship. It is the main mechanism for **code reusability** in OOP.

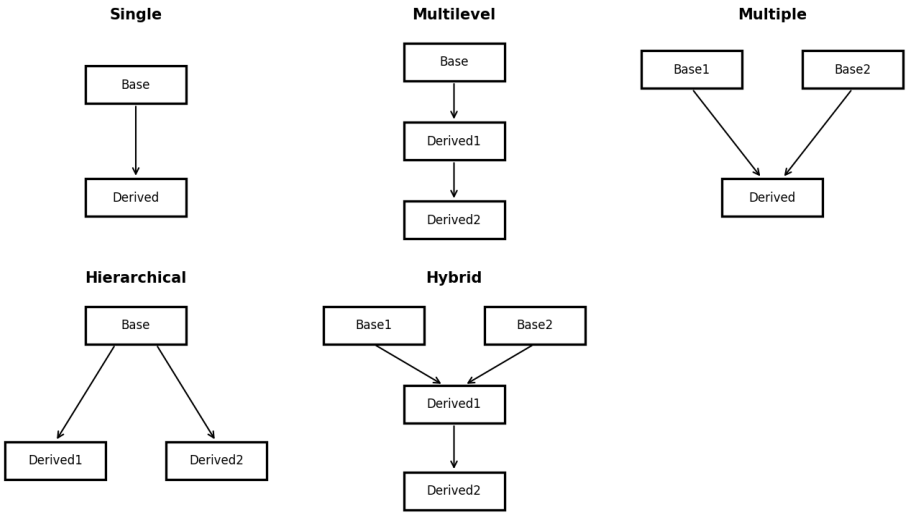
Syntax: class Derived : access_specifier Base { ... };

A. Visibility of inherited members (mode of inheritance)

Base member	public inheritance	protected inheritance	private inheritance
public	stays public	becomes protected	becomes private
protected	stays protected	stays protected	becomes private
private	NOT inherited (inaccessible)	NOT inherited	NOT inherited

Note: private members of a base class are **never directly accessible** in the derived class regardless of the inheritance mode — they can only be reached through public/protected member functions of the base.

B. The five types of inheritance



Type	Structure	Description
1. Single	`Base → Derived`	One base class, one derived class. Simplest form.

Type	Structure	Description
2. Multilevel	`Base → Derived1 → Derived2`	A chain: a derived class itself becomes the base for another class.
3. Multiple	`Base1 + Base2 → Derived`	ONE derived class inherits from TWO OR MORE base classes.
4. Hierarchical	`Base → Derived1, Derived2, Derived3`	ONE base class is inherited by SEVERAL derived classes.
5. Hybrid	`Combination`	Mix of two or more of the above (e.g. hierarchical + multiple).

Multipath inheritance and the Diamond Problem: in hybrid/multipath inheritance a class may inherit the same base **twice** through two different paths (the 'diamond'), creating **duplicate copies** of the base members and an **ambiguity** about which copy to use. The solution in C++ is **virtual base classes**:

Program 20: Virtual base class solving the diamond problem

```
class A { public: int x; };
class B : virtual public A { };    // 'virtual' keyword solves the diamond
class C : virtual public A { };
class D : public B, public C { };  // D now gets only ONE copy of A
```

C. Worked example — Single and Multilevel inheritance

Program 21: Multilevel inheritance with constructor/destructor order

```
#include <iostream>
using namespace std;
class Person {                        // BASE
protected:
    string name;
public:
    Person(string n) : name(n) { cout << "Person ctor\n"; }
    ~Person() { cout << "Person dtor\n"; }
};

class Student : public Person {       // SINGLE inheritance
protected:
    int roll;
public:
    Student(string n, int r) : Person(n), roll(r) { cout << "Student ctor\n"; }
    ~Student() { cout << "Student dtor\n"; }
};

class Engineer : public Student {     // MULTILEVEL inheritance
    string branch;
public:
    Engineer(string n, int r, string b)
        : Student(n, r), branch(b) { cout << "Engineer ctor\n"; }
    void show() { cout << name << " | Roll " << roll << " | " << branch << endl; }
    ~Engineer() { cout << "Engineer dtor\n"; }
};

int main() {
    Engineer e("Ram", 12, "Computer");
    e.show();
    return 0;
}
```

▶Output

```
Person ctor
Student ctor
Engineer ctor
```

```
Ram | Roll 12 | Computer
Engineer dtor
Student dtor
Person dtor
```

D. Constructor and destructor order in inheritance

Constructors: BASE first, then DERIVED (top → down)

Destructors: DERIVED first, then BASE (bottom → up, exact reverse)

Inheritance type	Constructor order	Destructor order
Single (A → B)	A then B	B then A
Multilevel (A → B → C)	A, B, C	C, B, A
Multiple (A, B → C)	In the order listed in the derived class declaration, then C	C, then reverse of the listed order

Easy Explanation

Think of constructing a building: you must build the FOUNDATION (base class) before the upper floors (derived class). When demolishing, you remove the TOP floors first and the foundation last. That is exactly the constructor/destructor ordering rule.

Probable MCQs: OOP Features and Inheritance

1. Which operator CANNOT be overloaded in C++?

a) +
b) ==
c) sizeof
d) []
✓Answer: c) sizeof
Why: sizeof, ::, ., . and ?: cannot be overloaded.*

2. A unary operator overloaded as a member function takes how many parameters?

a) Zero
b) One
c) Two
d) Three
✓Answer: a) Zero
Why: The single operand is the calling object itself, so no explicit parameter is needed.

3. In multiple inheritance, a derived class inherits from:

a) Exactly one base class
b) Two or more base classes
c) No base class
d) Only itself
✓Answer: b) Two or more base classes
Why: Multiple inheritance means more than one direct base class.

4. When one base class is inherited by several derived classes, it is called:

a) Multilevel inheritance
b) Multiple inheritance
c) Hierarchical inheritance
d) Hybrid inheritance
✓Answer: c) Hierarchical inheritance
Why: Hierarchical inheritance: one parent, many children.

5. A chain such as A → B → C is an example of:

a) Single inheritance
b) Multilevel inheritance
c) Multiple inheritance
d) Hierarchical inheritance
✓Answer: b) Multilevel inheritance
Why: Each derived class becomes the base for the next — multilevel.

6. The diamond problem in hybrid inheritance is solved using:

a) Friend functions
b) Virtual base classes
c) Static members
d) Inline functions
✓Answer: b) Virtual base classes
Why: Declaring the common base as virtual ensures only one shared copy exists.

7. A public member of the base class becomes _____ under protected inheritance.

a) public
b) protected
c) private
d) inaccessible
✓Answer: b) protected
Why: Protected inheritance downgrades public base members to protected in the derived class.

8. Private members of a base class are:

a) Directly accessible in derived classes
b) Never directly accessible in derived classes
c) Always public in derived classes
d) Automatically deleted
✓Answer: b) Never directly accessible in derived classes
Why: Private base members are inaccessible directly; only base member functions can reach them.

9. In multilevel inheritance A → B → C, the destructor order is:

a) A, B, C
b) C, B, A
c) B, A, C
d) Random
✓Answer: b) C, B, A
Why: Destructors run in reverse of construction order: most-derived first.

10. Converting a class object into a basic data type requires a:

a) Copy constructor
b) Conversion operator function

c) Friend class

d) Destructor

✓Answer: b) Conversion operator function

Why: A conversion operator such as operator int() performs class → basic conversion.

3.5 Pure Virtual Function and File Handling

Virtual Functions and Dynamic Binding

A **virtual function** is declared in a base class with the keyword `virtual` and **overridden** in derived classes. When called through a **base class pointer or reference**, the version that runs is decided by the **actual object type at runtime** — this is **runtime polymorphism**.

Feature	Static (Early) Binding	Dynamic (Late) Binding
Decided at	Compile time	Run time
Achieved by	Function/operator overloading	Virtual functions (overriding)
Speed	Faster	Slightly slower (vtable lookup)
Also called	Compile-time polymorphism	Runtime polymorphism

Program 22: Virtual function / runtime polymorphism

```
#include <iostream>
using namespace std;
class Shape {
public:
    virtual void draw() { cout << "Drawing a generic shape\n"; }
};
class Circle : public Shape {
public:
    void draw() override { cout << "Drawing a circle\n"; }
};
class Square : public Shape {
public:
    void draw() override { cout << "Drawing a square\n"; }
};

int main() {
    Shape *p;
    Circle c;  Square s;
    p = &c;  p->draw();     // runtime decides: Circle::draw
    p = &s;  p->draw();     // runtime decides: Square::draw
    return 0;
}
```

▶Output

```
Drawing a circle
Drawing a square
```

Without the `virtual` keyword, both calls would print "Drawing a generic shape" because the compiler would bind statically to `Shape::draw`.

Pure Virtual Function and Abstract Class

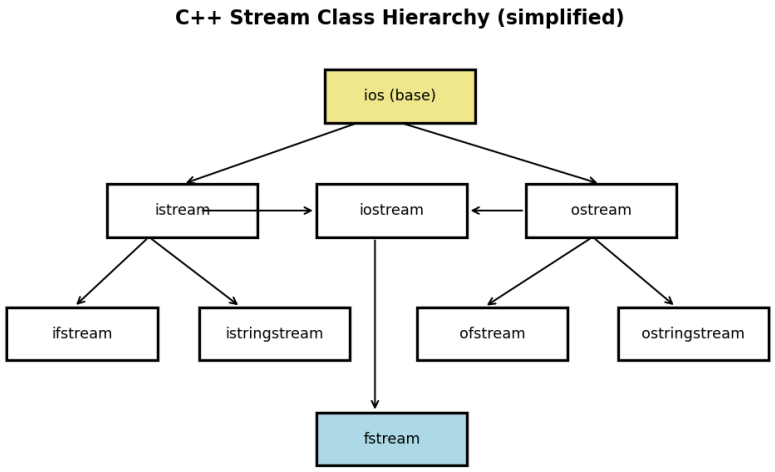
Pure virtual function: virtual void draw() = 0;

- A class containing **at least one** pure virtual function becomes an **abstract class**.
- An abstract class **cannot be instantiated** (no objects), but you **can** create pointers/references to it.
- Any derived class **must override** every pure virtual function, otherwise it too remains abstract.
- Purpose: defines a **common interface/contract** that all derived classes are forced to implement.

Easy Explanation

Virtual function = 'you MAY override me, and if you do, the right version is chosen at runtime.' Pure virtual function (=0) = 'you MUST override me; I provide no implementation at all.' The second makes the class abstract.

File Handling in C++ — Stream Class Hierarchy



Class	Header	Purpose
`ifstream`	<fstream>	Input file stream — READ from a file
`ofstream`	<fstream>	Output file stream — WRITE to a file
`fstream`	<fstream>	Both read and write
`istream / ostream`	<iostream>	Base classes for cin / cout

Program 23: C++ file write and read

```
#include <iostream>
#include <fstream>
using namespace std;
int main() {
    ofstream fout("marks.txt");        // open for writing
    if (!fout) { cout << "Cannot open file\n"; return 1; }
    fout << "Ram 82" << endl;
    fout << "Sita 91" << endl;
    fout.close();

    ifstream fin("marks.txt");         // open for reading
    string name;  int marks;
    while (fin >> name >> marks)        // loop until stream fails (EOF)
        cout << name << " scored " << marks << endl;
    fin.close();
    return 0;
}
```

▶Output

```
Ram scored 82
Sita scored 91
```

Formatted vs Unformatted I/O and Manipulators

Type	Functions / tools	Purpose
Unformatted	`get(), put(), read(), write(), getline()`	Raw characters or byte blocks

Type	Functions / tools	Purpose
Formatted (ios members)	`width(), precision(), fill(), setf()`	Control field width, decimals, fill char
Formatted (manipulators)	`setw(), setprecision(), fixed, showpoint, endl`	Same control, but usable inline with <<

Manipulators require the header `<iomanip>` (except simple ones like `endl`).

Program 24: Formatting with manipulators

```
#include <iostream>
#include <iomanip>
using namespace std;
int main() {
    double pi = 3.14159265;
    cout << setprecision(4) << fixed << pi << endl;   // 4 decimal places
    cout << setw(10) << setfill('*') << 42 << endl;    // width 10, pad with *
    return 0;
}
```

▶Output

```
3.1416
********42
```

Error Handling During I/O

Function	Returns true when
`eof()`	End of file has been reached
`fail()`	A formatting or read/write operation failed
`bad()`	An unrecoverable/serious error occurred
`good()`	No error flags are set — stream is healthy

? Probable MCQs: Virtual Functions and File Streams

1. A class containing at least one pure virtual function is called:

a) Concrete class
b) Abstract class
c) Friend class
d) Static class
✓Answer: b) Abstract class
Why: A pure virtual function (=0) makes the class abstract and non-instantiable.

2. Runtime polymorphism in C++ is achieved using:

a) Function overloading
b) Operator overloading
c) Virtual functions
d) Default arguments
✓Answer: c) Virtual functions
Why: Virtual functions enable dynamic (late) binding at run time.

3. A pure virtual function is declared as:

a) virtual void f();
b) virtual void f() = 0;
c) void f() = virtual;
d) static void f();
✓Answer: b) virtual void f() = 0;

Why: The '= 0' specifier marks the function as pure virtual.

4. Which stream class is used to READ from files?

a) ofstream
b) ifstream
c) sstream
d) cout
✓Answer: b) ifstream
Why: ifstream is the input file stream.

5. Which header provides setw() and setprecision()?

a) <iostream>
b) <iomanip>
c) <fstream>
d) <string>
✓Answer: b) <iomanip>
Why: <iomanip> supplies parameterized stream manipulators.

6. The correct virtual function is selected at runtime using a:

a) Symbol table
b) Vtable (virtual table)
c) Hash table
d) Stack frame
✓Answer: b) Vtable (virtual table)
Why: The compiler builds a vtable of function pointers per class for dynamic dispatch.

7. An abstract class can be used to create:

a) Objects directly
b) Pointers and references only
c) Nothing at all
d) Only static members
✓Answer: b) Pointers and references only
Why: You cannot instantiate an abstract class, but base pointers/references are allowed and essential for polymorphism.

8. Which function returns true when the end of a file is reached?

a) fail()
b) bad()
c) eof()
d) good()
✓Answer: c) eof()
Why: eof() tests the end-of-file flag on the stream.

3.6 Generic Programming and Exception Handling

Why Templates? (Generic Programming)

Templates let you write **one piece of code that works for many data types**. The compiler generates a separate concrete version for each type actually used — this happens at **compile time**, so there is no runtime cost.

Function Templates

template <class T> (or) template <typename T>
ReturnType functionName(T parameters) { ... }

Program 25: Function template

```
#include <iostream>
using namespace std;

template <class T>
T maxVal(T a, T b) {
    return (a > b) ? a : b;
}

int main() {
    cout << maxVal(3, 7)       << endl;   // T deduced as int
    cout << maxVal(2.5, 1.8)   << endl;   // T deduced as double
    cout << maxVal('a', 'z')   << endl;   // T deduced as char
    return 0;
}
```

▶Output

```
7
2.5
z
```

Overloading a function template: you may also provide a **specialised** version for a particular type, and the compiler prefers the more specific match.

Class Templates

Program 26: Class template — a generic Stack

```
#include <iostream>
using namespace std;

template <class T>
class Stack {
    T arr[100];
    int top;
public:
    Stack() { top = -1; }
    void push(T x) { arr[++top] = x; }
    T pop()        { return arr[top--]; }
};

int main() {
    Stack<int> si;  si.push(10); si.push(20);
    Stack<char> sc; sc.push('A');
    cout << si.pop() << " " << sc.pop() << endl;
```

```
    return 0;
}
```

▶Output

```
20 A
```

Defining a member function outside a class template requires repeating the template header:

```
template <class T>
void Stack<T>::push(T x) { arr[++top] = x; }
```

Standard Template Library (STL) — Three Components

Component	Purpose	Examples
Containers	Store collections of objects	`vector, list, deque, set, map, stack, queue`
Algorithms	Operate on container data	`sort(), find(), reverse(), count(), binary_search()`
Iterators	Pointer-like objects used to traverse containers, linking the two above	`begin(), end(), ++it, *it`

Container category	Members	Nature
Sequence containers	`vector, list, deque`	Elements in linear order
Associative containers	`set, map, multiset, multimap`	Sorted, key-based fast lookup
Container adapters	`stack, queue, priority_queue`	Built on top of other containers

Program 27: STL container + algorithm + iterators together

```
#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;
int main() {
    vector<int> v = {40, 10, 30, 20};
    sort(v.begin(), v.end());                // ALGORITHM using ITERATORS
    for (int x : v) cout << x << " ";        // CONTAINER traversal
    cout << endl;
    return 0;
}
```

▶Output

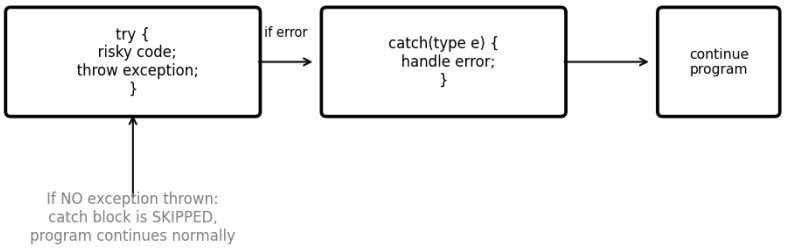
```
10 20 30 40
```

Easy Explanation

Remember STL as a three-part system: CONTAINERS hold the data, ALGORITHMS do the work, and ITERATORS are the bridge that lets any algorithm work with any container.

Exception Handling — try, throw, catch

try - throw - catch Flow



Keyword	Role
`try`	Encloses code that MIGHT cause an exception
`throw`	Signals that an exceptional condition has occurred; throws a value/object of any type
`catch`	Handles an exception of a matching type thrown from the try block

Program 28: Basic exception handling

```
#include <iostream>
using namespace std;
int main() {
    int a = 10, b = 0;
    try {
        if (b == 0)
            throw "Division by zero error!";   // throws a const char*
        cout << a / b << endl;
    }
    catch (const char *msg) {                  // matching handler
        cout << "Caught: " << msg << endl;
    }
    cout << "Program continues normally." << endl;
    return 0;
}
```

▶Output

```
Caught: Division by zero error!
Program continues normally.
```

Advanced Exception Handling Features

Feature	Syntax / behaviour
Multiple catch blocks	Several catch blocks after one try; matched top-to-bottom by TYPE
Catch-all handler	`catch(...) catches ANY exception type – must be placed LAST`
Rethrowing	`A plain throw; inside a catch passes the exception to an outer handler`
Exception with arguments	Throwing an object carries extra data to the handler
Uncaught exception	`If no handler matches, terminate() is called and the program aborts`

Program 29: Multiple catch blocks and catch-all

```
#include <iostream>
using namespace std;
void test(int x) {
    try {
```

```
        if (x == 1) throw 100;              // int
        if (x == 2) throw 'A';              // char
        if (x == 3) throw 9.5;              // double
    }
    catch (int e)    { cout << "Caught int: "    << e << endl; }
    catch (char e)   { cout << "Caught char: "   << e << endl; }
    catch (...)      { cout << "Caught something else" << endl; }
}
int main() { test(1); test(2); test(3); return 0; }
```

▶Output

```
Caught int: 100
Caught char: A
Caught something else
```

🔍 Easy Explanation

Order matters: catch blocks are matched top to bottom by type, so catch(...) must always come LAST. If you place it first it will swallow every exception and the specific handlers below it become unreachable.

🔍 Probable MCQs: Templates and Exception Handling

1. A function template allows:

a) Only integers to be used
b) The same logic to work for multiple data types
c) No parameters at all
d) Only string data
✓Answer: b) The same logic to work for multiple data types
Why: Templates generate type-specific versions from one generic definition.

2. Templates in C++ are resolved at:

a) Run time
b) Compile time
c) Link time only
d) Never resolved
✓Answer: b) Compile time
Why: The compiler instantiates a concrete version for each type used — zero runtime overhead.

3. Which is NOT a core component of the STL?

a) Containers
b) Algorithms
c) Iterators
d) Compilers
✓Answer: d) Compilers
Why: STL's three pillars are Containers, Algorithms and Iterators.

4. vector, list and deque belong to which STL container category?

a) Associative containers
b) Sequence containers
c) Container adapters
d) Iterators
✓Answer: b) Sequence containers

Why: They store elements in linear sequence; set/map are associative, stack/queue are adapters.

5. In exception handling, which keyword signals an error?

a) catch
b) try
c) throw
d) finally
✓Answer: c) throw
Why: throw raises the exception; try encloses risky code and catch handles it.

6. catch(...) is used to:

a) Catch only int exceptions
b) Catch any type of exception
c) Prevent throwing
d) Replace the try block
✓Answer: b) Catch any type of exception
Why: It is the catch-all handler and must be placed last.

7. If an exception is thrown but no matching catch exists, the program calls:

a) main() again
b) terminate()
c) throw() automatically
d) Nothing happens
✓Answer: b) terminate()
Why: An unhandled exception invokes terminate(), aborting the program.

8. Rethrowing an exception from inside a catch block uses:

a) throw;
b) rethrow;
c) catch;
d) retry;
✓Answer: a) throw;
Why: A bare 'throw;' propagates the current exception to an outer handler.

9. Which keyword can replace 'class' in a template declaration?

a) struct
b) typename
c) generic
d) auto
✓Answer: b) typename
Why: template<class T> and template<typename T> are interchangeable.