CPP CDAC CCEE 17 Session

I am preparing for the CDAC CCEE exam so please give me the best explanation for the exam preparation that explanation solve my each mcq Session 1: Getting Started
Lecture: Installation and Setup development environment The need of C++
 Features of C++
 C++ versus C
 History of C++
 Writing your first C++ program

Sure — here's a clear, exam-focused explanation of all topics in **Session 1: Getting Started with C++**. I'll explain concepts the way CDAC CCEE MCQs test them — definitions, reasoning points, differences, and tricky facts.

Use this as a conceptual guide to answer theory + MCQ questions confidently.

✅ Installation & Development Environment

To write and run C++ programs, you need:

✔ Compiler (translates C++ code → machine code)

Common compilers:

- GCC / G++ (Linux / Windows via MinGW)
- Clang
- MSVC (Visual Studio)
- Turbo C++ (legacy — still asked in exams)

✔ IDE (simplifies coding)

Popular IDEs:

- Code::Blocks
- Visual Studio
- Dev-C++
- CLion
- VS Code (with C++ extension)

✔ Compilation Steps (Important for MCQs)

1. Edit (write source code) → `.cpp`
2. Compile → object file `.o` / `.obj`

3. Link → executable `.exe` / binary

> ❗ MCQ Trap — Compilation vs Linking
> Compilation = syntax translation
> Linking = library + object merging

⭐ Why do we Need C++? (Motivation)

C (1972) → Good for system programming but lacked:

- Object Oriented Programming
- Abstraction and encapsulation
- Strong type safety
- Code reusability
- Real-world modeling

C++ was introduced to solve these gaps.

✔ C++ is used for:

- Operating systems (Windows, UNIX modules)
- Game engines (Unreal, Unity components)
- Compilers & Interpreters
- Database engines (MySQL)
- Embedded systems
- Real-time applications
- High-performance systems

> ❗ MCQ Concept
> C++ = **general purpose + high performance + OOP + low-level control**

🌟 Features of C++ (Very Important for MCQs)

C++ is also called a **multi-paradigm language** because it supports:

✔ 1) Object-Oriented Programming

- Classes & Objects
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

✔️ 2) Procedural programming (like C)

✔️ 3) Generic programming

- Templates (Function & Class templates)

✔️ 4) Low-level programming capability

- Pointers
- Memory management
- Direct hardware interaction

✔️ 5) Platform Independent (Source code)

Executable is OS-dependent.

✔️ 6) Faster execution than Java / Python

✔️ 7) Strong type checking

🆚 C vs C++ (Frequently Asked in MCQs)

Feature	C	C++
Paradigm	Procedural	Multi-paradigm
OOP support	❌ No	✔️ Yes
Data security	❌ No encapsulation	✔️ Encapsulation
Function overloading	❌ No	✔️ Yes
Operator overloading	❌ No	✔️ Yes
Templates	❌ No	✔️ Yes
Exception handling	❌ No	✔️ try-catch
Reference variables	❌ No	✔️ Yes
Namespace	❌ No	✔️ Yes
Inline functions	❌ No	✔️ Yes

> ❗ Trick MCQ
> C++ is **not purely object-oriented**
> because it allows:

- global functions
- procedural programming

🏛️ History of C++

Year	Event
1979	Bjarne Stroustrup at Bell Labs
1983	Name changed from *C with Classes* → C++
1985	First commercial release
1998	ISO C++ Standard (C++98)
2003	C++03
2011	C++11 (major upgrade)
2014	C++14
2017	C++17
2020	C++20

> ❗ Meaning of "C++"
> `++` operator means *increment* →
> i.e., **incremented version of C**

📝 Writing Your First C++ Program

🟢 Program

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

int main() {
    cout << "Hello, C++";
    return 0;
}
```

🔍 Explanation (Line-by-line)

`#include <iostream>`

Preprocessor directive — includes input/output library.

Runs **before compilation**.

`using namespace std;`

Avoids writing `std::cout`.

`int main()`

Execution entry point.

`cout << "Hello";`

Output statement.

`return 0;`

Indicates successful program termination.

⚡ Common MCQ Concepts from First Program

✔️ `main()` must return **int**
✔️ `execution starts from main()`
✔️ `#include` is processed by **preprocessor**
✔️ `cout` is an object of `ostream` class
✔️ `<<` is **insertion operator**

> ❗ MCQ Trap
> `cin` uses extraction operator `>>`

🎯 Quick MCQ Revision Points

1️⃣ C++ was developed by
👉 **Bjarne Stroustrup**

2️⃣ C++ original name
👉 **C with Classes**

3️⃣ C++ supports
👉 **OOP + procedural + generic programming**

4️⃣ Compilation produces
👉 object file

5️⃣ Linking produces
👉 executable file

6️⃣ `cout` belongs to
👉 namespace std

7️⃣ C++ is
👉 multi-paradigm language

👍 If you want, I can also provide:

✔️ Expected exam MCQs (with answers)
✔️ Previous-year styled questions
✔️ Concept-wise practice questions
✔️ Short notes for quick revision
✔️ Coding interview-type basics

Tell me what you want next:

1️⃣ MCQs from this topic
2️⃣ Short revision notes
3️⃣ Practice coding questions
4️⃣ Next Session explanation

Just reply with your preference 🙂

Sessions 2 & 3: Beginning with C++
Lecture: • C++Program structure
 Introduction of advanced C++ concepts and feature of C++ 17
 C++ Tokens Initialization
 Static Members Constant Members Expressions
ACTS, Pune
PG-DAC Page 2 of 5
Operators • Arithmetic Operator • Relational Operator • Logical Operator • Unary Operator • Ternary Operator • Assignment Operator

Great — let's continue with **Sessions 2 & 3: Beginning with C++** in the same exam-focused way. I'll explain concepts exactly how CDAC-CCEE asks them in MCQs, with definitions, logic points, and common traps.

✅ C++ Program Structure (Very Important)

A standard C++ program has:

1. **Documentation / Comments**
2. **Preprocessor directives**
3. **Global declarations**
4. **Class / Function definitions**
5. **main() function**

Example structure:

```cpp

#include <iostream>      // Preprocessor directive
using namespace std;     // Namespace

class Demo {             // Class definition
public:
    void show() {
        cout << "Demo Class";
    }
};

int main() {             // Entry point
    Demo d;
    d.show();
```

```
    return 0;
}
```

✔ MCQ Concepts

- Execution always starts from **main()**
- **`#include`** is handled by **preprocessor**
- **`cout`** belongs to **namespace std**
- A program can have multiple functions, but only **one main()**

🌟 Advanced C++ Concepts & C++17 Features (Exam Hotspot)

C++ evolved beyond OOP — supports:

✔ Procedural programming
✔ Generic programming (templates)
✔ Functional programming (lambda)

⭐ Key C++17 Features (asked in MCQs)

✔ 1) Structured Bindings

```
cpp

auto [a, b] = make_pair(10, 20);
```

Used for tuple / pair unpacking.

✔ 2) `std::optional`

Represents *value may or may not exist*.

✔ 3) `if constexpr`

Compile-time conditional evaluation.

✔ 4) File System Library

Used to manipulate directories & files.

✔ 5) Inline variables

Only one global instance created.

⚠️ MCQ Trap

C++17 emphasizes:

👉 performance
👉 compile-time evaluation
👉 type safety

NOT interpreted execution.

🧩 C++ Tokens (Lexical Elements)

Tokens = smallest individual units of program.

Types of Tokens

1️⃣ Keywords
2️⃣ Identifiers
3️⃣ Literals / Constants
4️⃣ Operators
5️⃣ Punctuation / Special symbols

✔️ Keywords

Reserved words — cannot be used as identifiers.

Examples:
`int, float, class, public, return, const, static`

✔️ Identifiers

Names for:

- variables
- functions
- classes
- objects

Rules:

✔️ Can contain letters, digits, underscore
✔️ Cannot start with digit
✔️ Case-sensitive
✔️ No special symbols

Example: **`totalMarks`**, **`employee_id`**

Invalid: **`2num`**, **`marks#`**

✔️ Constants / Literals

Types:

- Integer → `10`
- Float → `3.14`
- Character → `'A'`
- String → `"Hello"`
- Boolean → `true / false`

🟢 Initialization

Assigning initial value to a variable.

✔️ Direct Initialization

```cpp
int x(10);
```

✔️ Copy Initialization

```cpp
int x = 10;
```

✔️ Brace Initialization (C++11 onward)

```cpp
int x{10};
```

Prevents narrowing conversions — asked in MCQs.

🟦 Static Members (Class Level Members)

Used when a property is **shared across all objects**.

```cpp
class Test {
public:
    static int count;
};

int Test::count = 0;
```

✔️ Key Properties

- Only **one copy exists** (shared)
- Belongs to **class**, not objects
- Accessed using class name:

```
cpp

Test::count;
```

⚠️ MCQ Trap

Static member function:

- can access **only static members**
- cannot access non-static data directly

🟥 Constant Members

Used when value cannot change.

```
cpp

const int maxLimit = 100;
```

✔️ Constant Member Function

```
cpp

class Demo {
public:
    int x;
    int get() const {
        // x++; ❌ Not allowed
        return x;
    }
};
```

Meaning:

👉 function cannot modify object data

🔢 Expressions

Expression = combination of

- variables
- constants

- operators

Example:

```
cpp

int z = (a + b) * c;
```

Evaluated using **operator precedence**.

🧮 Operators in C++

1️⃣ Arithmetic Operators

`+ - * / %`

```
cpp

a + b
a - b
a * b
a / b
a % b
```

⚠️ Rules

- `%` works only on **integers**
- `/` performs integer division if both operands are int

Example:

`7 / 2 = 3`

2️⃣ Relational Operators

`> < >= <= == !=`

Return boolean result.

3️⃣ Logical Operators

`&& || !`

Support **short-circuit evaluation**.

4️⃣ Unary Operators

Operate on **single operand**.

```
++  --  +  -  sizeof
```

✔️ Pre vs Post Increment

```cpp
int x = 5;

int a = ++x;   // x=6 then a=6
int b = x++;   // b=6 then x=7
```

5️⃣ Ternary Operator (Conditional)

```cpp
int result = (a > b) ? a : b;
```

Equivalent to if-else.

6️⃣ Assignment Operators

```
=  +=  -=  *=  /=  %=
```

Example:

```cpp
x += 5;   // x = x + 5
```

🎯 High-Value MCQ Summary

✔️ Static member belongs to
👉 class, not object

✔️ Constant member function cannot
👉 modify data members

✔️ % operator is invalid for
👉 float operands

✔️ Execution begins from
👉 main()

✔️ == compares values
✔️ = assigns values

✔️ Post-increment returns original value first

👍 Tell me what you want next:

1️⃣ MCQs with answers from this topic
2️⃣ Short handwritten-style notes
3️⃣ Practice coding questions
4️⃣ Next session explanation (Functions, Classes, OOP basics)

Just reply with your preference 🙂

> MCQs with answers from this topic 20

Sure — here are **20 high-quality, exam-style MCQs** from Sessions 2 & 3 with answers and brief explanations (focused for CDAC-CCEE).

✅ MCQs — C++ Program Structure, Tokens, Initialization, Static & Const, Expressions, Operators

1. Which part of a C++ program is processed before compilation?

A) main()
B) Functions
C) Preprocessor directives
D) Class declarations

Answer: C

✔ `#include`, `#define` etc. are handled by **preprocessor**

---

2. Which of the following is NOT a valid C++ token?

A) Keyword
B) Identifier
C) Object
D) Literal

Answer: C

✔ Tokens = smallest units
👉 keywords, identifiers, literals, operators, symbols

---

3. Which of the following is a valid identifier?

A) 2value
B) value_2

C) value#2
D) value-2

Answer: B

✔️ Identifier cannot start with digit or contain special symbols.

---

4. In which initialization style is narrowing conversion prevented?

A) Copy initialization
B) Direct initialization
C) Brace initialization `{}`
D) Runtime initialization

Answer: C

✔️ `{}` prevents loss of precision.

---

5. Which member is shared among all objects of a class?

A) const member
B) static member
C) private member
D) inline member

Answer: B

✔️ Only one copy exists → class-level.

---

6. Which of the following is TRUE about static member functions?

A) They can access only static members
B) They can access both static & non-static
C) They cannot be called using class name
D) They execute before main()

Answer: A

✔️ No implicit `this` pointer.

---

7. A constant member function —

A) cannot return a value
B) cannot be called by objects
C) cannot modify data members
D) must be static

Answer: C

✔️ `const` after function ensures data safety.

---

8. Which operator has the highest precedence?

A) *
B) ++ (unary)
C) /
D) +

Answer: B

✔️ Unary operators > multiplicative > additive.

---

9. What will be the value of `x` after execution?

```cpp
int x = 5;
int y = x++;
```

A) x = 4
B) x = 5
C) x = 6
D) Error

Answer: C

✔️ Post-increment → update after assignment.

---

10. Which operator returns true only if both conditions are true?

A) ||
B) &&
C) !
D) ==

Answer: B

✔️ Logical AND.

---

11. Which operator is also known as conditional operator?

A) ?:
B) ->
C) sizeof
D) **

Answer: A

✔️ Ternary operator.

---

12. Which expression is equivalent to `x += y;`?

A) x = y + y;
B) x = x + y;
C) x = x * y;
D) x = y - x;

Answer: B

✔ Assignment shorthand.

---

13. Which of the following is invalid use of modulus operator (%) ?

A) 10 % 3
B) 15 % 4
C) 7 % 2
D) 5.5 % 2

Answer: D

✔ `%` works only on integers.

---

14. Which of the following is a relational operator?

A) =
B) ==
C) &&
D) +=

Answer: B

✔ `==` tests equality
`=` is assignment.

---

15. What is the output?

```cpp
int a = 10, b = 20;
int c = (a > b) ? a : b;
```

A) 10
B) 20
C) 30
D) Garbage

Answer: B

✔ Ternary → returns maximum value.

---

16. Which of the following is TRUE about expressions?

A) Every expression must contain variables
B) Expressions always produce a value
C) Expressions cannot contain constants
D) Expressions cannot be nested

Answer: B

✔️ Expression always yields a result.

---

17. What is the correct order of evaluation?

A) Logical → Arithmetic → Relational
B) Arithmetic → Relational → Logical
C) Relational → Arithmetic → Logical
D) Logical → Relational → Arithmetic

Answer: B

✔️ Precedence order:

Arithmetic > Relational > Logical.

---

18. Which of the following belongs to punctuation tokens?

A) class
B) ;
C) &&
D) ++

Answer: B

✔️ Semicolon terminates statements.

---

19. Which statement about `static` members is FALSE?

A) Only one copy exists
B) Shared across objects
C) Must be accessed using object
D) Defined outside class

Answer: C

✔️ Can be accessed using **class name**.

---

20. Which of the following is a unary operator?

A) +
B) &&
C) >=
D) ?

Answer: A

✔️ Unary plus/minus, ++, --, sizeof etc.

🎯 Quick Revision Summary

- Static → class level, shared, one copy
- Const function → cannot modify data
- `%` works only on integers
- Post-increment → returns old value first
- Arithmetic > Relational > Logical precedence
- Ternary `?:` acts like if-else

If you want, I can also give:

👉 20 tougher MCQs
👉 Numericals / output-based questions
👉 Short notes for revision
👉 Next Session MCQs

Just tell me 👍

Session 4: Conditional and Looping Statements
Lecture: • If, else if, switch
• for loop
• while loop
• do while loop
 Jump statement (break, continue& returnkeyword) Arrays Declaration and initialization of an array 1-D and 2-D arrays

Great — here is the **Session 4 explanation for Conditional Statements, Loops, Jump Statements, and Arrays**, written in an exam-focused, MCQ-oriented manner for CDAC-CCEE.

I'll cover concepts + rules + common traps.

✅ Conditional Statements

✔️ `if` Statement

Executes a block if condition is true.

```cpp
if (x > 0)
    cout << "Positive";
```

Condition must evaluate to **boolean**.

✔️ `if – else`

```cpp
cpp

if (x % 2 == 0)
    cout << "Even";
else
    cout << "Odd";
```

✔️ `else if` Ladder

```cpp
cpp

if (m >= 90) cout << "A";
else if (m >= 75) cout << "B";
else if (m >= 60) cout << "C";
else cout << "Fail";
```

Checked top → bottom.

✔️ `switch` Statement (Very Important for MCQs)

Used when multiple discrete choices exist.

```cpp
cpp

switch(ch) {
    case 'A': cout << "Apple"; break;
    case 'B': cout << "Ball";  break;
    default:  cout << "Other";
}
```

Rules

✔️ **`switch`** works with
`int`, `char`, `enum`, `short`, `long`, `bool` (and **`std::string`** in C++11)

✔️ **`case`** values must be:

- constant expression
- unique (no duplicates)
- same type as expression

✔️ **`break`** prevents fall-through

⚠️ Common MCQ Trap

Missing **break** causes:

👉 execution continues to next case.

🔁 Loops in C++

Loops repeat a block until condition fails.

✔️ `for` Loop

```
cpp

for (int i = 1; i <= 5; i++)
    cout << i;
```

Has 3 parts:

1. initialization
2. condition
3. increment/decrement

All three are optional.

```
cpp

for( ; ; )   // infinite loop
```

✔️ `while` Loop

Entry-controlled loop.

```
cpp

while (i <= 5) {
    cout << i;
    i++;
}
```

Condition checked first.

✔️ `do – while` Loop

Exit-controlled loop.

```
cpp

do {
    cout << i;
```

```
    i++;
} while (i <= 5);
```

Block executes **at least once**.

⚠️ MCQ Trap

Difference between **`while`** & **`do-while`**:

while	do-while
condition checked first	condition checked later
may not execute	executes at least once
no semicolon at end	semicolon required

🟨 Jump Statements

✔️ `break`

Exits:

- loop
- switch

```
cpp

if(x == 5) break;
```

✔️ `continue`

Skips current iteration.

```
cpp

if(i % 2 == 0)
    continue;
```

Control goes to loop update step.

✔️ `return`

Exits function and returns value.

```cpp
return 0;
```

Also stops loop if inside function.

📦 Arrays

Array = collection of elements of same data type stored in **contiguous memory**.

✔️ Declaring Array

```cpp
int a[5];
```

Size must be a **constant expression**.

✔️ Initialization

```cpp
int a[5] = {1,2,3,4,5};
```

Partial initialization:

```cpp
int a[5] = {1,2};   // remaining become 0
```

Implicit size:

```cpp
int a[] = {10,20,30};
```

Size = number of elements.

⚠️ MCQ Trap

Array index starts at **0**

Last element index = **`size – 1`**

🟢 1-D Array Example

```cpp
int a[5] = {2,4,6,8,10};

for(int i=0;i<5;i++)
    cout << a[i];
```

🔷 2-D Array (Matrix)

```cpp
int m[2][3] = {
    {1,2,3},
    {4,5,6}
};
```

Rows = 2
Columns = 3

Access:

```cpp
cout << m[1][2];   // output = 6
```

🎯 Important Exam Points & Traps

✔️ **`switch`** does NOT support floating values
✔️ missing **`break`** causes fall-through
✔️ **`do-while`** executes at least once
✔️ **`continue`** skips iteration
✔️ **`break`** exits loop completely
✔️ array index starts from 0
✔️ uninitialized array → garbage values
✔️ 2-D array stored in **row-major order**

If you want, I can provide:

👉 20 MCQs from this session
👉 Output-based loop questions
👉 Array tracing questions
👉 Short handwritten-style revision notes

Tell me what you want next 👍

> 20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 4 (Conditionals, Loops, Jump Statements & Arrays)** with answers and brief explanations — designed in CDAC-CCEE style.

✅ MCQs — Conditional Statements, Loops, Jump Statements & Arrays

1. Which control statement ensures that at least one iteration of the loop is executed?

A) for
B) while
C) do-while
D) infinite loop

Answer: C

✔ `do-while` is exit-controlled → executes first, condition later.

2. What will be the output?

```cpp
int x = 10;
if (x = 5)
    cout << "True";
else
    cout << "False";
```

A) True
B) False
C) Compilation error
D) Runtime error

Answer: A

✔ `=` assigns 5 and returns **true (non-zero)**.
(very common MCQ trap)

3. Which statement about switch is TRUE?

A) Case values may be variables
B) Duplicate case labels are allowed
C) Case values must be constant expressions
D) Break is mandatory in every case

Answer: C

✔ Case labels must be constant and unique.

4. What happens if `break` is missing in a switch case?

A) Program crashes
B) Compilation error
C) Control moves to next case
D) switch terminates automatically

Answer: C

✔ This is called **fall-through**.

---

5. Output of the following loop?

```
cpp

for (int i = 0; i < 3; i++);
    cout << i;
```

A) 0 1 2
B) 3
C) Compilation error
D) Undefined

Answer: B

✔ Semicolon ends loop.
`cout` executes once with `i = 3`.

---

6. Which loop is most suitable when number of iterations is known?

A) while
B) do-while
C) for
D) goto

Answer: C

✔ `for` loop preferred for fixed iterations.

---

7. Output?

```
cpp

int i = 1;
while (i < 3)
    cout << i++;
```

A) 1
B) 12
C) 23
D) Infinite loop

Answer: B

✔ prints 1 then 2.

---

8. Which loop may never execute?

A) for
B) while
C) do-while
D) None

Answer: B

✔ while is entry-controlled.

---

9. Which jump statement skips current iteration?

A) break
B) stop
C) return
D) continue

Answer: D

✔ **`continue`** → jumps to loop update step.

---

10. Which statement immediately terminates a loop?

A) exit
B) break
C) return 1
D) stop

Answer: B

✔ **`break`** exits current loop/switch.

---

11. Output?

```cpp
for (int i = 1; i <= 5; i++) {
    if (i == 3)
        continue;
    cout << i;
}
```

A) 12345
B) 1345
C) 1245
D) 1234

Answer: B

✔️ `3` is skipped.

---

12. Output of do-while?

```cpp
int n = 5;
do {
    cout << n;
} while (n < 5);
```

A) No output
B) 5
C) Infinite loop
D) 0

Answer: B

✔️ executes once even if condition false.

---

13. Which declaration is valid?

A) int a[5];
B) int[5] a;
C) array a[5];
D) int a[];

Answer: A

✔️ Syntax → `datatype arrayName[size];`

---

14. What is the index of the last element in `int a[10];`?

A) 9
B) 10
C) 8
D) 11

Answer: A

✔️ index = `size - 1`.

---

15. Output?

```cpp
int a[5] = {1,2};
cout << a[2];
```

A) 1
B) 2
C) 0
D) Garbage

Answer: C

✔️ Remaining elements auto-initialized to **0**.

---

16. What is stored in a 2-D array?

A) Rows stored randomly
B) Stored in column-major order
C) Stored in row-major order
D) Stored in heap only

Answer: C

✔️ C++ stores multi-dimensional arrays row-wise.

---

17. Which is correct 2-D declaration?

A) int a[2,3];
B) int a(2)(3);
C) int a[2][3];
D) int[2][3] a;

Answer: C

✔️ Proper syntax.

---

18. Output?

```cpp
int m[2][2] = {{1,2},{3,4}};
cout << m[1][0];
```

A) 1
B) 2
C) 3
D) 4

Answer: C

✔️ Row-1, Col-0 → value = 3.

---

19. What happens if array index goes out of bounds?

A) Compilation error
B) Runtime exception

C) Undefined behavior
D) Value becomes zero

Answer: C

✔️ C++ does **not** perform bounds checking.

---

20. Which keyword is used to exit a function?

A) break
B) stop
C) return
D) quit

Answer: C

✔️ `return` ends function execution.

🎯 Quick Revision Memory Tips

- `do-while` executes at least once
- missing `break` → fall-through
- `continue` skips iteration
- `break` exits loop / switch
- arrays are 0-indexed
- 2-D arrays → row-major order
- uninitialized elements → 0 (if partially initialized)
- out-of-bound access → undefined behavior

If you want, I can also provide:

👉 Output-tracing questions on loops & arrays
👉 20 tougher MCQs from this session
👉 Practice coding problems
👉 Next session explanation

Just tell me 👍

Session 5: Functions in C++
Lecture
 Different forms of functions Function prototyping
 Call by Reference
 Inline Functions Math library functions etc.

Sure — here's a clear, exam-oriented explanation of **Session 5: Functions in C++**, written to help you solve conceptual and MCQ questions confidently.

I will cover concepts, rules, tricky points, and common exam traps.

✅ What is a Function?

A function is a block of code that performs a specific task and can be reused.

Benefits:

✔ Code reusability
✔ Modularity
✔ Easier maintenance
✔ Reduces redundancy

🟢 Different Forms of Functions

Functions may differ by:

- return type
- parameters
- presence/absence of arguments

✔ 1) Function with no arguments & no return value

```cpp
cpp

void show() {
    cout << "Hello";
}
```

Called as:

```cpp
cpp

show();
```

✔ 2) Function with arguments & no return value

```cpp
cpp

void add(int a, int b) {
    cout << a + b;
}
```

✔️ 3) Function with no arguments & return value

```cpp
int getValue() {
    return 10;
}
```

✔️ 4) Function with arguments & return value

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

> These four forms are **frequently asked in MCQs**.

📄 Function Prototype (Declaration)

Tells the compiler:

- function name
- return type
- parameter types

```cpp
int add(int, int);
```

Prototype must appear **before function call**.

✔️ Why prototype is needed?

- enables type checking
- prevents wrong arguments

⚠️ MCQ Trap

Argument names in prototype are **optional**:

```cpp
int add(int, int);   // valid
```

Only data types are necessary.

🔁 Function Call Mechanisms

✔️ Call by Value (default in C++)

A copy of variable is passed.

```cpp
void fun(int x) {
    x = 20;
}
```

Does NOT affect actual variable.

✔️ Call by Reference (Important Topic)

We pass variable reference.

```cpp
void fun(int &x) {
    x = 20;
}
```

Changes reflect in actual variable.

✔️ Key Points

- uses reference operator **&**
- no copy created
- faster and memory efficient

⚠️ MCQ Trap

Reference must be initialized during declaration:

```cpp
int &r;   // ❌ invalid
```

⚡ Inline Functions (Performance-Oriented Feature)

Used for small, frequently called functions.

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

Instead of calling function,
compiler **expands code at call site**.

✔️ Advantages

- removes function call overhead
- improves performance

✔️ When inline is ignored?

Compiler may NOT inline if function:

- is recursive
- contains loops
- contains static variables
- is too large

👉 inline is a **request**, not guarantee

🧮 Math Library Functions

Defined in:

```cpp
#include <cmath>
```

Common functions:

Function	Meaning
`sqrt(x)`	square root
`pow(a,b)`	a^b
`abs(x)`	absolute value (int)
`fabs(x)`	absolute (float/double)
`ceil(x)`	round up
`floor(x)`	round down
`sin(x)` `cos(x)` `tan(x)`	trigonometric

⚠️ MCQ Notes

- `abs()` for integers
- `fabs()` for floating values
- angles are in **radians**, not degrees

🎯 Important Exam Points & Tricky Concepts

✔️ Function prototype enables type checking
✔️ Call by value → original data unchanged
✔️ Call by reference → changes reflect
✔️ Reference must be initialized
✔️ Inline functions expand code
✔️ Inline is only a suggestion
✔️ Math functions belong to `<cmath>`
✔️ `pow(2,3)` returns `8` (double type)

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based & tracing questions
👉 Practice problems on functions
👉 Short revision notes

Just tell me 👍

> 20 MCQs from this session

Sure — here are **20 exam-focused MCQs** from **Session 5: Functions in C++** with answers and short explanations, designed in CDAC-CCEE style.

✅ MCQs — Functions, Prototypes, Call by Reference, Inline, Math Library

1. What is the main purpose of a function in C++?

A) Reduce execution time
B) Avoid compilation errors
C) Achieve code reusability
D) Improve memory usage

Answer: C

✔️ Functions promote modular & reusable code.

---

2. Which of the following is a valid function declaration?

A) int add(a, b);
B) int add(int, int);
C) int add(); int;
D) add(int, int);

Answer: B

✔️ Only data types are required in prototype.

---

3. Which form of function is shown below?

```cpp
void show();
```

A) No arguments, no return value
B) Arguments, no return value
C) No arguments, return value
D) Arguments, return value

Answer: A

---

4. Which of the following passes arguments by reference?

```cpp
void fun(? x) { x = 10; }
```

A) int x
B) int *x
C) int &x
D) ref x

Answer: C

✔️ `&` in parameter = reference.

---

5. In call by value —

A) changes affect actual variable
B) new copy of variable is created
C) memory is shared
D) reference is passed

Answer: B

---

6. Which statement about references is TRUE?

A) Reference can be null
B) Reference must be initialized
C) Reference can be reseated
D) Reference stores new copy

Answer: B

✔️ Must be initialized at declaration.

---

7. Output?

```cpp
void fun(int &x) {
    x = 50;
}
int a = 10;
fun(a);
cout << a;
```

A) 10
B) 0
C) 50
D) Garbage

Answer: C

✔️ Call by reference modifies actual variable.

---

8. Inline functions are mainly used to —

A) reduce memory usage
B) reduce function call overhead
C) increase recursion
D) improve portability

Answer: B

---

9. Which keyword is used to define inline functions?

A) fast
B) inline
C) macro
D) static

Answer: B

---

10. Which case is NOT suitable for inline function?

A) Small function
B) Recursive function
C) One-line function
D) Frequently called function

Answer: B

✔️ Compiler generally rejects recursive inline.

---

11. Where are inline functions expanded?

A) In stack
B) In heap memory
C) At function call location
D) In register

Answer: C

✔️ Code is substituted directly.

---

12. What is required before calling a function from main()?

A) return statement
B) function prototype
C) namespace statement
D) inline specifier

Answer: B

✔️ Prototype enables type checking.

---

13. Which header file contains math functions?

A) <math.h> only
B) <cmath>
C) <math>
D) <stdlib>

Answer: B

✔️ Standard C++ header.

---

14. What is the return type of `sqrt()`?

A) int
B) float
C) double
D) depends on input

Answer: C

✔️ Most math functions return **`double`**.

15. Which function gives absolute value of a floating number?

A) abs()
B) fabs()
C) labs()
D) abf()

Answer: B

✔ `abs()` → integers
✔ `fabs()` → floating values

---

16. What is the result of `pow(2,3)`?

A) 6
B) 8
C) 9
D) 16

Answer: B

✔ $2^3 = 8$.

---

17. Which math function rounds value upward?

A) floor()
B) trunc()
C) ceil()
D) round()

Answer: C

✔ ceil → smallest integer ≥ x.

---

18. Which of the following is TRUE about function overloading?

A) Same name, different parameters
B) Same name, same parameters
C) Different name, same parameters
D) Different return type only

Answer: A

✔ Parameters must differ.

---

19. What happens if function prototype is missing?

A) Compiler assumes default return type
B) Function cannot be called

C) No type checking of arguments
D) Program terminates

Answer: C

✔️ Older behavior → may compile but unsafe.

---

20. Which of the following is NOT a valid advantage of inline functions?

A) Faster execution
B) Less function call overhead
C) Reduced program size
D) Code expansion at call site

Answer: C

✔️ Inline increases code size.

🎯 Quick Revision Notes

- Function prototype enables type safety
- Call by value → copy passed
- Call by reference → `&`, modifies original
- Reference must be initialized
- Inline expands code at call site
- Inline not suitable for recursion / large code
- `<cmath>` contains math functions
- `pow()` & `sqrt()` return double

If you want, I can also provide:

👉 Output-tracing questions
👉 Coding practice problems
👉 Trick MCQs (advanced)
👉 Next session explanation

Just tell me 👍

> 20 MCQs from this session

Sure — here are **20 NEW exam-level MCQs** from **Session 5: Functions in C++** (different from the previous set), with answers and short explanations — aligned to CDAC-CCEE style.

✅ MCQs — Functions, Prototypes, Call Mechanisms, Inline, Math Library

1. A function that does not return any value must be declared as —

A) null
B) empty
C) void
D) nil

Answer: C

✔ `void` → no return value.

2. Which of the following statements is TRUE?

A) Function name can start with a digit
B) Function name must be a keyword
C) Function name follows identifier rules
D) Function name must be uppercase

Answer: C

✔ Same naming rules as variables.

3. What is the correct function definition?

A) int add(int a, b) { }
B) add(int a, int b) { }
C) int add(int a, int b) { }
D) void int add(a,b) { }

Answer: C

✔ Both parameters must specify type.

4. Which of the following is NOT part of a function signature?

A) Function name
B) Parameter list
C) Return type
D) Number of parameters

Answer: C

✔ Function signature ignores return type.

5. The default mechanism of parameter passing in C++ is —

A) Call by reference
B) Call by address
C) Call by value
D) Call by copy reference

Answer: C

---

6. What will happen in call by value?

A) Actual value is modified
B) Copy of value is passed
C) Memory is shared
D) Reference is passed

Answer: B

---

7. Which parameter declaration enables call-by-reference?

A) int *x
B) int x&
C) int &x
D) ref int x

Answer: C

✔️ & with parameter name → reference.

---

8. Which of the following is NOT allowed for reference variables?

A) Must be initialized
B) Cannot be NULL
C) Can be reseated to new variable
D) Acts as alias of variable

Answer: C

✔️ Reference cannot be reassigned.

---

9. Which of the following is TRUE about inline functions?

A) They are executed faster because code expands
B) They allocate separate stack frame
C) They always reduce program size
D) They must not have return type

Answer: A

✔️ Expansion removes call overhead.

---

10. Inline function expansion happens —

A) At compile time
B) At runtime
C) In linker
D) Inside memory allocator

Answer: A

---

11. Which function type is MOST suitable for inline?

A) Long computational function
B) Function with loops
C) Recursive function
D) Small frequently-used function

Answer: D

---

12. Which case may cause compiler to IGNORE inline request?

A) Function contains return statement
B) Function is used once
C) Function is too large
D) Function has one parameter

Answer: C

✔ Large functions are not inlined.

---

13. What is the purpose of a function prototype?

A) Improve runtime performance
B) Inform compiler about function type
C) Reduce memory consumption
D) Convert code to binary

Answer: B

---

14. Function prototype must appear —

A) After main
B) Before function call
C) Inside loop
D) Inside class only

Answer: B

---

15. Which of the following statements is TRUE about default arguments?

A) They must appear at beginning of parameter list
B) They must appear at end of parameter list

C) They can appear anywhere
D) They are not allowed in C++

Answer: B

✔️ Parameters after default must also be default.

---

16. What is the output?

```cpp
int fun(int a = 5, int b = 10) {
    return a + b;
}
cout << fun(2);
```

A) 7
B) 10
C) 12
D) 15

Answer: C

✔️ b uses default → 2 + 10 = 12.

---

17. Math functions like `sqrt()` and `pow()` return value of type —

A) int
B) float
C) double
D) long double

Answer: C

✔️ Default return type → double.

---

18. Which function returns largest integer less than or equal to value?

A) ceil(x)
B) round(x)
C) floor(x)
D) trunc(x)

Answer: C

---

19. Which function calculates x raised to power y?

A) exp(x,y)
B) pow(x,y)
C) power(x,y)
D) raise(x,y)

Answer: B

---

20. Which header must be included to use math library functions?

A) <math>
B) <cmath>
C) <mth.h>
D) <stdmath>

Answer: B

✔️ Standard C++ math header.

🎯 Quick Revision Pointers

- Function prototype enables type checking
- Default passing → call-by-value
- Reference (`&`) modifies original value
- Reference cannot be NULL or re-assigned
- Inline expands code at compile time
- Compiler may ignore inline for large/recursive functions
- `<cmath>` contains math functions

If you want, I can also give:

👉 Output-based function tracing questions
👉 Coding practice problems
👉 Tough conceptual MCQs
👉 Next session explanation

Just tell me 👍

Session 8: OOP Concepts
Lecture
 Discussion on object oriented concepts Classes and Objects, Access Specifiers, Overloading, Inheritance,Polymorphism
 Namespaces

Sure — here's a clear, exam-focused explanation of **Session 8: Object-Oriented Programming Concepts** for CDAC-CCEE. I'll explain concepts, rules, terminology, examples, and common MCQ traps.

✅ What is Object-Oriented Programming (OOP)?

OOP is a programming paradigm that organizes programs using **objects** (real-world entities) and **classes** (their blueprints).

Goals of OOP:

✔️ Reusability
✔️ Modularity
✔️ Data security
✔️ Maintainability
✔️ Real-world modeling

C++ supports:

- Procedural programming
- Object-oriented programming
- Generic programming

Hence it is **multi-paradigm** (not purely OOP).

⭐ Core OOP Concepts (Very Important)

The four fundamental OOP pillars are:

1️⃣ Encapsulation
2️⃣ Abstraction
3️⃣ Inheritance
4️⃣ Polymorphism

🟢 Encapsulation

Binding **data + functions** together inside a class.

```cpp
cpp


class Student {
private:
    int marks;

public:
    void setMarks(int m) { marks = m; }
};
```

Benefits:

✔️ Data hiding
✔️ Security
✔️ Controlled access

🟣 Abstraction

Showing **essential features only**, hiding implementation details.

Example:

```
cpp

class Car {
public:
    void startEngine();   // What it does – visible
};
```

User doesn't know *how* engine starts internally.

🟡 Classes and Objects

✔️ Class = Blueprint / Template

```
cpp

class Employee {
public:
    int id;
    string name;
};
```

No memory allocated until object is created.

✔️ Object = Instance of class

```
cpp

Employee e1;
```

Memory allocated here.

🔐 Access Specifiers

Control visibility & accessibility of class members.

Specifier	Accessible From
public	Anywhere
private	Within class only
protected	Class + Derived class

Default access in C++ classes is:

👉 **private** (MCQ favorite)

🔁 Function Overloading

Same function name, different parameters.

```
cpp

int add(int a, int b);
double add(double a, double b);
```

Rules:

✔️ parameter type or count must differ
❌ return type alone cannot differ

🧬 Inheritance

Mechanism of acquiring properties of one class into another.

```
cpp

class Parent {};
class Child : public Parent {};
```

Types of inheritance:

- Single
- Multilevel
- Multiple
- Hierarchical
- Hybrid

Access mode matters (important for MCQs)

Base → Derived	Public	Protected	Private
public members	public	protected	private
protected members	protected	protected	private
private members	❌ not inherited		

🌀 Polymorphism

“Poly” = many
“Morph” = forms

Same function behaves differently.

Two types:

✔️ Compile-time Polymorphism

(Static / Early binding)

- Function overloading
- Operator overloading
- Inline expansion

✔️ Runtime Polymorphism

(Dynamic / Late binding)

Achieved via **virtual functions** & base class pointers.

```cpp
class A {
public:
    virtual void show() { cout << "A"; }
};

class B : public A {
public:
    void show() { cout << "B"; }
};
```

🧱 Namespaces

Avoids naming conflicts between identifiers.

```cpp
namespace Test {
    int x = 10;
}
```

Access:

```cpp
cout << Test::x;
```

Using directive:

```cpp
using namespace std;
```

MCQ Traps

✔️ Namespace groups classes, variables, and functions
✔️ `std` is the default C++ namespace
✔️ Multiple namespaces can exist in same program

🎯 High-Value MCQ Concepts

- C++ is **not purely OOP** (allows procedural code)
- Default class access = **private**
- Encapsulation = data hiding
- Abstraction = highlight essential features
- Function overloading → compile-time polymorphism
- Virtual function → runtime polymorphism
- Private members are **not inherited**
- Namespace prevents name conflicts
- Object = instance of class

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based OOP questions
👉 Short revision notes
👉 Coding examples

Tell me what you want next 👍

> 20 MCQs from this session

Sure — here are **20 exam-oriented MCQs from Session 8 (OOP Concepts)** with answers and brief explanations, designed in CDAC-CCEE style.

✅ MCQs — OOP Concepts, Classes, Objects, Access Specifiers, Inheritance, Polymorphism, Namespaces

1. Which of the following best defines encapsulation?

A) Hiding implementation details
B) Binding data and functions together
C) Reusing existing code
D) Creating multiple objects

Answer: B

✔️ Encapsulation = wrapping data + methods inside class.

---

2. Abstraction refers to —

A) Code reuse
B) Hiding unnecessary details
C) Function reuse
D) Data duplication

Answer: B

---

3. Which of the following is TRUE?

A) Class allocates memory
B) Object allocates memory
C) Namespace allocates memory
D) Function allocates memory

Answer: B

✔️ Memory is created when object is created.

---

4. What is the default access specifier in a C++ class?

A) public
B) private
C) protected
D) package

Answer: B

---

5. Which members are NOT inherited in derived classes?

A) public
B) protected
C) private
D) static

Answer: C

✔️ Private members are inaccessible outside base class.

---

6. Which of the following is an example of compile-time polymorphism?

A) Virtual functions
B) Function overloading
C) Abstract classes
D) Dynamic binding

Answer: B

---

7. Runtime polymorphism is achieved using —

A) Operator overloading
B) Function overloading
C) Virtual functions
D) Constructors

Answer: C

✔️ Requires base pointer/reference + virtual function.

---

8. Which inheritance type involves one base class and one derived class?

A) Multiple
B) Hierarchical
C) Single
D) Hybrid

Answer: C

---

9. Which of the following causes name conflicts in large programs?

A) Classes
B) Objects
C) Functions
D) Global identifiers

Answer: D

✔️ Namespaces are used to prevent this.

---

10. Which feature allows adding new features to existing class?

A) Polymorphism
B) Inheritance
C) Encapsulation
D) Abstraction

Answer: B

---

11. Which operator is used to access namespace members?

A) .
B) ->
C) ::
D) ::

Answer: C

✔️ Scope resolution operator.

---

12. In function overloading, functions must differ in —

A) Return type only
B) Number or type of parameters
C) Function name
D) Access specifier

Answer: B

✔️ Return type alone cannot distinguish overload.

---

13. Which of the following is TRUE about polymorphism?

A) Same function behaves differently
B) Same object behaves differently
C) Same class is reused
D) Same namespace is reused

Answer: A

---

14. Which feature provides security to data?

A) Inheritance
B) Abstraction
C) Encapsulation
D) Polymorphism

Answer: C

✔️ Data hiding through private members.

---

15. Which keyword enables runtime polymorphism?

A) static
B) inline
C) virtual
D) override

Answer: C

---

16. Which access specifier allows access in derived classes but not outside?

A) public
B) private
C) protected
D) default

Answer: C

17. Which statement about namespaces is FALSE?

A) They prevent identifier conflicts
B) They group logically related code
C) Only one namespace can exist in a program
D) std is a namespace

Answer: C

✔ Multiple namespaces are allowed.

---

18. Which of the following is an example of hierarchical inheritance?

A) One base → one derived
B) Multiple base → one derived
C) One base → many derived
D) Many base → many derived

Answer: C

---

19. Which of the following statements is TRUE?

A) C++ is a purely object-oriented language
B) C++ supports only OOP
C) C++ supports procedural + OOP
D) C++ does not support OOP

Answer: C

✔ C++ is multi-paradigm.

---

20. Late binding occurs —

A) At compile time
B) At runtime
C) During linking
D) During preprocessing

Answer: B

✔ Runtime polymorphism = late binding.

🎯 Quick Revision Pointers

- Encapsulation = binding data + methods
- Abstraction = show essential features
- Default class access → private
- Private members not inherited

- Overloading → compile-time polymorphism
- Virtual functions → runtime polymorphism
- Namespace avoids naming conflicts
- Object allocates memory, not class

If you want, I can also share:

👉 Output-based OOP questions
👉 Coding practice problems
👉 Tough conceptual MCQs
👉 Next session explanation

Just tell me 👍

Session 9: Constructors and Destructor
Lecture
 Constructors Parameterized constructors Multiple constructors in class Dynamic initialization of objects Copy Constructors Destructor

Sure — here's an exam-focused explanation of **Session 9: Constructors & Destructor in C++**, written to help you handle both theory and MCQ questions confidently.

✅ What is a Constructor?

A constructor is a **special member function of a class** that is automatically called when an object is created.

```cpp
class Demo {
public:
    Demo() {
        cout << "Constructor called";
    }
};
```

✔️ Key Properties (Frequently Asked in MCQs)

- Name is same as class name
- No return type (not even void)
- Automatically invoked
- Defined in public section (generally)
- Can be overloaded
- Cannot be virtual

- Cannot be inherited (but can be called in derived class)

🟢 Types of Constructors

✔️ Default Constructor

A constructor with **no arguments**.

```
cpp

class A {
public:
    A() {
        cout << "Default Constructor";
    }
};
```

If no constructor is defined → compiler creates **implicit default constructor**.

✔️ Parameterized Constructor

Constructor with parameters.

```
cpp

class Student {
    int marks;
public:
    Student(int m) {
        marks = m;
    }
};
```

Object creation:

```
cpp

Student s1(90);
```

✔️ Multiple Constructors in a Class (Constructor Overloading)

More than one constructor can exist with different parameters.

```
cpp

class Test {
public:
    Test() {}
    Test(int x) {}
    Test(int x, int y) {}
};
```

This is called **constructor overloading**
→ compile-time polymorphism.

🟡 Dynamic Initialization of Objects

Values are assigned to objects **at runtime** using constructors.

```
cpp

class Sample {
    int a, b;
public:
    Sample(int x, int y) {
        a = x;
        b = y;
    }
};

int x=10, y=20;
Sample s(x,y);     // runtime initialization
```

Useful when:

✔️ values come from input
✔️ runtime calculations
✔️ file / DB / API input

🧩 Copy Constructor

A constructor that initializes an object **from another object of same class**.

```
cpp

class Demo {
public:
    int x;
    Demo(int a) { x = a; }

    Demo(const Demo &d) {
        x = d.x;
    }
};
```

Usage:

```
cpp

Demo d1(10);
Demo d2(d1);
```

✔️ When is Copy Constructor Called?

1️⃣ Object initialized from another object

```cpp
Demo d2(d1);
```

2️⃣ Object passed by value to function

3️⃣ Function returns object by value

✔️ Default vs User-defined Copy Constructor

If you don't define one —
compiler supplies **shallow copy constructor**.

User-defined copy constructor is required when class contains:

✔️ pointers
✔️ dynamic memory
✔️ resource handles

to avoid **shallow copy problem**.

⚠️ MCQ Trap

Passing object by value also invokes **copy constructor**.

🔴 Destructor

Destructor is a special function called when object is destroyed.

```cpp
class Demo {
public:
    ~Demo() {
        cout << "Destructor called";
    }
};
```

✔️ Properties of Destructor

- Same name as class prefixed with ~
- No parameters
- No return type
- Cannot be overloaded
- Automatically invoked when:

✔️ object goes out of scope
✔️ program ends

✔️ delete is used on dynamically allocated object

✔️ Primary Use

To release resources:

- memory
- files
- network connections

⚠️ Important Exam Traps

- Destructor executes in **reverse order of object creation**
- There is only **one destructor per class**
- Destructor **cannot be virtual?**
  → It *can* be virtual (and usually should be in base classes).
- Destructor of base class must be virtual when deleting derived object via base pointer.

🎯 High-Value MCQ Summary

- Constructor name = class name
- Constructors cannot return values
- Constructors can be overloaded
- Parameterized constructors take arguments
- Dynamic initialization happens at runtime
- Copy constructor copies object to object
- Passing object by value → invokes copy constructor
- Destructor frees resources
- Constructor → called at object creation
- Destructor → called at object destruction

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based constructor/destructor questions
👉 Deep copy vs shallow copy explanation
👉 Coding practice problems

Just tell me 👍

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 9 (Constructors & Destructor)** with answers and short explanations — designed in CDAC-CCEE style.

✅ MCQs — Constructors, Parameterized Constructors, Copy Constructor, Dynamic Initialization, Destructor

1. A constructor is executed when —

A) Class is defined
B) Object is created
C) Program starts
D) Function is called

Answer: B

✔️ Automatically invoked during object creation.

2. Which of the following is TRUE about constructors?

A) They must have a return type
B) They can be virtual only
C) They have same name as class
D) They must be static

Answer: C

3. A constructor that accepts arguments is called —

A) Default constructor
B) Parameterized constructor
C) Copy constructor
D) Inline constructor

Answer: B

4. Which of the following is NOT allowed in constructors?

A) Overloading
B) Default arguments
C) Return type
D) Parameters

Answer: C

✔️ Constructors do not return values.

---

5. What does constructor overloading represent?

A) Runtime polymorphism
B) Compile-time polymorphism
C) Dynamic binding
D) Encapsulation

Answer: B

---

6. Which constructor is invoked in this statement?

```cpp
Sample s1 = Sample(10);
```

A) Default
B) Copy
C) Parameterized
D) Implicit

Answer: C

---

7. Dynamic initialization of objects means —

A) Object created dynamically
B) Object initialized at runtime using constructor
C) Object stored on heap
D) Object created inside loop

Answer: B

---

8. How many constructors can a class have?

A) Only one
B) Only two
C) Many, with different parameter lists
D) Depends on compiler

Answer: C

✔️ Constructors can be overloaded.

---

9. Copy constructor initializes an object using —

A) Constant values
B) Pointer

C) Another object of same class
D) Reference variable

Answer: C

---

10. Which is the correct form of copy constructor?

A) Demo(Demo d)
B) Demo(const Demo &d)
C) Demo(Demo *d)
D) Demo(Demo &&d)

Answer: B

✔ Pass by reference to avoid infinite recursion.

---

11. Copy constructor is called when —

A) Object is destroyed
B) Object is assigned
C) Object is passed by value
D) Default constructor is called

Answer: C

✔ Passing objects by value copies them.

---

12. If no copy constructor is defined —

A) Program fails
B) Object cannot be copied
C) Compiler provides default copy constructor
D) Shallow copy is not possible

Answer: C

---

13. User-defined copy constructor is needed when class contains —

A) Primitive variables
B) Static members
C) Pointers / dynamic memory
D) Inline functions

Answer: C

✔ To prevent shallow copy issues.

---

14. Destructor is executed when —

A) Object is created
B) Object goes out of scope
C) Class is declared
D) Program compiles

Answer: B

---

15. Destructor name must be prefixed with —

A) ?
B) ::
C) ~
D) #

Answer: C

---

16. Which of the following is FALSE about destructors?

A) They cannot take arguments
B) They cannot return values
C) They can be overloaded
D) They are called automatically

Answer: C

✔️ Only one destructor is allowed per class.

---

17. Order of destructor execution in case of multiple objects is —

A) Same as creation
B) Random
C) Reverse of creation order
D) Depends on OS

Answer: C

---

18. How many destructors can a class have?

A) One
B) Two
C) Any number
D) Depends on compiler

Answer: A

---

19. Destructor is mainly used to —

A) Initialize variables
B) Allocate memory

C) Release resources
D) Call functions

Answer: C

---

20. Which constructor gets called when object is returned by value?

A) Default constructor
B) Parameterized constructor
C) Copy constructor
D) Destructor

Answer: C

🎯 Quick Revision Pointers

- Constructor name = class name
- No return type
- Constructors can be overloaded
- Parameterized constructors accept arguments
- Dynamic initialization happens at runtime
- Copy constructor copies object → object
- Called when passed/returned by value
- Destructor → `~ClassName()`
- Only one destructor per class
- Destructor called in reverse object order

If you want, I can also provide:

👉 Output-tracing constructor/destructor questions
👉 Deep copy vs shallow copy explanation
👉 Coding practice problems
👉 Next session topics

Just tell me 👍

Session 10: Inheritance – extending class
Lecture
 Types of inheritance
 Single inheritance
 Multiple inheritance
 Multilevel inheritance

 Hierarchical inheritance
 Hybrid inheritance, etc. Virtual base class Constructors in derived class

Sure — here's an **exam-oriented explanation of Session 10: Inheritance — Extending Classes**, written to help you answer CDAC-CCEE theory and MCQ questions with clarity.

✅ What is Inheritance?

Inheritance allows a **derived class** to acquire properties & behavior of a **base class**.

```cpp
class Base { };
class Derived : public Base { };
```

Benefits:

✔ Code reusability
✔ Extensibility
✔ Reduced redundancy
✔ Supports polymorphism

Private members of base class are **not inherited**, but can be accessed using public/protected member functions.

⭐ Access Modes in Inheritance (Very Important)

Access in Base	Public Derivation	Protected Derivation	Private Derivation
public	public	protected	private
protected	protected	protected	private
private	not inherited	not inherited	not inherited

✔ Access mode affects visibility — not storage.
✔ Default inheritance mode in class → **private**.

🟢 Types of Inheritance

✔ 1) Single Inheritance

One base → one derived class

```cpp
class A {};
class B : public A {};
```

✔️ 2) Multiple Inheritance

More than one base → one derived class

```cpp
class A {};
class B {};
class C : public A, public B {};
```

Useful but can cause ambiguity.

✔️ 3) Multilevel Inheritance

Base → Derived → Further derived

```cpp
class A {};
class B : public A {};
class C : public B {};
```

✔️ 4) Hierarchical Inheritance

One base → many derived

```cpp
class A {};
class B : public A {};
class C : public A {};
```

✔️ 5) Hybrid Inheritance

Combination of two or more inheritances

Example: Multiple + Multilevel

This may create **diamond problem**.

🔷 Virtual Base Class (Important Topic)

Occurs in multiple inheritance when two derived classes inherit from same base, and another class inherits from both.

This creates **duplicate copies** of base class.

```
cpp

class A { };
class B : public A { };
class C : public A { };
class D : public B, public C { };   // duplicate A
```

To avoid ambiguity → make base class **virtual**.

```
cpp

class B : virtual public A { };
class C : virtual public A { };
```

Now class `D` has only **one shared copy of A**.

👉 This solves the **Diamond Problem**.

🟡 Constructors in Derived Classes

Order of constructor execution:

1️⃣ Base class constructor
2️⃣ Derived class constructor

Example:

```
cpp

class Base {
public:
    Base() { cout << "Base"; }
};

class Derived : public Base {
public:
    Derived() { cout << "Derived"; }
};
```

Output:

```
nginx

BaseDerived
```

✔️ Parameterized Base Constructor in Inheritance

Use initialization list:

```cpp
class Base {
public:
    Base(int x) { }
};

class Derived : public Base {
public:
    Derived(int x) : Base(x) { }
};
```

🔴 Destructor Order

Destructors are executed in **reverse order**:

1️⃣ Derived class destructor
2️⃣ Base class destructor

⚠️ Important Exam Traps

- Private members are not inherited
- Default inheritance mode → **private**
- Constructor of base executes first
- Destructor of derived executes first
- Multiple inheritance may cause ambiguity
- Virtual base class avoids duplication
- Access mode affects visibility, not memory
- `virtual` in inheritance ≠ virtual functions

🎯 High-Value MCQ Concepts

- Single vs Multiple vs Multilevel
- Hierarchical vs Hybrid
- Diamond problem & virtual base class
- Constructor call order
- Destructor call order
- Access mode effect
- Ambiguity resolution using scope operator
- Derived class has one base copy when virtual base
- Public derivation retains base accessibility
- Protected derivation limits outside access

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based inheritance questions
👉 Constructor-order tracing questions
👉 Summary revision sheet

Just tell me 👍

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 10 (Inheritance — Extending Classes)** with answers and short explanations, aligned to CDAC-CCEE style.

✅ MCQs — Types of Inheritance, Virtual Base Class, Constructor Order

1. Inheritance allows —

A) Code duplication
B) Code reusability
C) Faster execution
D) Memory reduction

Answer: B

---

2. Which is the default access mode for inheritance in a C++ class?

A) public
B) private
C) protected
D) default

Answer: B

✔️ Default inheritance is **private**.

---

3. Which type of inheritance involves one base class and one derived class?

A) Multiple
B) Hierarchical
C) Single
D) Hybrid

Answer: C

---

4. Which inheritance has more than one base class and one derived class?

A) Multilevel
B) Multiple
C) Hybrid
D) Hierarchical

Answer: B

---

5. In multilevel inheritance —

A) Many base → one derived
B) One base → many derived
C) Derived class becomes base class of another
D) Only one class participates

Answer: C

---

6. One base class → many derived classes refers to —

A) Hierarchical inheritance
B) Multiple inheritance
C) Hybrid inheritance
D) Multilevel inheritance

Answer: A

---

7. Hybrid inheritance is —

A) Single + Multiple
B) Combination of two or more types
C) Multiple + Private
D) Multilevel only

Answer: B

---

8. The diamond problem occurs in —

A) Single inheritance
B) Multilevel inheritance
C) Hierarchical inheritance
D) Multiple inheritance

Answer: D

---

9. Virtual base class is used to —

A) Increase speed
B) Avoid duplicate base copies
C) Improve security
D) Support abstraction

Answer: B

✔️ Prevents ambiguity of base members.

---

10. Which keyword is used for virtual base class?

A) abstract
B) virtual
C) override
D) friend

Answer: B

---

11. Order of constructor execution in inheritance is —

A) Derived → Base
B) Base → Derived
C) Random
D) Depends on OS

Answer: B

---

12. Order of destructor execution is —

A) Base → Derived
B) Derived → Base
C) Random
D) Parallel

Answer: B

✔️ Reverse of constructor order.

---

13. Which access specifier does NOT change when using public inheritance?

A) private members
B) protected members
C) public members
D) static members

Answer: C

✔️ Public remains public.

---

14. Private members of base class are —

A) Not inherited
B) Inherited as private
C) Inherited as protected
D) Inherited as public

Answer: A

---

15. Which inheritance may cause ambiguity in member access?

A) Single
B) Multiple
C) Hierarchical
D) Multilevel

Answer: B

---

16. Which operator resolves ambiguity between base classes?

A) .
B) ->
C) ::
D) **

Answer: C

✔️ Scope resolution operator.

---

17. Which type of inheritance best supports code reusability across multiple derived classes?

A) Hybrid
B) Hierarchical
C) Single
D) Multilevel

Answer: B

---

18. In virtual base inheritance, how many copies of base class exist in derived class?

A) 0
B) 1
C) 2
D) Depends on compiler

Answer: B

---

19. Which statement is TRUE?

A) Access mode affects memory allocation
B) Access mode affects visibility only
C) Access mode changes object size
D) Access mode deletes members

Answer: B

---

20. Which constructor is executed first in inheritance?

A) Most derived class
B) Intermediate derived class
C) Base class
D) Random

Answer: C

🎯 Quick Revision Pointers

- Default inheritance → private
- Multiple inheritance → ambiguity
- Virtual base class → avoids duplication
- Constructor order → Base → Derived
- Destructor order → Derived → Base
- Private members → not inherited
- Public inheritance retains visibility
- Scope resolution resolves ambiguity

If you want, I can also share:

👉 Output-based inheritance questions
👉 Constructor-order tracing exercises
👉 Tough conceptual MCQs
👉 Next session explanation

Just tell me 👍

Session 11: Polymorphism
Lecture
 Types of Polymorphism
 Overloading functions Overloading Operators Friend functions
ACTS, Pune
PG-DAC Page 4 of 5
 Constant functions

Sure — here's an **exam-focused explanation of Session 11: Polymorphism** with concepts, rules, examples, and common CDAC-CCEE MCQ traps.

✅ What is Polymorphism?

"Poly" = many
"Morph" = forms

Polymorphism allows the **same function / operator / entity to behave differently** in different situations.

It improves:

✔ Flexibility
✔ Reusability
✔ Extensibility

⭐ Types of Polymorphism in C++

There are **two major categories**:

🟢 1) Compile-Time (Static / Early Binding)

Happens during compilation.

Includes:

✔ Function Overloading
✔ Operator Overloading
✔ Default Arguments
✔ Inline expansion

Execution is resolved before program runs.

🟣 2) Runtime (Dynamic / Late Binding)

Happens during execution.

Achieved using:

✔ Virtual functions
✔ Base class pointers / references

(Though runtime polymorphism is mainly covered in virtual functions topics — awareness is still expected here.)

🟡 Function Overloading

Same function name, different:

- number of parameters
- type of parameters
- order of parameters

```cpp
int add(int a, int b);
double add(double a, double b);
int add(int a, int b, int c);
```

❌ Return type alone cannot distinguish overloaded functions

This causes compilation error:

```cpp
int fun(int a);
float fun(int a);   // invalid
```

✔️ Overloading Resolution Happens at Compile-Time

Called **static binding**.

🧮 Operator Overloading

Allows operators to work with user-defined types.

```cpp
class Complex {
public:
    int r, i;

    Complex operator + (const Complex &c) {
        Complex temp;
        temp.r = r + c.r;
        temp.i = i + c.i;
        return temp;
    }
};
```

Usage:

```cpp
c3 = c1 + c2;
```

⚠️ Important Rules

✔️ Only existing operators can be overloaded
✔️ Cannot create new operators
✔️ Precedence & associativity remain same
✔️ Few operators cannot be overloaded:

`:: . .* ?: sizeof`

👫 Friend Functions

A non-member function that is allowed to access **private and protected members** of a class.

```cpp
class Test {
private:
    int x;
public:
    friend void show(Test t);
};
```

Definition:

```cpp
void show(Test t) {
    cout << t.x;
}
```

✔️ Key Points (Frequently Asked)

- Declared using keyword **`friend`**
- Not a member of class
- Accesses private data
- Invoked like a normal function
- Does NOT receive **`this`** pointer
- Does NOT support inheritance

📎 Friend Function & Operator Overloading

Used when LHS is not a class object.

```cpp
friend Complex operator+(Complex, Complex);
```

🟣 Constant Member Functions

Declared using **`const`** keyword after function header.

```cpp
class Demo {
public:
    int x;

    int get() const {
        // x++; ❌ not allowed
        return x;
    }
};
```

Meaning:

✔️ Function **cannot modify data members**
✔️ **`this`** pointer becomes **const** inside function

⚠️ Exceptions

- Can modify data marked as **`mutable`**
- Can call other const member functions only

🎯 High-Value MCQ Concepts

- Function overloading = compile-time polymorphism
- Runtime polymorphism = virtual functions
- Return type alone cannot overload function
- Operator overloading keeps precedence same
- **`:: , . , .* , ?: , sizeof`** cannot be overloaded
- Friend function is **not a member**
- Friend function has no **`this`** pointer
- Const function cannot modify object data
- **`mutable`** allows modification inside const function

If you want, I can provide:

👉 20 MCQs from this session
👉 Output-based polymorphism questions
👉 Operator overloading coding examples
👉 Short revision notes

Just tell me 👍

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 11 (Polymorphism, Function Overloading, Operator Overloading, Friend Functions, Const Functions)** with answers and brief explanations — designed in CDAC-CCEE style.

✅ MCQs — Polymorphism, Overloading, Friend Functions, Const Functions

1. Polymorphism allows —

A) Same function to behave differently
B) Same class to behave differently
C) Same object created many times
D) Same program compiled twice

Answer: A

2. Function overloading is an example of —

A) Runtime polymorphism
B) Compile-time polymorphism
C) Late binding
D) Dynamic binding

Answer: B

3. Which of the following is NOT valid for function overloading?

A) Different number of arguments
B) Different types of arguments
C) Different order of arguments
D) Different return type only

Answer: D

✔️ Return type alone cannot overload.

4. Operator overloading is used to —

A) Create new operators
B) Change precedence of operators
C) Redefine existing operator behavior
D) Improve execution speed

Answer: C

5. Which operator CANNOT be overloaded?

A) +
B) =
C) ::
D) ++

Answer: C

---

6. Operator overloading must preserve —

A) Syntax
B) Precedence and associativity
C) Memory usage
D) Data type only

Answer: B

---

7. Friend function is —

A) A member of class
B) A public function
C) A non-member function with access to private data
D) A static function

Answer: C

---

8. Which keyword is used to declare a friend function?

A) private
B) friend
C) global
D) external

Answer: B

---

9. Friend functions —

A) Receive `this` pointer
B) Do not receive `this` pointer
C) Are called using object
D) Must be declared public

Answer: B

---

10. Which of the following is TRUE about friend functions?

A) They support inheritance
B) They cannot access private members
C) They break encapsulation to some extent
D) They are always inline

Answer: C

---

11. A const member function means —

A) Function cannot be overloaded
B) Function cannot return values
C) Function cannot modify data members
D) Function executes only once

Answer: C

---

12. Inside a const function, `this` pointer is treated as —

A) static pointer
B) constant pointer to constant object
C) pointer to temporary object
D) null pointer

Answer: B

---

13. Which members can still be modified inside a const function?

A) static members
B) mutable members
C) private members
D) protected members

Answer: B

---

14. Which binding is used in function overloading?

A) Late binding
B) Runtime binding
C) Static binding
D) Dynamic binding

Answer: C

---

15. Which of the following allows different meaning to same operator for objects?

A) Function overloading
B) Operator overloading

C) Abstraction
D) Inheritance

Answer: B

---

16. Which declaration is correct for operator overloading using friend function?

A) friend + operator();
B) friend operator+(A, B);
C) friend A operator+(A, A);
D) friend A + operator(A);

Answer: C

---

17. In operator overloading, which function gets invoked?

A) Based on runtime input
B) Based on operand types
C) Based on return type
D) Always same

Answer: B

✔️ Decided at compile-time.

---

18. Which of the following is TRUE about friend functions?

A) They are inherited in derived class
B) They are called using object name
C) They break data hiding partially
D) They require virtual keyword

Answer: C

---

19. Which of the following is a disadvantage of friend functions?

A) Cannot access private data
B) Increase execution time
C) Reduce encapsulation
D) Cannot overload operator

Answer: C

---

20. A const function can call —

A) Only const member functions
B) Only non-const member functions
C) Any member function
D) Static functions only

Answer: A

🎯 Quick Revision Pointers

- Overloading → compile-time polymorphism
- Return type alone cannot overload
- Some operators cannot be overloaded (`:: . .* ?: sizeof`)
- Friend function → non-member with private access
- Friend has **no `this` pointer**
- Breaks encapsulation slightly
- Const member → cannot modify object
- **`mutable`** allows modification inside const function

If you want, I can also provide:

👉 Output-tracing polymorphism questions
👉 Coding practice for operator overloading
👉 Trick conceptual MCQs
👉 Next session explanation

Just tell me 👍

> Session 12: Virtual Functions and Abstract Class
> Lecture
> □ Run Time Polymorphism
> □ Virtual Functions and Pure virtual functions □ dynamic_cast, static_cast,const_cast, reinterpret_cast □ Interfaces □ Abstract class

Sure — here's an **exam-focused explanation of Session 12: Virtual Functions & Abstract Classes** with concepts, rules, examples, and common CDAC-CCEE MCQ traps.

I'll cover **runtime polymorphism, virtual functions, abstract classes, casts, and interfaces** clearly.

✅ Run-Time Polymorphism (Dynamic / Late Binding)

Binding of function call to function body happens **at runtime**.

Achieved using:

✔ Virtual functions
✔ Base class pointer / reference
✔ Function overriding

🟢 Example of Runtime Polymorphism

```cpp
class Base {
public:
    virtual void show() {
        cout << "Base";
    }
};

class Derived : public Base {
public:
    void show() {
        cout << "Derived";
    }
};

Base *ptr;
Derived obj;
ptr = &obj;
ptr->show();   // Output → Derived
```

✔️ Base pointer refers to derived object
✔️ Virtual function ensures **derived function executes**

Without **`virtual`** → Base version executes (static binding)

⭐ Virtual Functions

A virtual function allows function overriding to resolve **at runtime**.

Key Properties (MCQ favorites)

- Declared using **`virtual`**
- Supports runtime polymorphism
- Called via **base class pointer**
- Must be **non-static**
- Can be overridden in derived class
- Accessed dynamically via **vtable**

⚠️ If function is NOT virtual

Binding is compile-time
→ Base class version executes

🟣 Pure Virtual Functions (Abstract Functions)

A function with **no implementation** in base class.

```cpp
class Shape {
public:
    virtual void draw() = 0;   // Pure virtual
};
```

✔️ Forces derived classes to override
✔️ Makes class **abstract**

🟡 Abstract Class

A class containing **at least one pure virtual function**.

```cpp
class Shape {
public:
    virtual void draw() = 0;
};
```

Properties:

- Cannot create objects of abstract class
- Can contain data & normal functions
- Can provide partial implementation
- Derived class must override pure virtual functions

✔️ Object of abstract class? — ❌ NOT allowed

```cpp
Shape s;   // Error
```

✔️ Pointer to abstract class? — ✅ Allowed

```cpp
Shape *ptr;
```

Used for runtime polymorphism.

🟢 Interfaces in C++ (Conceptual Term)

C++ does NOT have separate **`interface`** keyword.

Interface is implemented as:

✔️ Class with **only pure virtual functions**

```cpp
class Printable {
public:
    virtual void print() = 0;
};
```

Equivalent to **Java interface behavior**.

🔁 Type Casting Operators in C++

C++ provides **four explicit cast operators**.

🟦 `static_cast` (Compile-time cast)

Used for:

✔️ numeric conversions
✔️ up-casting in inheritance
✔️ pointer to void and back

```cpp
int x = static_cast<int>(5.6);
```

Safe and preferred over C-style cast.

🟣 `dynamic_cast` (Runtime cast — RTTI)

Used for:

✔️ down-casting in inheritance
✔️ requires at least one virtual function

```cpp
Base *b = new Derived;
Derived *d = dynamic_cast<Derived*>(b);
```

If cast fails → returns **NULL** (for pointers)

👉 Works only with **polymorphic classes**

🟡 `const_cast` (Remove / add const)

```cpp
const int x = 10;
int *p = const_cast<int*>(&x);
```

Used to:

✔️ Modify constness
✔️ Pass const object to legacy APIs

⚠️ Undefined behavior if used to modify truly constant memory.

🟥 `reinterpret_cast` (Low-level / Unsafe)

Used for:

✔️ bitwise reinterpretation of memory

```
cpp

int *p = reinterpret_cast<int*>(0x1234);
```

Dangerous — used rarely in system-level programming.

🎯 Virtual Destructor (Important Concept)

If deleting derived object through base pointer →
Base destructor must be **virtual**.

```
cpp

class Base {
public:
    virtual ~Base() {}
};
```

Prevents **memory leaks**.

⚠️ High-Value MCQ Traps

- Virtual function → runtime polymorphism
- Without virtual → static binding
- Pure virtual function → `= 0`
- Abstract class cannot be instantiated
- Interface = class with only pure virtual functions
- **`dynamic_cast`** requires virtual function
- **`static_cast`** = compile-time cast
- **`const_cast`** removes const property
- **`reinterpret_cast`** is unsafe + low-level
- Virtual destructor recommended in base class

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based runtime polymorphism questions
👉 Virtual destructor & casting tricky problems
👉 Short revision notes

Just tell me 👍

> 20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 12 (Virtual Functions, Runtime Polymorphism, Abstract Class, Casting Operators, Interfaces)** with answers and short explanations — aligned to CDAC-CCEE style.

✅ MCQs — Virtual Functions, Pure Virtual, Abstract Class, Casting, Interfaces

1. Runtime polymorphism in C++ is achieved using —

A) Function overloading
B) Operator overloading
C) Virtual functions
D) Default arguments

Answer: C

---

2. Which binding is used in virtual functions?

A) Static binding
B) Early binding
C) Late binding
D) Compile-time binding

Answer: C

✔️ Late binding happens at runtime.

---

3. If a base class pointer points to derived object and function is NOT virtual, which function executes?

A) Base version
B) Derived version

C) Random
D) None

Answer: A

✔️ Static binding.

---

4. A pure virtual function is declared as —

A) virtual void show();
B) virtual void show() {}
C) virtual void show() = 0;
D) void show() = 0;

Answer: C

---

5. A class that contains at least one pure virtual function is called —

A) Interface
B) Abstract class
C) Base class
D) Static class

Answer: B

---

6. Which of the following is TRUE about abstract classes?

A) Objects can be created
B) Objects cannot be created
C) Objects must be created
D) Only friend functions allowed

Answer: B

---

7. A pointer of abstract class type —

A) Cannot be declared
B) Can be declared but not instantiated
C) Must refer to same class only
D) Causes runtime error

Answer: B

---

8. What must derived class do for pure virtual function?

A) Ignore it
B) Overload it
C) Override it
D) Delete it

Answer: C

---

9. A class with ONLY pure virtual functions acts as —

A) Template
B) Interface
C) Namespace
D) Virtual base

Answer: B

---

10. Which casting operator is used for safe down-casting at runtime?

A) static_cast
B) dynamic_cast
C) const_cast
D) reinterpret_cast

Answer: B

---

11. `dynamic_cast` works only when class contains —

A) Friend function
B) Virtual function
C) Overloaded operator
D) Static member

Answer: B

---

12. Which cast removes constness of a variable?

A) static_cast
B) reinterpret_cast
C) const_cast
D) dynamic_cast

Answer: C

---

13. Which cast performs compile-time conversions?

A) static_cast
B) dynamic_cast
C) reinterpret_cast
D) const_cast

Answer: A

---

14. Which cast is considered most unsafe?

A) static_cast
B) const_cast
C) reinterpret_cast
D) dynamic_cast

Answer: C

---

15. Why should base class destructor be declared virtual?

A) To improve performance
B) To prevent memory leaks when deleting derived object via base pointer
C) To support overloading
D) To allow inheritance

Answer: B

---

16. Which of the following statements is TRUE?

A) Virtual functions must be private
B) Virtual functions cannot be overridden
C) Virtual functions support runtime polymorphism
D) Virtual functions execute at compile time

Answer: C

---

17. What happens if derived class does NOT override a pure virtual function?

A) Program crashes
B) Derived class also becomes abstract
C) Function is ignored
D) Compiler adds default version

Answer: B

---

18. Which feature makes function resolution occur at runtime?

A) vtable & vptr mechanism
B) Stack frame allocation
C) Const qualifiers
D) Templates

Answer: A

---

19. Which statement about interfaces in C++ is correct?

A) Dedicated interface keyword exists
B) Implemented using abstract classes

C) Interface cannot contain pure virtual functions
D) Interface must contain data members

Answer: B

---

20. Destructor call order in inheritance is —

A) Base → Derived
B) Derived → Base
C) Random
D) Depends on compiler

Answer: B

✔️ Reverse of constructor order.

🎯 Quick Revision Pointers

- Virtual → runtime polymorphism
- Pure virtual → `= 0`
- Abstract class → cannot instantiate
- Interface → class with only pure virtual functions
- **`dynamic_cast`** → runtime, requires virtual function
- **`static_cast`** → compile-time cast
- **`const_cast`** → removes const
- **`reinterpret_cast`** → unsafe bit cast
- Base destructor should be virtual

If you'd like, I can also provide:

👉 Output-based polymorphism questions
👉 Coding scenarios on casting & abstract classes
👉 Tough MCQs from previous-year patterns

Session 13: Exception Handling
Lecture
 Exception Handling Introduction
 Exception handling – throwing, catching, re-throwing anexception
 Specifying exceptions etc.

Great — here's an **exam-oriented explanation of Session 13: Exception Handling in C++**, with concepts, rules, examples, and common CDAC-CCEE MCQ traps.

✅ What is Exception Handling?

Exception handling deals with **runtime errors** and prevents program termination.

It allows:

✔️ Error detection
✔️ Graceful recovery
✔️ Program stability

C++ uses:

- `try`
- `throw`
- `catch`

🟢 Basic Exception Handling Structure

```
cpp

try {
    // risky code
}
catch(type var) {
    // handler
}
```

If exception occurs → control jumps to matching `catch`.

✔️ Example

```
cpp

try {
    int a = 10, b = 0;
    if (b == 0)
        throw "Division by zero";
}
catch(const char *msg) {
    cout << msg;
}
```

🟡 `throw` — Raising an Exception

Used to signal an error.

```
cpp

throw value;
throw "Error";
throw obj;
```

Throws can be of any type:

✔️ built-in
✔️ class object
✔️ pointer
✔️ string

🟣 `catch` — Handling Exception

Catches and processes thrown value.

```
cpp

catch(int x) { }
catch(string s) { }
```

Catch order matters

More specific handlers should appear first.

✔️ Catch-All Handler

```
cpp

catch(...) {
    cout << "Unknown exception";
}
```

Catches **any type**.

🔁 Re-throwing an Exception

Used when exception is caught but passed further.

```
cpp

catch(...) {
    throw;   // re-throw
}
```

Useful in layered programs.

🟥 Exception Propagation

If no matching **`catch`** exists:

✔️ program terminates
✔️ **`terminate()`** function is called

🟦 Specifying Exceptions (Old `throw()` Specification)

Earlier C++ allowed:

```
cpp

void fun() throw(int);
```

Meaning → function may throw only **`int`**.

But in modern C++:

❌ dynamic exception specification is deprecated
✔️ replaced by **`noexcept`**

✔️ `noexcept`

```
cpp

void fun() noexcept;        // will not throw
void fun2() noexcept(false); // may throw
```

If function marked **`noexcept`** throws → program terminates.

⚠️ Common MCQ Traps

- Exception handling is for **runtime errors**, not compile-time
- **`try`** must be followed by at least one **`catch`**
- **`catch(...)`** → catch-all
- Re-throw uses **`throw;`**
- If exception not caught → program terminates
- Order of catch handlers is important
- Old **`throw()`** exception spec is deprecated
- **`noexcept`** indicates a function will not throw

🎯 Quick Concept Summary

- **`throw`** → generate exception
- **`catch`** → handle exception
- **`try`** → wrap risky code
- Matching handler chosen by **type**
- Can re-throw exception to outer block
- Catch-all handler catches everything

- `noexcept` replaces old throw-specification

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based exception questions
👉 Practical coding scenarios
👉 Short revision notes

Just tell me 👍

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 13 (Exception Handling)** with answers and brief explanations — aligned to CDAC-CCEE style.

✅ MCQs — Exception Handling, Throwing, Catching, Re-throwing, noexcept

---

1. Exception handling in C++ is mainly used to handle —

A) Syntax errors
B) Logical errors
C) Runtime errors
D) Linker errors

Answer: C

---

2. Which keywords are used in exception handling?

A) try, catch, throw
B) if, else, switch
C) new, delete, free
D) break, continue, goto

Answer: A

---

3. Which block must follow a try block?

A) finally
B) throw
C) catch
D) return

Answer: C

✔️ At least one `catch` is required.

---

4. Which statement is used to generate an exception?

A) error
B) throw
C) raise
D) signal

Answer: B

---

5. Which handler catches all types of exceptions?

A) catch(int)
B) catch(...)
C) catch(*)
D) catch(any)

Answer: B

---

6. What happens if no catch block matches a thrown exception?

A) It is ignored
B) Program continues
C) Program terminates
D) Exception auto-handled

Answer: C

✔️ `terminate()` is called.

---

7. Which of the following is TRUE about catch blocks?

A) Order does not matter
B) Generic handler must be first
C) Specific handlers must appear before catch(...)
D) Only one catch block allowed

Answer: C

---

8. Which of the following can be thrown as an exception?

A) int
B) string
C) object
D) All of the above

Answer: D

✔️ Any type may be thrown.

---

9. Re-throwing an exception is done using —

A) throw e;
B) rethrow();
C) throw;
D) resend();

Answer: C

---

10. Which of the following is FALSE?

A) try block can exist without catch
B) catch block cannot exist without try
C) try must have at least one catch
D) nested try blocks are allowed

Answer: A

✔️ A `try` without `catch` is invalid.

---

11. What is the purpose of exception propagation?

A) Pass exception to outer try block
B) Convert exception type
C) Stop program immediately
D) Ignore exception

Answer: A

---

12. What is the output of the following?

```cpp
try {
    throw 10;
}
catch(double x) {
    cout << "double";
}
catch(...) {
    cout << "default";
}
```

A) 10
B) double
C) default
D) error

Answer: C

✔️ Type did not match → caught by catch(...)

---

13. Dynamic exception specification like `void f() throw(int);` is —

A) Recommended
B) Optional
C) Deprecated in modern C++
D) Mandatory

Answer: C

---

14. Which keyword indicates that a function will not throw exceptions?

A) safe
B) noexcept
C) const
D) static

Answer: B

---

15. If a function marked `noexcept` throws an exception —

A) Exception is ignored
B) Control goes to catch
C) Program terminates
D) Compiler converts it to warning

Answer: C

---

16. Which of the following is TRUE about re-throwing exceptions?

A) It changes the exception value
B) It throws a new exception
C) It propagates the same exception
D) It converts exception into string

Answer: C

---

17. Which statement is valid for nested try blocks?

A) Inner try cannot throw
B) Only outer catch executes
C) Inner and outer catches may both handle
D) Not allowed in C++

Answer: C

---

18. Which type of errors cannot be handled using exception handling?

A) Division by zero
B) Array out of bounds
C) Syntax errors
D) Null pointer dereference

Answer: C

✔️ Compile-time errors.

---

19. What is the role of `throw;` inside a catch block?

A) Terminates program
B) Clears exception
C) Re-throws current exception
D) Creates new exception

Answer: C

---

20. Which of the following is TRUE?

A) Exceptions replace normal error handling
B) Exceptions must only be used for exceptional conditions
C) Every function should throw exceptions
D) All runtime errors are automatically caught

Answer: B

🎯 Quick Revision Pointers

- **`try–throw–catch`** → exception handling flow
- **`catch(...)`** → catch-all
- Order of catch blocks matters
- Uncaught exception → program terminates
- **`throw;`** → re-throw
- Exception propagation allowed
- Old **`throw()`** spec deprecated
- **`noexcept`** means function will not throw

If you'd like, I can also provide:

👉 Output-based exception questions
👉 Coding practice problems
👉 Tough conceptual MCQs
👉 Next session explanation

Session 14: Managing Console I/O Operations
Lecture
 Introduction
 C++ streams C++ stream classes Unformatted I/O operations Formatted I/O operations
Managing output with manipulators

Sure — here's an **exam-focused explanation of Session 14: Managing Console I/O Operations in C++**, with concepts, rules, examples, and MCQ-relevant points.

✅ Introduction to C++ I/O System

C++ uses **streams** to perform input and output.

A stream is a **flow of data** between program & device.

✔ Input stream → data flows **into** program
✔ Output stream → data flows **out of** program

Unlike C (scanf/printf), C++ uses

- **`cin`** → input
- **`cout`** → output
- **`cerr / clog`** → error streams

🟢 C++ Streams

C++ I/O system is defined in:

```cpp
#include <iostream>
```

Important standard streams:

Stream	Purpose
cin	standard input (keyboard)
cout	standard output (screen)
cerr	unbuffered error output
clog	buffered error/log output

🟣 C++ Stream Classes (Hierarchy — MCQ topic)

All I/O classes are defined in **`<iostream>`** & **`<fstream>`**.

Key base classes:

Class	Purpose
ios	base class
istream	input stream
ostream	output stream
iostream	both input & output

Relationships:

- **`cin`** is object of **`istream`**
- **`cout`**, **`cerr`**, **`clog`** are objects of **`ostream`**
- File streams derive from **`fstream`**, **`ifstream`**, **`ofstream`**

🟡 Unformatted I/O Operations

These input/output operations work on **characters / raw data**.

✔️ Character Input

```cpp
char ch;
cin.get(ch);
```

Reads even whitespace & newline.

✔️ Character Output

```cpp
cout.put(ch);
```

Outputs a single character.

✔️ Line Input

```cpp
cin.getline(str, size);
```

Reads spaces until newline.

⚠️ MCQ Tip

`cin >>` skips whitespace,
`get()` & `getline()` do not skip.

🔵 Formatted I/O Operations

Uses extraction (>>) and insertion (<<) operators.

```cpp
int a;
cin >> a;
cout << "Value = " << a;
```

Formatting controls include:

✔️ field width
✔️ alignment
✔️ precision
✔️ base (hex, oct, dec)

🟠 Managing Output with Manipulators

Manipulators modify output display style.

Available in:

```cpp
#include <iomanip>
```

⭐ Common Manipulators (Exam-favourites)

Manipulator	Meaning
`setw(n)`	set field width
`setfill(ch)`	fill empty space
`setprecision(n)`	set decimal precision
`fixed`	fixed-point notation
`scientific`	scientific notation
`showpoint`	force decimal point
`hex` / `oct` / `dec`	change number base
`left` / `right`	alignment

✔️ Example

```cpp
cout << setw(5) << setfill('*') << 25;
```

Output:

```
***25
```

✔️ Precision Example

```cpp
cout << fixed << setprecision(2) << 5.6789;
```

Output:

```
5.68
```

✔️ Hex / Oct Output

```cpp
cout << hex << 255;   // ff
cout << oct << 255;   // 377
```

🔻 Difference Between cerr & clog

Stream	Buffering	Usage
cerr	unbuffered	immediate error output
clog	buffered	program logs

🎯 Important MCQ Traps

- Streams are objects — not keywords
- **`cout`** belongs to **`ostream`**
- **`cin.get()`** reads whitespaces
- **`setw()`** applies to **next output only**

- `setprecision()` affects floating output
- `cerr` is unbuffered
- `clog` is buffered
- `ios` is base class for all streams

📝 Quick Revision Summary

- I/O in C++ is stream-based
- `cin`, `cout`, `cerr`, `clog` are predefined objects
- Unformatted I/O → `get()`, `put()`, `getline()`
- Formatted I/O → insertion/extraction operators
- Manipulators → formatting control (`setw`, `setprecision`, etc.)

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based formatting questions
👉 Practical console I/O examples
👉 Short revision notes

Just tell me 👍

> 20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 14 (Managing Console I/O Operations)** with answers and short explanations — aligned to CDAC-CCEE style.

✅ MCQs — C++ Streams, Stream Classes, Unformatted & Formatted I/O, Manipulators

1. In C++, input and output operations are performed using —

A) Functions
B) Streams
C) Macros
D) Templates

Answer: B

---

2. Which header file defines standard I/O streams?

A) stdio.h
B) iostream
C) iomanip
D) fstream

Answer: B

---

3. `cout` is an object of which class?

A) istream
B) ostream
C) iostream
D) ios

Answer: B

---

4. Which stream is used for standard input?

A) cout
B) clog
C) cerr
D) cin

Answer: D

---

5. Which stream is unbuffered and used for error messages?

A) cout
B) cerr
C) clog
D) cin

Answer: B

---

6. Which stream is buffered and used for logging?

A) cerr
B) clog
C) cout
D) ios

Answer: B

---

7. Which function reads a single character including whitespace?

A) cin >> ch
B) cin.get(ch)

C) getline(ch)
D) put()

Answer: B

---

8. Which function outputs a single character?

A) putchar()
B) cout.put(ch)
C) write()
D) send()

Answer: B

---

9. Which function reads a line of text including spaces?

A) cin >> str
B) gets(str)
C) cin.getline(str, n)
D) read(str)

Answer: C

---

10. The extraction operator in C++ is —

A) <<
B) >>
C) ::
D) ->

Answer: B

---

11. The insertion operator in C++ is —

A) ::
B) >>
C) <<
D) ::

Answer: C

---

12. `setw(n)` manipulator is used to —

A) set precision
B) set field width
C) set fill character
D) align text

Answer: B

13. Which header file contains manipulators?

A) iostream
B) cmath
C) iomanip
D) stdio

Answer: C

---

14. `setprecision(n)` affects —

A) integer values only
B) next output only
C) floating-point output
D) string output

Answer: C

---

15. Which manipulator forces decimal point display?

A) show
B) showfloat
C) showpoint
D) displaypoint

Answer: C

---

16. Which manipulator prints numbers in hexadecimal form?

A) hex
B) base16
C) hprint
D) sethex

Answer: A

---

17. Which manipulator fills unused field space with a character?

A) setw()
B) setfill()
C) fill()
D) pad()

Answer: B

---

18. `setw()` applies to —

A) all subsequent outputs
B) current stream only
C) next output item only
D) numeric values only

Answer: C

---

19. `cin >>` operator —

A) reads whitespace
B) skips whitespace
C) reads only numbers
D) reads only characters

Answer: B

---

20. Which of the following is TRUE?

A) ios is the base class of all stream classes
B) ostream derives from iostream
C) cin belongs to ostream
D) clog belongs to istream

Answer: A

🎯 Quick Revision Pointers

- Streams perform I/O in C++
- **`cout`** → ostream, **`cin`** → istream
- **`cerr`** unbuffered | **`clog`** buffered
- Unformatted I/O → get(), put(), getline()
- Formatted I/O → << and >> operators
- Manipulators → setw, setfill, setprecision
- **`setw()`** affects next output only

If you'd like, I can also provide:

👉 Output-based formatting questions
👉 Practice programs on I/O manipulators
👉 Short handwritten-style revision notes

Session 15: File Handling in C++
Lecture
 Definition of file

 File handling in C++
 Doing read, write operation in files

Sure — here's an **exam-focused explanation of Session 15: File Handling in C++**, with concepts, rules, examples, and MCQ-relevant points — aligned to CDAC-CCEE style.

✅ What is a File?

A file is a **named collection of data** stored permanently on secondary storage (disk).

Unlike variables (temporary, in RAM):

✔️ Data in files is **persistent**
✔️ Can be read/written across program runs

🟢 File Handling in C++

C++ provides file handling through **streams**, defined in:

```
cpp

#include <fstream>
```

✔️ File Stream Classes

Class	Purpose
`ifstream`	input (read from file)
`ofstream`	output (write to file)
`fstream`	both input & output

These classes are derived from **`istream`** / **`ostream`**.

🟡 Opening a File

Two ways to open files.

1️⃣ Using Constructor

```
cpp

ofstream fout("data.txt");
ifstream fin("data.txt");
```

2️⃣ Using `open()` Function

```cpp
ofstream fout;
fout.open("data.txt");
```

🔴 File Opening Modes (MCQ Important)

Common modes:

Mode	Meaning
ios::in	open for reading
ios::out	open for writing
ios::app	append to end
ios::ate	open & seek to end
ios::trunc	delete old contents
ios::binary	binary mode

Modes may be combined using `|`.

Example:

```cpp
fstream f("data.txt", ios::in | ios::out);
```

🟣 Writing to a File

```cpp
ofstream fout("data.txt");
fout << "Hello File";
fout.close();
```

Uses insertion operator (<<).

🔵 Reading from a File

```cpp
ifstream fin("data.txt");
string s;
fin >> s;
```

Reads word by word (skips whitespace).

✔️ Reading Full Line

```cpp
getline(fin, s);
```

🟤 Checking File Open Status

```cpp
if (!fin) {
    cout << "File not found";
}
```

or

```cpp
if (fin.fail())
```

🟠 End of File (EOF)

Loop until file ends:

```cpp
while (!fin.eof()) {
    fin >> s;
}
```

(Better approach → read in loop condition)

🟢 Closing File

```cpp
fin.close();
fout.close();
```

✔️ flushes buffer
✔️ releases resources

🗒️ Binary File Operations (Concept Level)

```cpp
fout.write((char*)&obj, sizeof(obj));
fin.read((char*)&obj, sizeof(obj));
```

Used for:

✔ structures
✔ class objects
✔ raw bytes

⚠ Important MCQ Traps

- **`ofstream`** truncates file by default (**`ios::trunc`**)
- **`ios::app`** always writes at end
- **`ios::ate`** positions pointer at end but allows seek
- **`ifstream`** → read only
- **`ofstream`** → write only
- **`fstream`** → read + write
- Must close file to flush buffer
- **`getline()`** reads spaces, `>>` does not
- File handling uses **`<fstream>`**

📝 Quick Revision Summary

- Files store persistent data
- **`ifstream`**, **`ofstream`**, **`fstream`** for I/O
- Files may be opened via constructor / open()
- Open modes control behavior
- Read → `>>` or **`getline()`**
- Write → `<<`
- Close after use

If you want, I can also provide:

👉 20 MCQs from this session
👉 Practical coding questions on file handling
👉 Output-based file reading problems
👉 Short revision notes

Just tell me 👍

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 15 (File Handling in C++)** with answers and brief explanations — aligned to CDAC-CCEE style.

✅ MCQs — Files, Streams, Read/Write, File Modes, EOF

1. A file is best described as —

A) Temporary collection of data
B) Permanent collection of data on secondary storage
C) Random memory block
D) Compiler output

Answer: B

2. Which header file is required for file handling in C++?

A) iostream
B) stdio.h
C) fstream
D) iomanip

Answer: C

3. Which class is used for reading from a file?

A) ofstream
B) ifstream
C) fstream
D) iostream

Answer: B

4. Which class is used for writing to a file?

A) fstream
B) ostream
C) ofstream
D) ifstream

Answer: C

5. Which class supports both read and write operations?

A) iostream
B) fstream
C) filebuf
D) ios

Answer: B

---

6. Which file opening mode is used for reading?

A) ios::out
B) ios::app
C) ios::in
D) ios::binary

Answer: C

---

7. Which mode truncates an existing file when opened with ofstream?

A) ios::app
B) ios::trunc
C) ios::ate
D) ios::in

Answer: B

---

8. Which mode always writes data at end of file?

A) ios::trunc
B) ios::binary
C) ios::app
D) ios::ate

Answer: C

---

9. Which mode opens file and positions pointer at end but allows seeking?

A) ios::app
B) ios::ate
C) ios::in
D) ios::binary

Answer: B

---

10. Which statement writes data to file?

A) fin >> data
B) fout << data

C) write(data)
D) send(data)

Answer: B

---

11. The extraction operator >> when used with file input —

A) reads entire line
B) reads including spaces
C) skips spaces and newlines
D) reads binary data

Answer: C

---

12. Which function reads a full line from file (including spaces)?

A) fin >> str
B) getline(fin, str)
C) fin.get(str)
D) read(str)

Answer: B

---

13. Which function is used to write binary data to file?

A) write()
B) put()
C) send()
D) append()

Answer: A

---

14. Which function is used to read binary data from file?

A) fetch()
B) get()
C) read()
D) scan()

Answer: C

---

15. Which statement checks whether file opened successfully?

A) if (!fin)
B) if (fin == 0)
C) if (fin.open)
D) if (fin.file)

Answer: A

16. The EOF condition is used to detect —

A) beginning of file
B) empty file
C) end of file
D) file not found

Answer: C

---

17. Which is the best way to read until EOF?

A) while(!fin.eof())
B) while(fin)
C) while(fin >> str)
D) while(true)

Answer: C

✔ Reads and checks success in condition.

---

18. Which of the following is TRUE?

A) ifstream is for writing
B) ofstream is for reading
C) fstream supports both reading and writing
D) ios::app deletes existing contents

Answer: C

---

19. Which pointer is modified when reading a file?

A) put pointer
B) get pointer
C) file pointer
D) data pointer

Answer: B

---

20. Which statement is recommended after file operations?

A) stop()
B) quit()
C) close()
D) remove()

Answer: C

✔ Releases resources & flushes buffer.

🎯 Quick Revision Pointers

- **`fstream`** → read + write
- **`ifstream`** → read
- **`ofstream`** → write
- **`ios::trunc`** clears file
- **`ios::app`** appends
- **`ios::ate`** opens at end but allows seek
- **`getline()`** reads full line
- Close files after use

If you'd like, I can also provide:

👉 Coding exercises on file handling
👉 Output-based file read/write questions
👉 Short handwritten-style notes

Session 16: Templates
Lecture
 Introduction to Templates Function Templates Class Templates

Sure — here's an **exam-focused explanation of Session 16: Templates in C++**, with concepts, rules, examples, and common CDAC-CCEE MCQ traps.

✅ What are Templates?

Templates support **generic programming** — writing code that works with **multiple data types** without duplication.

Benefits:

✔️ Reusability
✔️ Type safety
✔️ Compile-time type resolution
✔️ Better performance than void pointers

Two main kinds:

1️⃣ Function Templates
2️⃣ Class Templates

Defined using the **`template`** keyword.

🟢 Function Templates

Used when **same logic** works for multiple data types.

```cpp
template <class T>
T add(T a, T b) {
    return a + b;
}
```

Call:

```cpp
cout << add(2, 3);        // int
cout << add(2.5, 3.7);    // double
```

✔️ `class` and `typename` are interchangeable

```cpp
template <typename T>
```

Both are valid.

⚠️ MCQ Trap

Template function is instantiated only when **called** with a type.

🟡 Function Template with Multiple Parameters

```cpp
template <class T1, class T2>
void show(T1 a, T2 b) {
    cout << a << " " << b;
}
```

🟣 Class Templates

Used to create **generic classes**.

```cpp
template <class T>
class Box {
    T value;
public:
    void set(T v) { value = v; }
    T get() { return value; }
};
```

Creating objects:

```cpp
Box<int> b1;
Box<double> b2;
```

✔️ Class Template with Multiple Types

```cpp
template <class T1, class T2>
class Pair {
public:
    T1 x;
    T2 y;
};
```

🔵 Member Function Definition Outside Class (Template Class)

Must repeat template header.

```cpp
template <class T>
T Box<T>::get() {
    return value;
}
```

If template header is missing → compilation error.

🟠 Template Specialization (Concept Level)

Allows custom implementation for a specific type.

```cpp
template <>
class Box<char> {
    // specialized behavior
};
```

🎯 Important MCQ Concepts

- Templates enable **generic programming**
- **`class`** and **`typename`** are same in template parameter list
- Template is instantiated when used
- Function overloading vs function template can coexist

- Class templates must include template header in external definitions
- Template specialization overrides generic version
- Template code is generated at **compile-time**

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based template questions
👉 Coding exercises on function & class templates
👉 Short revision notes

20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Session 16 (Templates in C++)** with answers and brief explanations — aligned to CDAC-CCEE style.

✅ MCQs — Templates, Function Templates, Class Templates, Generic Programming

---

1. Templates in C++ are used to implement —

A) Structured programming
B) Generic programming
C) Procedural programming
D) Object-oriented programming only

Answer: B

---

2. Which keyword is used to define a template?

A) generic
B) define
C) template
D) param

Answer: C

---

3. Function template works for —

A) one specific data type only
B) multiple data types
C) integer only
D) class type only

Answer: B

---

4. Which of the following is valid template header?

A) template <class T>
B) template (class T)
C) template {T}
D) template<T>

Answer: A

---

5. `class` and `typename` keywords in template parameter list are —

A) not related
B) interchangeable
C) opposites
D) deprecated

Answer: B

---

6. Template instantiation happens —

A) at runtime
B) when template is declared
C) when template is called with a type
D) during linking

Answer: C

---

7. Which of the following defines a function template?

```cpp

? <class T>
T add(T a, T b);
```

A) template
B) generic
C) typedef
D) macro

Answer: A

---

8. What is the advantage of templates?

A) Increase runtime
B) Avoid code duplication

C) Reduce type safety
D) Reduce compilation speed

Answer: B

---

9. Which of the following creates an object of a class template?

```cpp
template<class T> class Box {};
Box<int> b;
```

The template parameter is —

A) int
B) Box
C) b
D) T

Answer: A

---

10. Which statement is TRUE about class templates?

A) Only one type parameter allowed
B) Can have multiple type parameters
C) Cannot have functions
D) Cannot have data members

Answer: B

---

11. When defining a template class member function outside class, we must —

A) omit template header
B) repeat template header
C) use typedef
D) use namespace

Answer: B

---

12. Which of the following is valid multiple-parameter function template?

A) template <T1, T2>
B) template <class T1, class T2>
C) template <typename>
D) template <T>

Answer: B

---

13. Templates improve performance because —

A) they run in parallel
B) type resolution happens at compile-time
C) they reduce code size
D) they use pointers

Answer: B

---

14. Function template and overloaded function can —

A) never coexist
B) coexist together
C) cause runtime error
D) override each other

Answer: B

---

15. What is template specialization?

A) Removing templates
B) Providing custom implementation for specific type
C) Converting template to macro
D) Preventing instantiation

Answer: B

---

16. Which of the following is TRUE about class templates?

A) Objects must be created without type
B) Type must be specified when creating object
C) Compiler automatically detects type
D) Only integers allowed

Answer: B

---

17. Which of the following is NOT a benefit of templates?

A) Code reusability
B) Type safety
C) Generic design
D) Runtime overhead

Answer: D

✔️ Templates avoid runtime overhead.

---

18. Which template parameter keyword is preferred in modern C++?

A) class only
B) typename only

C) either class or typename
D) none

Answer: C

---

19. Template code is generated —

A) dynamically at runtime
B) during linking
C) at compile-time
D) by interpreter

Answer: C

---

20. Which of the following is TRUE?

A) Templates are resolved at runtime
B) Templates eliminate need for function overloading
C) Template functions are created for each used type
D) Templates store single universal object

Answer: C

🎯 Quick Revision Pointers

- Templates → generic programming
- `template <class T>` or `template <typename T>`
- Instantiation happens when used
- Class templates require type parameter in object creation
- Member functions outside class → repeat template header
- Specialization provides custom type behavior
- Templates compile per-type → compile-time resolution

If you'd like, I can also provide:

👉 Output-based template tracing questions
👉 Coding exercises on templates
👉 Short revision notes

Sessions 17 & 18: STL and RTTI
Lecture
 Introduction to C++ Standard Library Working with Stack, Vector, Queue, Map
 Introduction to RTTI (Run-Time Type Information) in C++

Sure — here's an **exam-focused explanation of Sessions 17 & 18: STL and RTTI**, written to match CDAC-CCEE style and MCQ expectations.

✅ C++ Standard Library (STL) — Introduction

STL provides **ready-made generic data structures & algorithms** implemented using **templates**.

Major STL Components:

1️⃣ Containers — store data
2️⃣ Iterators — access container elements
3️⃣ Algorithms — operate on containers (sort, find, etc.)

STL improves:

✔️ reusability
✔️ performance
✔️ reliability
✔️ development speed

Header files come from **<vector>**, **<stack>**, **<queue>**, **<map>**, etc.

🟢 Vector (Dynamic Array)

Defined in:

```
cpp

#include <vector>
```

Features:

✔️ Dynamic size
✔️ Random access
✔️ Fast insertion at end
✔️ Slower insertion in middle

Example:

```
cpp

vector<int> v = {1,2,3};
v.push_back(4);
v[0] = 10;
```

Useful functions:

- **push_back()**
- **pop_back()**
- **size()**
- **front()**, **back()**

- `at()` (bounds-checked)

🟣 Stack (LIFO — Last In First Out)

Header:

```
cpp

#include <stack>
```

Operations:

- `push(x)`
- `pop()`
- `top()`
- `empty()`
- `size()`

Example:

```
cpp

stack<int> s;
s.push(10);
s.push(20);
s.pop();
```

No direct iteration — must pop elements to traverse.

🟡 Queue (FIFO — First In First Out)

Header:

```
cpp

#include <queue>
```

Operations:

- `push(x)` → insert at rear
- `pop()` → remove front
- `front()`
- `back()`
- `empty()`
- `size()`

Example:

```cpp
queue<int> q;
q.push(1);
q.push(2);
q.pop();
```

🟠 Map (Associative Container — Key/Value)

Header:

```cpp
#include <map>
```

Stores **sorted key-value pairs** (by default).

Keys are **unique**.

Example:

```cpp
map<int,string> m;
m[1] = "A";
m[2] = "B";
```

Functions:

- **`insert()`**
- **`erase()`**
- **`find()`**
- **`count()`**
- **`size()`**

Traversal:

```cpp
for(auto &p : m)
    cout << p.first << " " << p.second;
```

Internally implemented using **balanced binary tree (RB-Tree)**.

🧩 Iterators (Concept Level)

Act like **pointers** to container elements.

Examples:

```cpp
vector<int>::iterator it;
for(it = v.begin(); it != v.end(); ++it)
    cout << *it;
```

⭐ RTTI (Run-Time Type Information)

RTTI allows program to know **object's type at runtime**.

It supports:

✔ **`typeid`** operator
✔ **`dynamic_cast`**

Works only with **polymorphic types** (i.e., classes having at least one virtual function).

🟦 `typeid` Operator

Returns type information.

```cpp
cout << typeid(x).name();
```

Used to:

- compare object types at runtime
- debug / logging / safe casting

🟣 `dynamic_cast`

Used for **safe down-casting**.

```cpp
Base *b = new Derived;
Derived *d = dynamic_cast<Derived*>(b);
```

If cast fails:

- pointer → **`NULL`**
- reference → throws **`bad_cast`**

Requires **virtual function in base class**.

⚠️ Common MCQ Traps

- Vector allows random access, stack/queue do not
- Stack → LIFO, Queue → FIFO
- Map stores **sorted** keys (unordered_map does not)
- Map keys are unique
- `at()` is bounds-checked, `[]` is not
- `pop()` does not return value (stack/queue)
- RTTI works only with polymorphic classes
- `typeid` returns type information
- `dynamic_cast` is safe for down-casting

If you want, I can also provide:

👉 20 MCQs from this session
👉 Output-based STL questions
👉 Coding practice problems on containers & RTTI
👉 Short revision notes

> 20 MCQs from this session

Sure — here are **20 exam-focused MCQs from Sessions 17 & 18 (STL and RTTI)** with answers and brief explanations — aligned to CDAC-CCEE style.

✅ MCQs — STL Containers, Vector, Stack, Queue, Map, Iterators, RTTI

---

1. STL mainly provides support for —

A) Memory management
B) Generic containers & algorithms
C) Compiler optimization
D) Exception handling only

Answer: B

---

2. Which STL container implements a dynamic array?

A) list
B) deque
C) vector
D) array

Answer: C

---

3. Which operation adds an element at end of vector?

A) push()
B) append()
C) push_back()
D) add()

Answer: C

---

4. Which function gives number of elements in vector?

A) length()
B) count()
C) size()
D) total()

Answer: C

---

5. `v.at(i)` differs from `v[i]` because —

A) at() is faster
B) at() checks bounds
C) at() works only on strings
D) at() modifies vector

Answer: B

---

6. Stack follows which principle?

A) FIFO
B) LILO
C) FILO / LIFO
D) Random

Answer: C

---

7. Which operation returns top element of stack?

A) front()
B) peek()
C) top()
D) head()

Answer: C

---

8. Which of the following is TRUE about stack pop()?

A) Removes and returns element
B) Removes element only
C) Returns element only
D) Deletes entire stack

Answer: B

✔ pop() does **not** return value.

---

9. Queue follows which principle?

A) LIFO
B) FIFO
C) FILO
D) Random

Answer: B

---

10. In queue, insertion takes place at —

A) front
B) middle
C) rear
D) anywhere

Answer: C

---

11. Which function returns front element of queue?

A) top()
B) head()
C) front()
D) first()

Answer: C

---

12. Which STL container stores key–value pairs in sorted order?

A) unordered_map
B) set
C) map
D) multimap

Answer: C

---

13. Keys in map are —

A) duplicate allowed
B) always unique

C) always integers
D) inserted in FIFO order

Answer: B

---

14. Internal implementation of map is based on —

A) Hash table
B) Binary search tree (RB-Tree)
C) Linked list
D) Array

Answer: B

---

15. Which of the following is TRUE about iterators?

A) They are objects behaving like pointers
B) They store elements
C) They are keywords
D) They are macros

Answer: A

---

16. Which header must be included for vector?

A) <list>
B) <vector>
C) <stack>
D) <queue>

Answer: B

---

17. RTTI stands for —

A) Run-Time Type Identification
B) Real-Time Type Interface
C) Random Type Inference
D) Runtime Template Implementation

Answer: A

---

18. Which operator is used to get type information at runtime?

A) typeof
B) instanceOf
C) typeid
D) classof

Answer: C

19. `dynamic_cast` is mainly used for —

A) Up-casting
B) Compile-time conversion
C) Safe down-casting
D) Pointer arithmetic

Answer: C

---

20. `dynamic_cast` works only when —

A) class has virtual function
B) class is template
C) class is static
D) class is abstract

Answer: A

✔️ RTTI requires polymorphic type.

🎯 Quick Revision Pointers

- Vector → dynamic array, random access
- Stack → LIFO | Queue → FIFO
- pop() removes but does not return
- Map → sorted unique keys, RB-tree
- Iterators act like pointers
- RTTI → typeid + dynamic_cast
- dynamic_cast → safe runtime down-casting

If you'd like, I can also provide:

👉 Output-based STL questions
👉 Coding exercises on containers & RTTI
👉 Short revision notes