



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

C++ Programming Study Book

The complete study notes, practice questions and glossary for C++ Programming

INSIDE THIS BOOK

- C++ Basics: Input, Output and References
- Classes, Objects and Access Specifiers
- Constructors and Destructors
- Inheritance
- Polymorphism and Virtual Functions
- ...and 2 more chapters, practice questions and a glossary

Beginner to intermediate

Students who know some C and are moving to C++

engineergurukul.com

+91 97116 99511

C++ Programming Study Book

Move from C to object-oriented C++17: classes, constructors, inheritance, virtual functions, overloading, templates, the STL containers vector, string and map, and exception handling.

WHO IT IS FOR	Students who know some C and are moving to C++
LEVEL	Beginner to intermediate
SUBJECT AREA	Programming and web
WHAT IS INSIDE	7 chapters, practice questions with answers, and a glossary

CONTENTS

1	C++ Basics: Input, Output and References	3
2	Classes, Objects and Access Specifiers	5
3	Constructors and Destructors	8
4	Inheritance	10
5	Polymorphism and Virtual Functions	13
6	Function Overloading, Operator Overloading and Templates	16
7	STL Containers and Exceptions	19
	Practice questions	22
	Answers	26
	Glossary	28

Free ebook: you may read, print and share it for learning; it may not be sold or changed. © 2026 Gurukul Technicals Institute. First edition, October 2026. Contact details are on the back cover.

C++ Basics: Input, Output and References

From C to C++

C++ grew out of C and keeps almost all of its syntax, so variables, `if`, loops and functions work the same way. On top of that it adds classes, references, templates, exceptions and a large standard library. Source files usually end in `.cpp`, and with GCC you compile one as C++17 like this:

TEXT

```
g++ -std=c++17 billing.cpp -o billing
```

Input and output streams

CPP

```
#include <iostream>
#include <string>

int main()
{
    std::string name;
    int age;

    std::cout << "Enter your first name and age: ";
    std::cin >> name >> age;
    std::cout << "Hello, " << name << "! Next year you will be " << age + 1 << ".\n";
    return 0;
}
```

- `std::cout` is the standard output stream, and `<<`, the insertion operator, sends values to it. You can chain several values in one statement.
- `std::cin` is the standard input stream, and `>>`, the extraction operator, reads into variables. It stops at whitespace, so if the user types "Ravi Kumar", only `Ravi` goes into the string. Use `std::getline(std::cin, name)` to read a whole line.
- `std::string` is a real string type: it grows as needed and can be joined with `+`.
- The `std::` prefix means these names belong to the standard namespace. Small programs often write `using namespace std;` once instead, but avoid it in header files, where it can cause name clashes.

C++ also has a `bool` type with the values `true` and `false`, and since C++11 the keyword `auto` lets the compiler work out a variable's type from its initial value.

References

A reference is another name, an alias, for an existing variable. You declare it with `&` after the type:

CPP FRAGMENT

```
int price = 500;
int& ref = price;           // ref is another name for price
ref = 650;                  // changes price
std::cout << price << '\n'; // 650
```

Three rules apply:

1. A reference must be initialised when it is declared.
2. Once bound, it always refers to the same variable; it cannot be made to refer to another one.
3. There is no such thing as a null reference.

Passing by reference

By default C++ passes arguments by value, just like C. A reference parameter lets a function change the caller's variable without any pointer syntax:

CPP

```
#include <iostream>

void addBonus(int& salary, int bonus)
{
    salary += bonus;
}

int main()
{
    int salary = 25000;
    addBonus(salary, 3000);
    std::cout << salary << '\n'; // 28000
    return 0;
}
```

For a large object that a function only reads, pass a `const` reference, as in `void printBill(const std::string& text)`. This avoids copying the object, and the compiler stops the function from modifying it.

Choose by intent: pass small types by value, use a reference when the function must change the caller's variable, and use a `const` reference to read a large object.

KEY POINTS

- `std::cout` with `<<` prints output; `std::cin` with `>>` reads input and stops at whitespace.
- `std::getline` reads a whole line, spaces included.
- A reference is an alias that must be initialised and cannot be re-bound.
- Pass by reference to change the caller's variable, and by `const` reference to read large objects without copying.

TRY THESE EXERCISES

1. Write a program that reads the user's full name with `std::getline` and then their age with `std::cin`, and prints a greeting with their age next year; then change it to read the name with `std::cin`, type a two-word name, and note what goes wrong.
2. Write a function `void addBonus(int& salary, int bonus)` and a second version that takes salary by value instead; call both on a salary of 25000 with a bonus of 3000 and print the salary after each call to see which one changes it.
3. Write a function `void printBill(const std::string& item, int qty, double rate)` that prints one bill line, call it for three items from a stationery shop, then add a line inside it that tries to change item and note the error the compiler reports.

CHAPTER 2

Classes, Objects and Access Specifiers

Classes and objects

A class is a user-defined type that bundles data, called data members, with the functions that work on that data, called member functions. An object is one instance of a class, just as `int x;` creates one integer. The class is the plan, and every object built from it has its own copy of the data members. For example, a `BankAccount` class describes what every account holds and can do, while the accounts of Kavya and Rohit are two separate objects with separate balances.

Access specifiers

- `public` members can be used from anywhere, including `main`.
- `private` members can be used only by the class's own member functions (and by its friends).
- `protected` members behave like private ones, except that derived classes can also use them. You will meet them again in the lesson on inheritance.

Members of a `class` are private by default, and members of a `struct` are public by default. Apart from such defaults, the two keywords work the same way.

Hiding data behind public functions is called encapsulation. The class below never lets the balance go negative, because outside code cannot change it directly:

```

#include <iostream>
#include <string>

class BankAccount {
private:
    std::string holder;
    double balance = 0;

public:
    void open(const std::string& name) { holder = name; }

    void deposit(double amount)
    {
        if (amount > 0)
            balance += amount;
    }

    bool withdraw(double amount)
    {
        if (amount <= 0 || amount > balance)
            return false;
        balance -= amount;
        return true;
    }

    double getBalance() const { return balance; }
    std::string getHolder() const { return holder; }
};

int main()
{
    BankAccount acc;
    acc.open("Kavya Nair");
    acc.deposit(5000);
    if (!acc.withdraw(7000))
        std::cout << "Insufficient balance\n";
    acc.withdraw(1500);
    std::cout << acc.getHolder() << ": Rs " << acc.getBalance() << '\n';
    // acc.balance = 1000000; // error: balance is private
    return 0;
}

```

The program prints `Insufficient balance` and then `Kavya Nair: Rs 3500`.

Member functions in detail

- The `const` after `getBalance()` promises that the function does not change the object, so it can also be called on a `const` object.
- Writing `double balance = 0;` inside the class gives the member a default value, which is allowed since C++11.
- Inside a member function, the hidden pointer `this` points to the object the function was called on.

Longer functions are usually declared inside the class and defined outside it with the scope resolution operator `::`:

CPP TOP

```
class Rectangle {
    double length = 0, width = 0;

public:
    void set(double l, double w);
    double area() const;
};

void Rectangle::set(double l, double w)
{
    length = l;
    width = w;
}

double Rectangle::area() const
{
    return length * width;
}
```

`Rectangle::area` reads as "the `area` function that belongs to `Rectangle`". You then use the dot operator on an object: after `Rectangle r;` and `r.set(4, 2.5);`, the call `r.area()` returns 10.

KEY POINTS

- A class is a blueprint; an object is one instance with its own data members.
- private members are reachable only inside the class, public members from anywhere.
- class members are private by default and struct members are public by default.
- A const member function promises not to change the object.
- The scope resolution operator `::` defines a member function outside its class.

TRY THESE EXERCISES

1. Type the `BankAccount` class from this lesson, add a member function that prints a mini statement with the holder's name and balance, create accounts for two friends, and make several deposits and withdrawals on each to show that their balances stay separate.
2. Write a `Rectangle` class with private `length` and `width`, a `set` function that ignores negative values, and const member functions `area()` and `perimeter()` that are defined outside the class with the scope resolution operator `::`; check that `length 4` and `width 2.5` give an area of 10 and a perimeter of 13.
3. In `main`, try to change the private `balance` of a `BankAccount` object directly and copy the compiler error into your notebook; then change the keyword `class` to `struct` and delete the `private:` label, and confirm that the same line now compiles because struct members are public by default.

Constructors and Destructors

Constructors

A constructor is a special member function that runs automatically when an object is created. It has the same name as the class and no return type, not even `void`. Its job is to put the new object into a valid starting state.

A class can have several constructors, as long as their parameter lists differ:

- A default constructor takes no arguments.
- A parameterised constructor takes values to start from.
- A copy constructor builds a new object from an existing object of the same class. It is called when you write `Student c = a;` or pass an object by value.

If you write no constructor at all, the compiler supplies a default one. As soon as you declare any constructor yourself, it no longer does, so a class whose only constructor is `Box(int side)` cannot be created with `Box b;`.

Member initialiser lists

The cleanest way to set members is an initialiser list, written after a colon, which initialises them directly instead of assigning to them afterwards. Members that are `const` or references cannot be assigned to, so a constructor sets them in its initialiser list.

Destructors

A destructor runs automatically when an object is destroyed, for example when a local object goes out of scope. Its name is the class name with a tilde in front, such as `~Student()`. It takes no parameters, returns nothing, and a class has only one. Local objects are destroyed in the reverse order of their creation.

C++

```
#include <iostream>
#include <string>

class Student {
    std::string name;
    int roll;

public:
    Student() : name("Unknown"), roll(0)
    {
        std::cout << "Default constructor\n";
    }

    Student(const std::string& n, int r) : name(n), roll(r)
    {
        std::cout << "Created " << name << " (roll " << roll << ")\n";
    }

    Student(const Student& other) : name(other.name), roll(other.roll)
    {
        std::cout << "Copied " << name << '\n';
    }

    ~Student()
    {
        std::cout << "Destroyed " << name << '\n';
    }
};

int main()
{
    Student a("Arjun", 1);
    Student b("Meera", 2);
    Student c = a;
    return 0;
}
```

Output:

TEXT

```
Created Arjun (roll 1)
Created Meera (roll 2)
Copied Arjun
Destroyed Arjun
Destroyed Meera
Destroyed Arjun
```

The first `Destroyed Arjun` belongs to `c`, the copy, because it was created last. Writing `Student d;` would print `Default constructor`.

Why destructors matter

When a class acquires a resource, such as memory from `new`, an open file or a network connection, the destructor is the place to release it. Because C++ calls destructors automatically, the resource is freed even when a function returns early. This idea is called RAII, short for Resource Acquisition Is Initialisation, and standard classes such as `std::string`, `std::vector` and `std::ifstream` rely on it: a file stream closes its file when the object is destroyed.

KEY POINTS

- A constructor has the class's name and no return type, and runs when an object is created.
- Declaring any constructor stops the compiler from generating the default constructor.
- A copy constructor creates a new object from an existing one of the same class.
- The destructor, named with a tilde, runs automatically when an object is destroyed.
- Local objects are destroyed in the reverse order of their creation.

TRY THESE EXERCISES

1. Type the Student class from this lesson and run it, then add `Student d;` as the last line before `return` in `main` and write down the complete output you expect, including the order of the `Destroyed` lines, before running it again to check.
2. Write a `Product` class whose only constructor takes a name and a price and sets them in a member initialiser list, try to create an object with `Product p;` and note the compiler error, then fix it by adding a default constructor.
3. Write a class `Guard` whose constructor prints `Entering` and whose destructor prints `Leaving`, create a `Guard` object at the top of a function that has two `return` statements, and call the function so that each `return` is taken once to confirm that `Leaving` is printed both times, which is the idea behind RAII.

CHAPTER 4

Inheritance

What inheritance gives you

Inheritance lets a new class, the derived class, reuse and extend an existing class, the base class. It models an "is-a" relationship: a manager is an employee, so `Manager` can inherit from `Employee` and add only what is new.

```
#include <iostream>
#include <string>

class Employee {
protected:
    std::string name;
    double salary;

public:
    Employee(const std::string& n, double s) : name(n), salary(s)
    {
        std::cout << "Employee constructor\n";
    }

    void show() const
    {
        std::cout << name << " earns Rs " << salary << '\n';
    }
};

class Manager : public Employee {
    int teamSize;

public:
    Manager(const std::string& n, double s, int t) : Employee(n, s), teamSize(t)
    {
        std::cout << "Manager constructor\n";
    }

    void showTeam() const
    {
        std::cout << name << " leads " << teamSize << " people\n";
    }
};

int main()
{
    Manager m("Sanjay Verma", 85000, 6);
    m.show();           // inherited from Employee
    m.showTeam();       // added by Manager
    return 0;
}
```

Output:

TEXT

```
Employee constructor
Manager constructor
Sanjay Verma earns Rs 85000
Sanjay Verma leads 6 people
```

Constructor and destructor order

The base part of an object is built first, so the base constructor runs before the derived one. The derived constructor passes arguments to it in its initialiser list, as in `: Employee(n, s)`. Destructors run in the reverse order: derived first, then base.

Access and inheritance

With public inheritance, each base member is reachable as follows:

Base member	Usable in the derived class	Usable from outside, such as in main
public	Yes	Yes
protected	Yes	No
private	No	No

Private members are still part of the derived object; the derived class simply cannot access them directly and must use the base class's public or protected functions instead.

If you leave out the access word, `class Dog : Animal` uses private inheritance, while `struct Dog : Animal` uses public inheritance. Write `public` explicitly, because it is what you want almost every time.

Types of inheritance

- **Single:** one derived class from one base, such as `Manager` from `Employee`.
- **Multilevel:** a chain, such as `C` from `B` and `B` from `A`.
- **Hierarchical:** several classes from one base, such as `Car` and `Bike` from `Vehicle`.
- **Multiple:** one class from two or more bases, as in `class SmartPhone : public Phone, public Camera`.
- **Hybrid:** a mix of the above.

Multiple inheritance can lead to the "diamond problem", where a class receives two copies of a common base. C++ solves it with virtual inheritance, which keeps a single shared copy.

KEY POINTS

- A derived class inherits the members of its base class and can add its own.
- The base constructor runs before the derived constructor; destructors run in reverse.
- protected members are usable in derived classes but not from outside code.
- Write `public` before the base class name; class inheritance is private by default.

TRY THESE EXERCISES

1. Type the Employee and Manager classes from this lesson, add a destructor to each class that prints a message, and confirm from the output that the constructors run base first while the destructors run derived first.
2. Create a base class Vehicle with protected members brand and wheels, and two derived classes Car and Bike that each add one member of their own and a show function; create one object of each in main, then try to print brand directly from main and note the error.
3. Build a multilevel chain in which Student inherits publicly from Person and Topper inherits publicly from Student, give each class one data member and a constructor that passes values to its base through the initialiser list, and print every detail from a single Topper object.

CHAPTER 5

Polymorphism and Virtual Functions

One interface, many forms

Polymorphism means one name can behave differently depending on the type it works with. C++ has two kinds:

- **Compile-time polymorphism:** function overloading, operator overloading and templates. The compiler picks the right version while compiling.
- **Run-time polymorphism:** virtual functions. The program picks the right version while it runs, based on the actual object.

The problem virtual functions solve

A base class pointer or reference can refer to a derived object. Without `virtual`, a call through it uses the base version:

```
#include <iostream>

class Shape {
public:
    void draw() const { std::cout << "Shape\n"; }
};

class Circle : public Shape {
public:
    void draw() const { std::cout << "Circle\n"; }
};

int main()
{
    Circle c;
    Shape& s = c;
    s.draw();    // prints Shape
    return 0;
}
```

The compiler chose `Shape::draw` from the declared type of `s`. This is static, or early, binding. Add `virtual` before `void draw()` in `Shape` and the same call prints `Circle`, because the function is now chosen from the object's real type. This is dynamic, or late, binding.

A practical example

CPP

```
#include <iostream>

class Payment {
public:
    virtual void pay(double amount) const = 0;    // pure virtual
    virtual ~Payment() = default;
};

class UpiPayment : public Payment {
public:
    void pay(double amount) const override
    {
        std::cout << "Paid Rs " << amount << " by UPI\n";
    }
};

class CardPayment : public Payment {
public:
    void pay(double amount) const override
    {
        std::cout << "Paid Rs " << amount << " by card\n";
    }
};

void checkout(const Payment& method, double amount)
{
    method.pay(amount);
}

int main()
{
    UpiPayment upi;
    CardPayment card;
    checkout(upi, 499);    // Paid Rs 499 by UPI
    checkout(card, 1250); // Paid Rs 1250 by card
    return 0;
}
```

`checkout` knows nothing about UPI or cards, yet each payment behaves correctly. A new payment type can be added later without changing `checkout` at all.

Key terms

- **Pure virtual function:** a virtual function declared with `= 0` in the base class. A class with at least one is an abstract class, and you cannot create objects of it. A derived class must override every pure virtual function before you can create objects of it.
- **override (C++11):** written after a function in a derived class, it makes the compiler report an error if the function does not actually override a virtual function of the base, for example because of a spelling mistake or a missing `const`.

- **Virtual destructor:** if objects are deleted through a base class pointer, the base destructor must be virtual. Otherwise the behaviour is undefined, and in practice the derived destructor is usually skipped.

Mark a function virtual when derived classes are meant to replace it, and give every class with virtual functions a virtual destructor.

KEY POINTS

- Overloading and templates are compile-time polymorphism; virtual functions are run-time polymorphism.
- Without virtual, a call through a base pointer or reference uses the base version.
- A class with a pure virtual function ($= 0$) is abstract and cannot be instantiated.
- `override` asks the compiler to confirm that a function really overrides a base virtual function.
- Give a base class with virtual functions a virtual destructor, so deleting through a base pointer is safe.

TRY THESE EXERCISES

1. Type the Shape and Circle program from this lesson and run it, then add virtual before void `draw()` in Shape and run it again; write down both outputs and explain in one sentence why the second one changed.
2. Add a third class, `CashPayment`, to the Payment example that overrides `pay`, and call `checkout` with it without changing the `checkout` function at all; then misspell `pay` in your new class and note the error that `override` makes the compiler report.
3. Write an abstract class Shape with a pure virtual function `area()` and a virtual destructor, derive Circle and Rectangle from it using `override`, store the addresses of one of each in an array of two Shape pointers and print every area in a loop; then try to create a Shape object directly and note the compiler error.

CHAPTER 6

Function Overloading, Operator Overloading and Templates

Function overloading

Several functions can share one name if their parameter lists differ in the number or types of parameters. The compiler picks the version that matches the arguments:

```
int area(int side) { return side * side; }
int area(int length, int width) { return length * width; }
double area(double radius) { return 3.14159 * radius * radius; }
```

`area(5)` returns 25, `area(4, 6)` returns 24, and `area(2.0)` calls the `double` version. A different return type alone is not enough: `int total(int)` and `double total(int)` cannot both exist, because a call such as `total(3)` could not tell them apart. Default arguments are a related tool: `void printBill(int copies = 1);` can be called as `printBill()` or `printBill(3)`.

Operator overloading

You can define what an operator means for your own class by writing a function named `operator` followed by the symbol. Here a `Money` class supports `+` and works with `std::cout`:

CPP

```
#include <iostream>

class Money {
    long paise;    // whole paise avoid rounding errors

public:
    Money(long rupees, long p = 0) : paise(rupees * 100 + p) {}

    Money operator+(const Money& other) const
    {
        Money result(0);
        result.paise = paise + other.paise;
        return result;
    }

    friend std::ostream& operator<<(std::ostream& out, const Money& m)
    {
        long p = m.paise % 100;
        out << "Rs " << m.paise / 100 << (p < 10 ? ".0" : ".") << p;
        return out;
    }
};

int main()
{
    Money a(250, 50), b(99, 75);
    std::cout << a + b << '\n';    // Rs 350.25
    return 0;
}
```

The rules: you cannot invent new operators, change an operator's precedence, or redefine operators for built-in types such as `int`. A few operators cannot be overloaded at all, including `::`, `.`, `?:` and `sizeof`. `operator<<` cannot be a member of `Money`, because its left operand is the stream, not a `Money` object; it is declared a `friend` so that it can read the private `paise` member.

Templates

A template lets you write a function or class once and use it with many types. The compiler generates a separate version for each type you use:

CPP TOP

```
template <typename T>
T largest(T a, T b)
{
    return (a > b) ? a : b;
}
```

`largest(3, 9)` returns 9 and `largest(2.5, 1.5)` returns 2.5. For text, pass `std::string` objects, as in `largest(std::string("apple"), std::string("mango"))`, which returns `"mango"`. Plain string literals would make the template compare memory addresses instead of text. A template is only fully checked when it is used with a particular type: if that type lacks an operator the template needs, such as `>` in `largest`, the code fails to compile.

Class templates work the same way:

CPP TOP

```
template <typename T>
class Box {
    T value;

public:
    explicit Box(T v) : value(v) {}
    T get() const { return value; }
};
```

`Box<int> b(5);` stores an `int`, and `Box<std::string>` stores a string. Since C++17 the compiler can often deduce the type from the constructor argument, so `Box b(5);` also works. The STL containers, from `std::vector` to `std::map`, are all class templates.

KEY POINTS

- Overloaded functions share a name but differ in the number or types of parameters.
- A different return type alone cannot overload a function.
- Operator overloading gives operators a meaning for your own classes; `::`, `.`, `?:` and `sizeof` cannot be overloaded.
- A template is written once and works with many types.

TRY THESE EXERCISES

1. Write three overloaded functions named `area` for a square, a rectangle and a circle, call each one from `main`, and then add a fourth version that differs from one of them only in its return type to see the compiler reject it.
2. Type the `Money` class from this lesson, add an operator- that subtracts one amount from another, and print Rs 350.25 minus Rs 99.75 to check that the answer is Rs 250.50.
3. Write a function template `largest` that returns the larger of two values and test it with two ints, two doubles and two `std::string` objects; then write a class template `Pair` that stores two values of the same type with a member function that returns their sum, and test it with `int` and `double`.

CHAPTER 7

STL Containers and Exceptions

The Standard Template Library

The STL is the part of the C++ standard library that provides ready-made containers, algorithms such as `std::sort` and `std::find`, and iterators that connect the two. Three containers cover most beginner programs. Each one is a class template, so you write the element type in angle brackets, as in `std::vector<int>`, and include its header, such as `<vector>`, `<string>` or `<map>`.

`std::vector`

A `vector` is a dynamic array that grows as you add elements.

CPP FRAGMENT

```
std::vector<int> marks = {72, 88, 65};
marks.push_back(91);           // add at the end
std::cout << marks.size() << '\n'; // 4
std::cout << marks[0] << '\n';    // 72
int total = 0;
for (int m : marks)            // range-based for loop
    total += m;
std::cout << total << '\n';      // 316
```

`marks[i]` does no range check, while `marks.at(i)` checks the index and throws `std::out_of_range` if it is invalid. Other useful members are `pop_back()`, `front()`, `back()`, `empty()` and `clear()`.

std::string

CPP FRAGMENT

```
std::string city = "Hyderabad";
std::cout << city.length() << '\n';    // 9
std::cout << city.substr(0, 5) << '\n'; // Hyder
std::cout << city.find("bad") << '\n';  // 6
```

`substr(pos, len)` takes `len` characters starting at position `pos`. `find` returns the position of the first match, or `std::string::npos` when there is none. Strings compare with `==` and `<`, and join with `+`.

std::map

A `map` stores key and value pairs. Keys are unique and kept in sorted order.

CPP FRAGMENT

```
std::map<std::string, int> stock;
stock["pen"] = 40;
stock["notebook"] = 25;
stock["eraser"] = 60;
stock["pen"] -= 5;                                // update an existing key

for (const auto& [item, qty] : stock)             // C++17 structured binding
    std::cout << item << ": " << qty << '\n';
```

This prints `eraser: 60`, `notebook: 25` and `pen: 35`, in alphabetical order of the keys. Take care: using `[]` with a missing key silently inserts it with a default value, which is 0 for an `int`. Use `count` or `find` to test for a key without inserting it.

Exceptions

Exceptions separate error handling from normal code. A function that meets a problem it cannot handle uses `throw`; the caller wraps risky code in a `try` block and handles errors in `catch` blocks.

```

#include <iostream>
#include <stdexcept>
#include <vector>

double divide(double a, double b)
{
    if (b == 0)
        throw std::invalid_argument("division by zero");
    return a / b;
}

int main()
{
    std::vector<int> v = {10, 20, 30};
    try {
        std::cout << divide(100, 4) << '\n';    // 25
        std::cout << v.at(5) << '\n';           // throws
    } catch (const std::out_of_range&) {
        std::cout << "Bad index\n";
    } catch (const std::exception& e) {
        std::cout << "Error: " << e.what() << '\n';
    }
    return 0;
}

```

- Catch exceptions by `const` reference, and put specific types before general ones such as `std::exception`.
- `catch (...)` catches any exception at all.
- If no handler matches, the program ends by calling `std::terminate`.

When an exception is thrown, C++ immediately leaves every block and function between the `throw` and the matching handler, destroying their local objects on the way. This is called stack unwinding, and it is why RAII classes still release their resources when an error occurs.

KEY POINTS

- `vector` is a growable array; `at()` checks the index and throws `std::out_of_range`.
- `string` offers `length`, `substr`, `find` and comparison with `==`.
- `map` keeps unique keys in sorted order, and `[]` inserts a missing key.
- `throw` raises an exception, and the matching `catch` block after a `try` handles it.
- Catch exceptions by `const` reference, specific types first.

TRY THESE EXERCISES

1. Write a program that reads marks until the user enters -1, stores them in a `std::vector` of `int` with `push_back`, and prints how many marks were entered, their total, their average and the highest mark using a range-based for loop; if the user types -1 straight away, check `empty()` and print No marks entered instead of dividing by zero.
2. Keep the stock of five stationery items in a `std::map` from item name to quantity, let the user type an item name and print its quantity, and use `find` or `count` so that a name not in the map prints Not in stock instead of being added to the map.
3. Write a divide function that throws `std::invalid_argument` when the divisor is 0, call it inside a try block with two numbers typed by the user, catch it with a catch block for `const std::exception&` and print `e.what()`; then, inside the same try block, add a call to `at(10)` on a three-element vector and a separate catch block for `std::out_of_range` placed before the general `std::exception` one.

CHECK YOURSELF

Practice questions

Choose the best answer for each question. The answers, with a short explanation, are in the next chapter.

1. Which object prints output to the console in standard C++?

- a) `std::cout`
- b) `std::cin`
- c) `std::string`
- d) `std::vector`

2. What does this code print? `int a = 10; int& r = a; r = 20; std::cout << a;`

- a) 10
- b) 20
- c) 30
- d) The address of a

3. Which statement about C++ references is true?

- a) A reference can later be changed to refer to a different variable
- b) A reference must be initialised when it is declared
- c) A reference can be set to null
- d) A reference always takes exactly 8 bytes of memory

4. A user types Ravi Kumar when this runs: `std::string name; std::cin >> name;` What does name hold?

- a) Ravi Kumar
- b) Ravi
- c) Kumar
- d) An empty string

5. Why is a large `std::string` often passed to a function as `const std::string&`?

- a) It makes a shorter copy of the string for the function to use
- b) It lets the function change the caller's string directly
- c) It avoids copying the string while stopping the function from changing it
- d) It converts the string into a C-style char array before the call

6. What is an object in C++?

- a) An instance of a class, with its own copy of the data members
- b) A function that belongs to a class and works on its data
- c) A keyword used in place of class to declare a new type
- d) A header file that stores the declaration of a class

7. What is the difference between class and struct in C++?

- a) A struct can hold only data members, while a class can also have member functions
- b) Members of a class are private by default, while members of a struct are public by default
- c) A class cannot have data members, only member functions and constructors
- d) Only a class can be used to create objects; a struct only describes a memory layout

8. Which operator is used to define a member function outside its class, as in `void Account::show()`?

- a) `.` (member access on an object)
- b) `->` (member access through a pointer)
- c) `::` (scope resolution)
- d) `#` (preprocessor directive)

9. What does the `const` in `double getBalance() const` mean?

- a) The function returns a value that can never change anywhere in the program
- b) The function promises not to change the object's data members
- c) The function can be called only once
- d) The function is private

10. `balance` is a private data member of class `Account`. What happens when `main` contains `acc.balance = 100; ?`

- a) `balance` becomes 100, because `main` may change any member
- b) The program compiles but crashes when that line runs
- c) The statement compiles but is silently ignored at run time
- d) The compiler reports an error because `balance` is not accessible there

11. Which statement about constructors is true?

- a) A constructor must return an `int`
- b) A constructor has the same name as its class and no return type
- c) A constructor must be called by hand after an object is created
- d) A class can have only one constructor

12. What is the name of the destructor of a class called `Invoice`?

- a) `~Invoice`
- b) `Invoice`
- c) `!Invoice`
- d) `delete Invoice`

13. What does this program print? `class T { char c; public: T(char x) : c(x) {} ~T() { std::cout << c; }; int main() { T a('X'); T b('Y'); }`

- a) XY
- b) X
- c) Y
- d) YX

14. When is a copy constructor called?

- a) When an object goes out of scope at the end of a block
- b) When a new object is initialised from an existing object of the same class
- c) When two objects of the same class are compared with ==
- d) Every time a member function is called on an existing object

15. A class has only one constructor, Box(int side). What happens with the statement Box b; ?

- a) b is created, and the compiler sets side to 0 automatically
- b) b is created, with side holding an unpredictable value
- c) The compiler calls Box(int) with the value 1
- d) It does not compile, because the class has no default constructor

16. A base class member is declared protected. Where can it be used?

- a) In the base class and its derived classes, but not from outside code such as main
- b) Anywhere in the program, including main and unrelated classes
- c) Only inside the base class, never in its derived classes
- d) Only in main and other free functions, not in member functions

17. When an object of a derived class is created, which constructor runs first?

- a) The derived class constructor
- b) Whichever constructor appears first in the file
- c) Only the derived class constructor runs
- d) The base class constructor

18. Class C inherits from B, and B inherits from A. What type of inheritance is this?

- a) Multiple
- b) Hierarchical
- c) Hybrid
- d) Multilevel

19. Which line makes Car publicly inherit from Vehicle in C++?

- a) class Car extends Vehicle { };
- b) class Car inherits Vehicle { };
- c) class Car : public Vehicle { };
- d) class Car :: Vehicle { };

20. With no access word, as in class Dog : Animal { }; what kind of inheritance does C++ use?

- a) Public
- b) Protected
- c) It is a compile error
- d) Private

21. What does this code print? struct A { void f() { std::cout << "A"; } }; struct B : A { void f() { std::cout << "B"; } }; B b; A& r = b; r.f();

- a) B
- b) A
- c) AB
- d) It does not compile

22. What does declaring a member function virtual in a base class achieve?

- a) A call through a base pointer or reference runs the version that matches the object's actual type
- b) Derived classes can no longer provide their own version of the function
- c) The function runs faster, because the compiler inlines every call to it
- d) The function can be called on the class itself, without creating an object

23. What makes a class abstract in C++?

- a) Every one of its constructors is declared private
- b) It has member functions but no data members
- c) It has at least one pure virtual function, declared with = 0
- d) It is declared with the keyword abstract before class

24. Why should a base class with virtual functions have a virtual destructor?

- a) So that objects of the class can be copied safely with the copy constructor
- b) So that the derived constructor runs before the base constructor
- c) So that no other class can inherit from the base class
- d) So that deleting a derived object through a base pointer also runs the derived destructor

25. What does the override keyword do when written after a member function in a derived class?

- a) Makes the base class version virtual, even if it was not declared virtual
- b) Makes the compiler report an error if the function does not actually override a base virtual function
- c) Removes the base class version so that it can never be called again
- d) Lets the function differ from the base version only in its return type

26. Which pair of functions cannot both be declared in the same scope?

- a) `int total(int a);` and `int total(int a, int b);`
- b) `int total(int a);` and `double total(double a);`
- c) `int total(int a);` and `double total(int a);`
- d) `void print(int n);` and `void print(char c);`

27. Given template `<typename T> T twice(T x) { return x + x; }` what does `twice(std::string("ab"))` return?

- a) "abab"
- b) "ab"
- c) "2ab"
- d) It does not compile

28. Which of these operators cannot be overloaded in C++?

- a) `+` (addition)
- b) `<<` (left shift and stream output)
- c) `::` (scope resolution)
- d) `==` (equality comparison)

29. What is the main benefit of templates?

- a) Programs can run straight from the source code without compiling
- b) Data members are automatically hidden from all other classes
- c) One function or class can be written once and used with many data types
- d) Objects can be created without any constructor being called

30. When does C++ decide which version of an overloaded function to call?

- a) At compile time, from the number and types of the arguments
- b) At run time, from the actual type of the object
- c) At random, each time the program runs
- d) At link time, from the order the functions appear in the file

31. What does `v.size()` return after `std::vector<int> v = {4, 8}; v.push_back(15);` ?

- a) 3
- b) 2
- c) 15
- d) 27

32. What is the difference between `v[i]` and `v.at(i)` for a `std::vector`?

- a) `v[i]` checks the index and throws an exception, while `at(i)` does no check
- b) `at(i)` returns a copy, while `v[i]` returns the element's address
- c) `at(i)` can be used only with vectors of strings, while `v[i]` works with any type
- d) `at(i)` checks the index and throws `std::out_of_range` if it is invalid, while `v[i]` does no check

33. A `std::map<std::string, int>` holds the keys "pen", "bag" and "ink". In what order does a range-based for loop visit them?

- a) bag, ink, pen
- b) pen, bag, ink
- c) ink, pen, bag
- d) A different random order on each run

34. Which block handles an exception thrown inside a try block?

- a) The finally block written after the try block
- b) The return statement at the end of main
- c) The constructor of the class that threw it
- d) A matching catch block that follows the try block

35. What does this code print? `std::string s = "Hyderabad"; std::cout << s.substr(0, 5);`

- a) Hyde
- b) Hydera
- c) Hyder
- d) abad

ANSWER KEY

Answers

- 1. **(a) `std::cout`.** `std::cout` is the standard output stream; values are sent to it with the `<<` operator.
- 2. **(b) 20.** `r` is a reference, another name for `a`, so assigning 20 to `r` changes `a`.
- 3. **(b) A reference must be initialised when it is declared.** A reference must be bound to a variable when declared and stays bound to it; C++ has no null references.
- 4. **(b) Ravi.** The `>>` operator stops reading at whitespace, so only the first word is stored; `std::getline` reads the whole line.
- 5. **(c) It avoids copying the string while stopping the function from changing it.** A reference avoids the copy, and `const` makes the compiler reject any attempt to modify the caller's object.
- 6. **(a) An instance of a class, with its own copy of the data members.** A class is the blueprint; each object created from it is an instance holding its own values for the data members.
- 7. **(b) Members of a class are private by default, while members of a struct are public by default.** Both can have data, functions, constructors and inheritance; they differ in the default access level.
- 8. **(c) `::` (scope resolution).** The scope resolution operator `::` tells the compiler that `show` belongs to the class `Account`.
- 9. **(b) The function promises not to change the object's data members.** A `const` member function cannot modify the object, which also lets it be called on `const` objects.

10. (d) **The compiler reports an error because balance is not accessible there.** Private members can be used only inside the class's own member functions (and by its friends), so the compiler rejects the access.
11. (b) **A constructor has the same name as its class and no return type.** Constructors run automatically when an object is created; a class can have several, as long as their parameter lists differ.
12. (a) **~Invoice.** A destructor is named with a tilde followed by the class name, and it takes no parameters.
13. (d) **YX.** Both objects are destroyed when main ends, in reverse order of creation, so b prints Y before a prints X.
14. (b) **When a new object is initialised from an existing object of the same class.** Statements such as `Student c = a;` and passing an object by value create a new object from an existing one, which uses the copy constructor.
15. (d) **It does not compile, because the class has no default constructor.** Once a class declares any constructor, the compiler no longer generates a default constructor.
16. (a) **In the base class and its derived classes, but not from outside code such as main.** protected works like private for outside code but also gives derived classes access.
17. (d) **The base class constructor.** The base part of the object is built first, then the derived part; destructors run in the reverse order.
18. (d) **Multilevel.** A chain of classes, each derived from the one before, is multilevel inheritance; multiple inheritance means one class with two or more direct bases.
19. (c) **class Car : public Vehicle { };** C++ lists the base class after a colon with an access specifier; extends is Java syntax.
20. (d) **Private.** A class inherits privately by default, while a struct inherits publicly by default.
21. (b) **A.** f is not virtual, so the call is decided by the declared type of r, which is A.
22. (a) **A call through a base pointer or reference runs the version that matches the object's actual type.** Virtual functions use dynamic binding, so the derived class's override runs even when the object is reached through a base pointer or reference.
23. (c) **It has at least one pure virtual function, declared with = 0.** A class with a pure virtual function cannot be instantiated; C++ has no abstract keyword.
24. (d) **So that deleting a derived object through a base pointer also runs the derived destructor.** Without a virtual destructor, deleting through a base pointer is undefined behaviour and the derived clean-up is usually skipped.
25. (b) **Makes the compiler report an error if the function does not actually override a base virtual function.** override, added in C++11, catches mistakes such as a misspelt name or a missing const that would otherwise create a new function silently.
26. (c) **int total(int a); and double total(int a);** Overloads must differ in their parameter lists; two functions that differ only in return type cannot be told apart by a call.
27. (a) **"abab".** T becomes std::string, and adding two strings joins them, so the result is "abab".
28. (c) **:: (scope resolution).** The scope resolution operator, the dot operator, ?: and sizeof are among the operators C++ does not allow you to overload.
29. (c) **One function or class can be written once and used with many data types.** The compiler generates a version of the template for each type it is used with, so the code is written only once.

- 30. (a) At compile time, from the number and types of the arguments.** Overload resolution is compile-time polymorphism: the compiler matches the call's arguments against each version's parameters.
- 31. (a) 3.** `push_back` adds one element to the two already there, so the vector holds 3 elements.
- 32. (d) `at(i)` checks the index and throws `std::out_of_range` if it is invalid, while `v[i]` does no check.** Using `[]` with a bad index is undefined behaviour; `at()` checks the index and throws an exception you can catch.
- 33. (a) bag, ink, pen.** A `std::map` keeps its keys sorted, so it visits them in alphabetical order, not in the order they were added.
- 34. (d) A matching catch block that follows the try block.** C++ hands the exception to the first following catch block whose type matches; C++ has no finally keyword.
- 35. (c) Hyder.** `substr(0, 5)` returns 5 characters starting at position 0, which are H, y, d, e and r.

WORD LIST

Glossary

Abstract class

A class with at least one pure virtual function, so no objects of it can be created and it serves only as a base for other classes.

Access specifier

One of the keywords `public`, `private` or `protected`, which decide where a member of a class can be used from.

Class

A user-defined type that bundles data members with the member functions that work on them, acting as the plan from which objects are made.

Constructor

A special member function with the same name as its class and no return type that runs automatically when an object is created, to give it a valid starting state.

Copy constructor

A constructor that builds a new object from an existing object of the same class, used for example when one object is initialised from another or passed by value.

Destructor

A member function named with a tilde before the class name that runs automatically when an object is destroyed, usually to release resources the object holds.

Encapsulation

Keeping a class's data private and allowing changes only through its public member functions, so that the class can keep its data valid.

Exception

An object raised with `throw` to signal an error the current code cannot handle, which is then handled by a matching catch block after a `try` block.

Function overloading

Defining several functions with the same name but different numbers or types of parameters, with the compiler choosing the one that matches each call.

Inheritance

A feature that lets a derived class reuse and extend the members of a base class, modelling an is-a relationship such as a manager being an employee.

Member initialiser list

The list written after a colon in a constructor, such as : name(n), roll(r), which initialises data members directly instead of assigning to them in the constructor body.

Namespace

A named scope that groups related names so they do not clash with others; the standard library's names, such as cout and string, belong to the std namespace.

Object

One instance of a class, with its own copy of the class's data members.

Operator overloading

Giving an operator such as + or == a meaning for your own class by writing a function named operator followed by the symbol.

override

A specifier written after a member function in a derived class that makes the compiler report an error if the function does not actually override a virtual function of the base class.

Polymorphism

The ability of one name to behave differently depending on the type it works with, achieved at compile time through overloading and templates and at run time through virtual functions.

Pure virtual function

A virtual function declared with = 0 in a base class, which a derived class must override before objects of that derived class can be created.

RAII

Short for Resource Acquisition Is Initialisation, the C++ idea of tying a resource such as memory or an open file to an object so that the object's destructor releases it automatically.

Reference

Another name, or alias, for an existing variable, declared with & after the type, which must be initialised and cannot later be made to refer to a different variable.

Scope resolution operator

The :: operator, used to name something that belongs to a class or namespace, as in Rectangle::area or std::cout.

Stack unwinding

The process, after an exception is thrown, of leaving every block and function up to the matching catch handler and destroying their local objects on the way.

STL

The Standard Template Library, the part of the C++ standard library that provides containers such as vector and map, algorithms such as sort and find, and the iterators that connect them.

Template

A function or class written once with a type parameter, from which the compiler generates a separate version for each type it is used with.

this pointer

A hidden pointer available inside a member function that points to the object on which the function was called.

Virtual function

A member function declared with the keyword virtual in a base class, so that a call through a base pointer or reference runs the version that belongs to the object's actual type.



Gurukul Technicals Institute

Computer education institute in Delhi, with authorised member centres, robotics classes and science projects, working models and DIY kits.

COURSES

Computer & Office Skills · Accounting & Business Software · Programming & Web Development · Design & Digital Skills · Hardware & Networking · Robotics & Coding · Other Programmes

PROJECTS AND KITS

Ready-made school projects · DIY and STEM kits · Robotics and coding · Components and spares · Engineering projects

CONTACT

Head office: A-10, Chief Commissioner Colony, opposite Rana Pratap Bagh, Delhi 110007

Phone: 011 2721 1512 · **Mobile and WhatsApp:** +91 97116 99511

Email: engineergurukul@yahoo.co.in

Website: engineergurukul.com · **Instagram:** [@science_projects_club](https://www.instagram.com/science_projects_club)

More free ebooks for computer courses, robotics and science projects: engineergurukul.com/resources/ebooks/

Free to read, print and share. Not for sale.