



Gurukul Technicals Institute

Computer education · Robotics · Science projects · Delhi

FREE EBOOK

PROGRAMMING AND WEB

Object-Oriented Programming in C++

Classes, objects, inheritance and polymorphism with practice programs

INSIDE THIS BOOK

- From C to C++
- Classes and objects
- Constructors and destructors
- The this pointer, static members and friends
- Function and operator overloading
- ...and 3 more chapters, practice questions and a glossary

Intermediate

Students who know C basics and are learning C++

engineergurukul.com

+91 97116 99511

Object-Oriented Programming in C++

Move from C to C++ and learn object-oriented programming, from classes and constructors to inheritance, virtual functions, templates, exceptions, the STL and files. Nineteen practice programs, such as a bank account, shape areas and complex numbers, show every idea working.

WHO IT IS FOR	Students who know C basics and are learning C++
LEVEL	Intermediate
SUBJECT AREA	Programming and web
WHAT IS INSIDE	8 chapters, practice questions with answers, and a glossary

CONTENTS

1	From C to C++	3
2	Classes and objects	5
3	Constructors and destructors	8
4	The this pointer, static members and friends	10
5	Function and operator overloading	12
6	Inheritance and virtual functions	14
7	Templates and exception handling	20
8	The STL and file streams	22
	Practice questions	26
	Answers	27
	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.

From C to C++

C++ began as "C with classes". Almost every C program is also valid C++, so what you know about variables, loops, arrays and functions still works. The most important addition is the **class**, which lets you build your own types.

Feature	C	C++
Style	procedural: data and functions kept apart	object-oriented: data and functions joined in classes
Input and output	<code>printf, scanf</code>	<code>cout, cin</code>
Text	character arrays	the <code>string</code> class
Letting a function change a variable	pointers	references (pointers still work)
Handling errors	return codes	exceptions as well
Ready-made containers	none	the STL: <code>vector</code> , <code>map</code> and more

Compiling with g++

Save C++ files with the extension `.cpp` and compile them with `g++`, the C++ partner of `gcc`:

1. Save the program as `bill.cpp`.
2. In a terminal, type `g++ -std=c++17 -Wall bill.cpp -o bill`.
3. Run it with `./bill` on Linux or macOS, or `bill` on Windows.

`-std=c++17` asks for the C++17 standard, which every program in this book uses, and `-Wall` turns on warnings.

Program 1: A shop bill with cin and cout

CPP

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

int main() {
    string item;
    int qty;
    double price;
    cout << "Item: ";
    cin >> item;
    cout << "Quantity and price: ";
    cin >> qty >> price;
    cout << qty << " x " << item << " = Rs " << qty * price << '\n';
    return 0;
}
```

```
Item: Notebook
Quantity and price: 5 45
5 x Notebook = Rs 225
```

- `iostream` gives `cout` (console output) and `cin` (console input).
- `<<` is the **insertion operator**: it sends values to `cout`. `>>` is the **extraction operator**: it takes values from `cin`. No format specifiers are needed, because C++ knows each variable's type.
- `using namespace std;` lets us write `cout` instead of `std::cout`. A **namespace** is a family name that keeps library names apart from yours. In small programs this line is fine; in large projects, writing `std::` is safer.
- `'\n'` ends the line. You will also see `endl`, which does the same and also flushes (forces out) the output.

REMEMBER

`cin >> item` stops reading at a space, so "Gel pen" would give only "Gel". To read a whole line, use `getline(cin, item);`.

Program 2: References

A **reference** is a second name for an existing variable. It is declared with `&` after the type.

```

#include <iostream>
using namespace std;

void swapValues(int &a, int &b) {
    int t = a;
    a = b;
    b = t;
}

int main() {
    int x = 5, y = 9;
    swapValues(x, y);
    cout << "x = " << x << ", y = " << y << '\n';
    int &r = x;
    r = 100;
    cout << "x = " << x << '\n';
    return 0;
}

```

```

x = 9, y = 5
x = 100

```

Because `a` and `b` are references, they are `x` and `y` themselves, not copies, so the swap works. Compare the C version with pointers: here there is no `&` at the call and no `*` inside the function. `int &r = x;` makes `r` another name for `x`, so setting `r` changes `x`.

CHAPTER 2

Classes and objects

Object-oriented programming (OOP) builds a program out of objects, each holding its own data and the functions that work on that data. Think of a railway reservation system: it has passengers, trains and tickets, and each is naturally an object.

- A **class** is a design, like the architect's plan of a flat.
- An **object** is a real thing made from that design, like one actual flat. One plan can produce many flats.

Variables inside a class are **data members**. Functions inside a class are **member functions** (also called methods). Together they are the class's **members**.

Program 3: A Rectangle class

CPP

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

class Rectangle {
public:
    double length, width;
    double area() const { return length * width; }
};

int main() {
    Rectangle room;
    room.length = 4.5;
    room.width = 3;
    cout << "Area = " << room.area() << " sq m\n";
    return 0;
}
```

Area = 13.5 sq m

`class Rectangle { ... };` designs the type; note the semicolon after the closing brace. `Rectangle room;` creates an object. The dot reaches its members: `room.length`, `room.area()`. The word `const` after `area()` promises that this function only reads the data and never changes it.

Access specifiers

An **access specifier** decides who may use a member.

Specifier	Who can use the member
<code>public</code>	any code, including <code>main</code>
<code>private</code>	only the class's own member functions (and friends)
<code>protected</code>	the class itself and classes derived from it

Members of a `class` are private unless you say otherwise. Keeping data private and offering public functions to use it is called **encapsulation**. It is like an ATM: you can deposit and withdraw through the slot, but you cannot open the machine and change your balance by hand.

Program 4: A bank account

CPP

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

class BankAccount {
private:
    double balance = 0;
public:
    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; }
};

int main() {
    BankAccount acc;
    acc.deposit(5000);
    if (!acc.withdraw(8000)) cout << "Not enough balance\n";
    acc.withdraw(1200);
    cout << "Balance: Rs " << acc.getBalance() << '\n';
    return 0;
}
```

```
Not enough balance
Balance: Rs 3800
```

`balance` is private, so a line such as `acc.balance = 1000000;` in `main` would not compile. The only ways to change it are `deposit` and `withdraw`, and both check their input. `withdraw` returns `true` or `false` so the caller knows whether it worked.

TIP

Name a class after a thing (a noun, such as `BankAccount`) and its member functions after actions (verbs, such as `deposit`). Your code then reads almost like English.

Constructors and destructors

In Program 4, `balance = 0` written inside the class gave every account a starting value. A **constructor** goes further. It is a member function with the same name as the class and no return type, it runs automatically when an object is created, and it can take values to start the object with.

Kind	Example	Runs when
Default constructor	<code>Student()</code>	an object is made with no values
Parameterised constructor	<code>Student(string n, int m)</code>	values are given
Copy constructor	<code>Student(const Student &other)</code>	an object is made as a copy of another

A **destructor** has the class name with a tilde in front, such as `~Student()`. It runs automatically when the object is destroyed, for example at the end of the block where it was created. It is the place to release anything the object was holding, such as an open file.

Program 5: Constructors and destructor in action

CPP

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

class Student {
    string name;
    int marks;
public:
    Student() : name("Unknown"), marks(0) { cout << "Default constructor\n"; }
    Student(string n, int m) : name(n), marks(m) {
        cout << "Constructor for " << name << '\n';
    }
    Student(const Student &other) : name(other.name), marks(other.marks) {
        cout << "Copy of " << name << '\n';
    }
    ~Student() { cout << "Destructor for " << name << '\n'; }
    void show() const { cout << name << ": " << marks << '\n'; }
};

int main() {
    Student a;
    Student b("Kiran", 88);
    Student c = b;
    a.show();
    c.show();
    return 0;
}
```

```
Default constructor
Constructor for Kiran
Copy of Kiran
Unknown: 0
Kiran: 88
Destructor for Kiran
Destructor for Kiran
Destructor for Unknown
```

The part after the colon, `: name(n), marks(m)`, is a **member initialiser list**. It fills the members as the object is born, which is better than assigning them inside the braces. Look at the last three lines: objects are destroyed in the reverse order of creation, so `c` goes first and `a` last, like plates taken off a stack.

REMEMBER

If you write no constructor at all, C++ supplies a default one. As soon as you write any constructor, that free default constructor disappears, so `Student a;` would only compile if you write `Student()` yourself.

The this pointer, static members and friends

The this pointer

Inside a member function, `this` is a pointer to the object the function was called on. You need it in two common cases: when a parameter has the same name as a member, and when a function wants to return its own object.

Program 6: Chaining calls with this

CPP

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

class Order {
    string item;
    int qty = 0;
public:
    Order &setItem(const string &item) {
        this->item = item;
        return *this;
    }
    Order &setQty(int qty) {
        this->qty = qty;
        return *this;
    }
    void show() const { cout << qty << " x " << item << '\n'; }
};

int main() {
    Order o;
    o.setItem("Samosa").setQty(12).show();
    return 0;
}
```

12 x Samosa

In `this->item = item`, the left side is the member and the right side is the parameter. Each setter returns `*this`, the object itself, by reference, so the next call can follow straight after it with a dot. This style is called **method chaining**. Passing `const string &item` avoids copying the text and promises not to change it.

Static members

Normally each object has its own copy of every data member. A **static data member** is shared by all objects of the class: there is only one copy. A **static member function** belongs to the class rather than to an object, so you call it with the class name and `::`, the **scope resolution operator**.

Program 7: Counting admissions

CPP

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

class Admission {
    static int count;
    int rollNo;
public:
    Admission() { rollNo = ++count; }
    int roll() const { return rollNo; }
    static int total() { return count; }
};

int Admission::count = 0;

int main() {
    Admission a, b, c;
    cout << "c has roll number " << c.roll() << '\n';
    cout << "Total admissions: " << Admission::total() << '\n';
    return 0;
}
```

```
c has roll number 3
Total admissions: 3
```

Every new object adds 1 to the shared `count` and takes the new value as its roll number. The line `int Admission::count = 0;` outside the class creates and sets the single shared copy.

DID YOU KNOW

Since C++17 you can write `inline static int count = 0;` inside the class, and the separate line outside is no longer needed.

Friend functions

A **friend function** is not a member, but the class gives it permission to use private members. Use friends sparingly, because each one weakens encapsulation.

Program 8: Adding distances with a friend

CPP

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

class Distance {
    int km, m;
public:
    Distance(int k, int mt) : km(k), m(mt) {}
    friend Distance add(const Distance &a, const Distance &b);
    void show() const { cout << km << " km " << m << " m\n"; }
};

Distance add(const Distance &a, const Distance &b) {
    int total = (a.km + b.km) * 1000 + a.m + b.m;
    return Distance(total / 1000, total % 1000);
}

int main() {
    Distance home(3, 750), school(2, 600);
    add(home, school).show();
    return 0;
}
```

6 km 350 m

The class declares `add` as its friend, so `add` may read `km` and `m` although they are private. Converting everything to metres first ($3,750 + 2,600 = 6,350$) makes the carry from metres to kilometres automatic.

CHAPTER 5

Function and operator overloading

Overloading means giving one name several meanings. The compiler picks the right one from the arguments you pass. This is **compile-time polymorphism**: "polymorphism" means "many forms", and the choice is made while compiling.

Program 9: Function overloading

CPP

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

double area(double r) { return 3.14159 * r * r; }
double area(double l, double w) { return l * w; }
int area(int side) { return side * side; }

int main() {
    cout << "Circle: " << area(7.0) << '\n';
    cout << "Rectangle: " << area(4.0, 2.5) << '\n';
    cout << "Square: " << area(6) << '\n';
    return 0;
}
```

```
Circle: 153.938
Rectangle: 10
Square: 36
```

The three functions differ in the number or the types of their parameters, so each call matches exactly one. `area(7.0)` passes a double and gets the circle; `area(6)` passes an int and gets the square. Two functions that differ only in their return type are not allowed, because the call would look the same.

A **default argument** is another way to make one function serve several calls: `double interest(double p, double rate = 7.0)` can be called with one value or two. Default arguments go at the end of the parameter list.

Operator overloading

Operator overloading lets operators such as `+`, `==` and `<<` work with your own classes. You write a function named `operator+`, and `a + b` calls it. Use it only when the meaning is obvious, as with numbers, money or dates.

Program 10: Complex numbers

A **complex number** has a real part and an imaginary part, written like $3 + 2i$.

```

#include <iostream>
using namespace std;

class Complex {
    double re, im;
public:
    Complex(double r = 0, double i = 0) : re(r), im(i) {}
    Complex operator+(const Complex &o) const {
        return Complex(re + o.re, im + o.im);
    }
    Complex operator*(const Complex &o) const {
        return Complex(re * o.re - im * o.im, re * o.im + im * o.re);
    }
    friend ostream &operator<<(ostream &out, const Complex &c) {
        out << c.re << (c.im < 0 ? " - " : " + ")
            << (c.im < 0 ? -c.im : c.im) << "i";
        return out;
    }
};

int main() {
    Complex a(3, 2), b(1, -4);
    cout << "a + b = " << a + b << '\n';
    cout << "a * b = " << a * b << '\n';
    return 0;
}

```

```

a + b = 4 - 2i
a * b = 11 - 10i

```

- The constructor uses default arguments, so `Complex()`, `Complex(5)` and `Complex(3, 2)` all work.
- `operator+` adds real parts and imaginary parts separately. `operator*` uses the rule $(a + bi)(c + di) = (ac - bd) + (ad + bc)i$.
- `operator<<` cannot be a member of our class, because the left side of `cout << a` is `cout`, not our object. So it is declared a **friend**: it is written inside the class for convenience, but it is not a member, and being a friend lets it read the private parts. It returns the stream so that several `<<` can be chained.

CHAPTER 6

Inheritance and virtual functions

Inheritance lets a new class take over the members of an existing one and add its own. The existing class is the **base class** (parent) and the new one is the **derived class** (child). It models an "is a" relationship: a student is a person, a car is a vehicle.

Type	Shape	Example
Single	one base, one derived	Person → Student
Multilevel	a chain	Vehicle → Car → ElectricCar
Multiple	two or more bases	Exam + Sports → Result
Hierarchical	one base, many derived	Shape → Circle, Rectangle
Hybrid	a mix of the above	multiple and multilevel together

The access mode in `class Student : public Person` decides what the inherited members become:

Member in base	public inheritance	protected inheritance	private inheritance
public	public	protected	private
protected	protected	protected	private
private	not accessible	not accessible	not accessible

Public inheritance is used almost always, because it keeps the "is a" meaning.

Program 11: Single and multilevel inheritance

CPP

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

class Person {
protected:
    string name;
public:
    Person(const string &n) : name(n) { cout << "Person part built\n"; }
    void greet() const { cout << "Hello, I am " << name << '\n'; }
};

class Student : public Person {
    string course;
public:
    Student(const string &n, const string &c) : Person(n), course(c) {
        cout << "Student part built\n";
    }
    void info() const { cout << name << " studies " << course << '\n'; }
};

class Monitor : public Student {
public:
    Monitor(const string &n, const string &c) : Student(n, c) {
        cout << "Monitor part built\n";
    }
};

int main() {
    Monitor m("Arjun", "C++");
    m.greet();
    m.info();
    return 0;
}
```

```
Person part built
Student part built
Monitor part built
Hello, I am Arjun
Arjun studies C++
```

- `Student` inherits from `Person` (single inheritance) and `Monitor` from `Student` (multilevel), so a `Monitor` gets `greet()` and `info()` for free.
- Each constructor passes the name up with `: Person(n)` or `: Student(n, c)`. Because `name` is `protected`, `Student`'s own function `info()` can use it, while `main` still cannot.
- Constructors run from the top of the family down, just as a building rises from its foundation. Destructors run in the opposite order.

Program 12: Multiple inheritance for a student result

CPP

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

class Exam {
protected:
    int theory = 0, practical = 0;
public:
    void setMarks(int t, int p) { theory = t; practical = p; }
};

class Sports {
protected:
    int bonus = 0;
public:
    void setBonus(int b) { bonus = b; }
};

class Result : public Exam, public Sports {
    string name;
public:
    Result(const string &n) : name(n) {}
    void show() const {
        int total = theory + practical + bonus;
        cout << name << ": " << theory << " + " << practical
            << " + " << bonus << " bonus = " << total << '\n';
    }
};

int main() {
    Result r("Sana Khan");
    r.setMarks(62, 25);
    r.setBonus(5);
    r.show();
    return 0;
}
```

Sana Khan: 62 + 25 + 5 bonus = 92

`Result` inherits from two bases separated by a comma and can use members of both. The marks scheme is only an example.

DID YOU KNOW

If two bases both inherit from one common class, the derived class receives two copies of it. This is the **diamond problem**. Declaring the middle classes with `virtual public` inheritance keeps a single shared copy.

Early and late binding

A base-class pointer may point at a derived object: a `Shape *` can hold the address of a `Circle`. When you call a function through that pointer, which version runs?

- For a normal function, the compiler decides from the pointer's type. This is **early binding**.
- For a **virtual function**, the program decides while running, from the actual object. This is **late binding**, and it gives **runtime polymorphism**: one call, many behaviours.

Program 13: Virtual and non-virtual calls

CPP

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

class Base {
public:
    void hello() { cout << "Base hello\n"; }
    virtual void show() { cout << "Base show\n"; }
    virtual ~Base() { cout << "Base destroyed\n"; }
};

class Derived : public Base {
public:
    void hello() { cout << "Derived hello\n"; }
    void show() override { cout << "Derived show\n"; }
    ~Derived() override { cout << "Derived destroyed\n"; }
};

int main() {
    Base *p = new Derived;
    p->hello();
    p->show();
    delete p;
    return 0;
}
```

```
Base hello
Derived show
Derived destroyed
Base destroyed
```

`hello` is not virtual, so the pointer's type (`Base`) wins. `show` is virtual, so the object's real type (`Derived`) wins. `->` calls a member through a pointer. The word `override` asks the compiler to check that you really are replacing a virtual function; a spelling mistake then becomes an error instead of a silent bug.

REMEMBER

A base class with virtual functions needs a virtual destructor. Without it, `delete p` is unsafe: usually only `~Base()` runs, and the derived part is never cleaned up.

Abstract classes

A **pure virtual function** is declared with `= 0` and has no body in the base class. A class with at least one is an **abstract class**: you cannot create objects of it, only of derived classes that supply every missing function. It works like a contract: "every shape must be able to tell its area".

Program 14: Areas of shapes

CPP

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

class Shape {
public:
    virtual double area() const = 0;
    virtual string name() const = 0;
    virtual ~Shape() = default;
};

class Circle : public Shape {
    double r;
public:
    Circle(double radius) : r(radius) {}
    double area() const override { return 3.14159 * r * r; }
    string name() const override { return "Circle"; }
};

class Rect : public Shape {
    double l, w;
public:
    Rect(double len, double wid) : l(len), w(wid) {}
    double area() const override { return l * w; }
    string name() const override { return "Rectangle"; }
};

int main() {
    Circle c(2);
    Rect r(4, 3);
    Shape *shapes[] = {&c, &r};
    for (Shape *s : shapes)
        cout << s->name() << ": " << s->area() << '\n';
    return 0;
}
```

```
Circle: 12.5664
Rectangle: 12
```

The loop knows only about `Shape`, yet each object answers in its own way. To add a triangle later, you write one new class and the loop does not change at all. That is the real power of polymorphism. `Shape s;` would not compile, because `Shape` is abstract.

Templates and exception handling

Templates

A **template** lets you write a function or class once and use it with many types. The compiler makes a separate version for each type you use. This is called **generic programming**.

Program 15: A function template and a class template

CPP

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

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

template <typename T>
class Box {
    T item;
public:
    Box(T x) : item(x) {}
    T get() const { return item; }
};

int main() {
    cout << biggest(45, 72) << '\n';
    cout << biggest(3.5, 2.25) << '\n';
    cout << biggest(string("Agra"), string("Patna")) << '\n';
    Box<int> b1(500);
    Box<string> b2("Chai");
    cout << b1.get() << " " << b2.get() << '\n';
    return 0;
}
```

```
72
3.5
Patna
500 Chai
```

T is a placeholder for a type. In `biggest(45, 72)` the compiler sees two ints and makes an int version. For a class template, you name the type in angle brackets: `Box<int>`. We wrap the city names in `string(...)` because two bare quoted words would be compared as memory addresses, not alphabetically.

Exceptions

An **exception** is a way to report an error that the current function cannot handle. The function **throws** it, and the nearest matching **catch** block, after a **try** block, handles it. Code between the throw and the catch is skipped.

Program 16: Throwing and catching

CPP

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

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

class Wallet {
    double balance;
public:
    Wallet(double b) : balance(b) {}
    void pay(double amount) {
        if (amount > balance) throw runtime_error("not enough money");
        balance -= amount;
    }
    double left() const { return balance; }
};

int main() {
    try {
        cout << divide(10, 4) << '\n';
        cout << divide(5, 0) << '\n';
    } catch (const invalid_argument &e) {
        cout << "Error: " << e.what() << '\n';
    }
    Wallet w(300);
    try {
        w.pay(120);
        w.pay(250);
    } catch (const exception &e) {
        cout << "Error: " << e.what() << '\n';
    }
    cout << "Left: Rs " << w.left() << '\n';
    return 0;
}
```

```
2.5
Error: division by zero
Error: not enough money
Left: Rs 180
```

`invalid_argument` and `runtime_error` are standard exception classes from `stdexcept`, and `what()` returns their message. The second `catch` names `exception`, their common base class, so it catches both kinds. The failed payment of ! 250 changed nothing, because the throw happened before the subtraction.

TIP

Catch exceptions by `const` reference, as here. It avoids a copy and keeps the real type of the exception.

CHAPTER 8

The STL and file streams

The **Standard Template Library** (STL) is a set of ready-made, tested templates. Three parts are used constantly:

Part	What it is	Example use
Containers	objects that hold data	<code>vector</code> , <code>map</code>
Algorithms	functions that work on containers	<code>sort</code> , <code>find</code>
Iterators	markers that point into a container	<code>v.begin()</code> , <code>v.end()</code>

A **vector** is an array that can grow. It knows its own size and adds items with `push_back`.

Program 17: Marks in a vector, text in a string

CPP

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
using namespace std;

int main() {
    vector<int> marks = {72, 95, 58, 81, 66};
    marks.push_back(89);
    sort(marks.begin(), marks.end());
    int sum = 0;
    for (int m : marks) {
        cout << m << ' ';
        sum += m;
    }
    cout << "\nAverage: " << static_cast<double>(sum) / marks.size() << '\n';

    string place = "New Delhi";
    place += " Railway Station";
    cout << place << " has " << place.length() << " characters\n";
    cout << "Delhi starts at position " << place.find("Delhi") << '\n';
    cout << place.substr(4, 5) << '\n';
    return 0;
}
```

```
58 66 72 81 89 95
Average: 76.8333
New Delhi Railway Station has 25 characters
Delhi starts at position 4
Delhi
```

- `sort(marks.begin(), marks.end())` sorts the whole vector. `begin()` and `end()` are iterators marking the start and one past the end.
- `for (int m : marks)` is a **range-based for loop**: it visits every element in turn, with no index to get wrong.
- `static_cast<double>(sum)` converts `sum` so the division keeps decimals.
- A `string` joins with `+=`, knows its `length()`, finds text with `find()` (positions start at 0) and cuts a piece with `substr(start, count)`.

Program 18: Counting words with a map

A **map** stores pairs of a **key** and a **value**, kept sorted by key, like a dictionary of words and meanings.

```

#include <iostream>
#include <map>
#include <sstream>
#include <string>
using namespace std;

int main() {
    string order = "chai samosa chai pakora samosa chai";
    map<string, int> orders;
    istringstream words(order);
    string w;
    while (words >> w)
        orders[w]++;
    for (const auto &[item, n] : orders)
        cout << item << ": " << n << '\n';
    cout << "Idli ordered? " << (orders.count("idli") ? "yes" : "no") << '\n';
    return 0;
}

```

```

chai: 3
pakora: 1
samosa: 2
Idli ordered? no

```

`istringstream` lets us read words out of a string exactly as `cin` reads from the keyboard. `orders[w]++` creates the key with value 0 the first time a word appears, then adds 1. The loop prints in alphabetical order because a map keeps its keys sorted. `const auto &[item, n]` is a C++17 **structured binding**: it unpacks each pair into two names. `orders.count("idli")` checks whether a key exists without creating it.

TRY THIS

Change Program 18 so it reads an order typed by the user. Replace the fixed string with `getline(cin, order);`.

File streams

C++ reads and writes files with the same `<<` and `>>` you use for the screen. The `fstream` header gives three classes:

Class	Use
<code>ofstream</code>	write to a file (output file stream)
<code>ifstream</code>	read from a file (input file stream)
<code>fstream</code>	both read and write

Opening `ofstream out("file.txt");` creates the file or empties an existing one. To add to the end instead, open it with `ofstream out("file.txt", ios::app);`. A stream closes itself when the object is destroyed, thanks to its destructor, though you can call `close()` earlier.

Program 19: Saving and reading student marks

CPP

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

int main() {
    ofstream out("students.txt");
    if (!out) { cout << "Cannot create file\n"; return 1; }
    out << "Rahul 81\n" << "Divya 94\n" << "Imran 77\n";
    out.close();

    ifstream in("students.txt");
    string name;
    int marks, total = 0, count = 0;
    while (in >> name >> marks) {
        cout << name << " scored " << marks << '\n';
        total += marks;
        count++;
    }
    if (count > 0)
        cout << "Average: " << static_cast<double>(total) / count << '\n';
    return 0;
}
```

```
Rahul scored 81
Divya scored 94
Imran scored 77
Average: 84
```

`if (!out)` is true when the file could not be opened, for example in a folder you may not write to. `while (in >> name >> marks)` keeps going as long as both values are read, and stops cleanly at the end of the file. To read whole lines, including spaces, use `while (getline(in, line))`.

TRY THIS

Combine Programs 4 and 19: give `BankAccount` a `save()` function that writes its balance to a file, and a constructor that reads it back, so the balance survives after the program closes.

Practice questions

1. Give three differences between C and C++.
2. Write the `g++` command to compile `shop.cpp` with the C++17 standard and warnings turned on.
3. Why does `cin >> name` read only "Gel" when the user types "Gel pen"? How do you read the whole line?
4. What is a reference, and how does it make a swap function simpler than in C?
5. What is the difference between a class and an object?
6. Explain the three access specifiers in one line each.
7. What is encapsulation, and how does Program 4 show it?
8. Name the three kinds of constructor and say when each runs.
9. In what order are objects destroyed at the end of `main`?
10. Give two uses of the `this` pointer.
11. How is a static data member different from an ordinary one?
12. What is a friend function, and why should it be used sparingly?
13. Can two overloaded functions differ only in their return type? Why or why not?
14. Why is `operator<<` for a class written as a friend instead of a member function?
15. Name the five types of inheritance.
16. In which order do constructors run in multilevel inheritance?
17. What is the difference between early binding and late binding?
18. Why does a base class with virtual functions need a virtual destructor?
19. What is an abstract class, and can you create an object of one?
20. Why does Program 15 wrap the city names in `string(...)`?
21. What happens to the code between a `throw` and its matching `catch`?
22. How is a `vector` better than an ordinary array?
23. How do you open a file so that new data is added to its end?

Answers

1. C++ adds classes, references, exceptions, templates and the STL; uses `cin` and `cout` for input and output; and has a `string` class.
2. `g++ -std=c++17 -Wall shop.cpp -o shop`
3. `>>` stops reading at a space. Use `getline(cin, name);` to read the whole line.
4. A second name for an existing variable. Reference parameters change the caller's variables directly, with no `&` at the call or `*` inside.
5. A class is a design for a type; an object is one real instance made from it. One class can make many objects.
6. `public`: anywhere; `private`: only inside the class (and by friends); `protected`: the class and its derived classes.
7. Keeping data private and reachable only through public functions. In Program 4, only `deposit` and `withdraw`, which check their input, can change `balance`.
8. Default: when no values are given; parameterised: when values are given; copy: when an object is made as a copy of another.
9. In reverse order of creation: the last object created is destroyed first.
10. To tell a member apart from a parameter with the same name (`this->qty = qty`), and to return the object itself (`return *this`) for method chaining.
11. A static member has one copy shared by all objects; an ordinary one has a copy in each object.
12. A non-member function allowed to use private members. Each one weakens encapsulation.
13. No. A call does not show which return type is wanted, so the compiler could not choose between them.
14. In `cout << a` the left side is `cout`, not our object, so it cannot be our member; as a friend it can still read the private data.
15. Single, multilevel, multiple, hierarchical and hybrid.
16. From the topmost base class down to the most derived class. Destructors run in the reverse order.
17. Early binding: the compiler chooses from the pointer's type. Late binding, for virtual functions: the program chooses at run time from the actual object.
18. So that deleting a derived object through a base pointer runs the derived destructor too. Otherwise the derived part is never cleaned up.
19. A class with at least one pure virtual function (`= 0`). No: only derived classes that define them all can make objects.
20. Two bare quoted words are character arrays, so `>` would compare their memory addresses. As `string` objects they are compared alphabetically.
21. It is skipped. Control jumps straight to the matching `catch` block.
22. A vector can grow with `push_back`, knows its own `size()`, and works with STL algorithms such as `sort`.
23. Open it with `ios::app`, for example `ofstream out("log.txt", ios::app);`.

Glossary

Abstract class

a class with at least one pure virtual function, used only as a base.

Access specifier

the keyword `public`, `private` or `protected`, which decides who may use a member.

Base class

the class another class inherits from.

Class

a user-made type joining data and functions.

Constructor

a member function that sets up a new object.

Derived class

a class that inherits members from a base class.

Destructor

`~ClassName`, run when an object is destroyed.

Encapsulation

keeping data private behind public functions.

Exception

an object thrown to report an error, handled by a matching catch block.

Friend function

a non-member allowed to use private members.

Inheritance

building a new class from an existing one.

Iterator

an object marking a position in a container.

Member initialiser list

the part after a constructor's colon that sets members as the object is created.

Namespace

a named group of names, such as `std`.

Object

one real instance of a class.

Operator overloading

making `+` and others work with your class.

Polymorphism

one name or call behaving in different ways for different types.

Pure virtual function

a virtual function declared with `= 0` and no body.

Reference

a second name for an existing variable.

STL

the Standard Template Library of ready-made containers, algorithms and iterators.

Template

a class or function written for many types.

this

a pointer, inside a member function, to the object the function was called on.

Virtual function

a member function whose version is chosen at run time from the actual object.



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.