The -> operator

```
Dog* d = new Dog("Rex");
d-> bark(); // shorthand
(*d).b ark(); // same thing
```

-> operator — shorthand for dereferencing a pointer and accessing a member. Equivalent to (*pointer).member.

Building a class with a header file

```
// Foo.h
struct Foo {
        <Ty pe> <at tri but eNa me>;
        static <Re tur nTy pe> <fu nct ion Nam e>( <Ty pe> <va ria ble >);
        static <Re tur nTy pe> <fu nct ion Nam e>( <Ty pe> <va ria ble >);
        static <Re tur nTy pe> <fu nct ion Nam e>( <Ty pe> <va ria ble >);
};
// Foo.cpp
#include " Foo.h"
<Re tur nTy pe> Foo::< fun cti onN ame >(< Typ e> <va ria ble >)
{
       // implem ent ation
}
<Re tur nTy pe> Foo::< fun cti onN ame >(< Typ e> <va ria ble >)
{
       // implem ent ation
}
<Re tur nTy pe> Foo::< fun cti onN ame >(< Typ e> <va ria ble >)
{
       // implem ent ation
}
```

enum class

```
enum class <EnumName> { <Val1>, <Val2>, <Val3> };
<En umN ame> <va r> = <En umN ame >:: <Va l_n >; // where n is an Val in the defini tion.
```

You can think of these as a variable, but ahead of time it is constrained to a specific set of values.

Definition of a Dangling Pointer

```
int* createNumber() {
       int x = 42; // lives on the stack
        return &x; // return its address
} // x is destroyed here
int main() {
       int* p = create Num ber();
       *p; // ✕ dangling — x is gone, memory no longer belongs to you
}
```

Dangling pointer — a pointer that points to memory that no longer belongs to you. The object it pointed to has been destroyed, but the pointer still holds its old address.

Address of Operator

```
int x = 42;
int* ptr = &x; // ptr holds the memory address of x
```

Definition of a Caller

```
int add(int a, int b) { // callee
        return a + b;
}
int main() { // caller
        add(1, 2);
}
```

Caller — the function that calls another function.

static_cast

```
targetType result = static_cast<targetType>(sourceValue);
```

A C++ operator that explicitly converts a value from one type to another at compile time, making the conversion visible and intentional in the code. It is safer than a C-style cast because the compiler checks that the conversion is valid. It is used when you need to convert between related types such as int and float, or long and int.

Direct versus copy initialization of a constructor

```
T b(a); // direct initialization
T b = a; // copy initia liz ation
```

Direct initialization T b(a) and copy initialization T b = a are two syntax styles that both invoke the copy constructor to create a new object from an existin…Direct initialization T b(a) and copy initialization T b = a are two syntax styles that both invoke the copy constructor to create a new object from an existing one. They produce the same result — a new object constructed as a copy of a.

Operator=

```
T& operator=(const T& other) {
        return *this;
}
```

operator= defines what happens when you use = on an object.operator= defines what happens when you use = on an object. It takes a reference to another object of the same type and returns a reference to itself via return *this, enabling chained assignments like c = b = a.

Default parameter values

```
// In C++, function parameters can be given default values, meaning the caller does not need to provide
them
// explic itly. Default parameters must appear at the end of the parameter list, so no non-de faulted
parameter
// can follow a defaulted one. If the caller omits a defaulted argument, the default value is used
// automa tic ally; otherwise the caller's value takes preced ence.
void foo(int a, int b, int c = 10); // c defaults to 10
void bar(int a, int b = 5, int c = 10); // b defaults to 5, c to 10
// void baz(int a = 1, int b, int c); // ERROR: non-de fault after default
// Valid calls:
foo(1, 2); // c = 10
foo(1, 2, 3); // c = 3
bar(1); // b = 5, c = 10
bar(1, 2); // b = 2, c = 10
bar(1, 2, 3); // b = 2, c = 3
```

static

```
static variableName = <value>;
```

Signals that a value belongs to the class, not any instance, and is fixed at compile time.

const placed after a member function

```
class Object {
       int m_value = 10;
public:
       int getValue() const { return m_value; } // const: can only read
       void setVal ue(int v) { m_value = v; } // non-const: can modify
};
```

A const placed after a method signature means the function is read-only — it cannot modify any**member variables** of the object. It acts as a guardrail, forcing the compiler to catch accidental modifications inside the function. Use it on any method that only needs to read state, such as getters, to make your intent clear and your code safer.

Definition of a Non-Static Member Function

```
class Dog {
public:
        std ::s tring name;
       void bark() {
                std ::cout << name << " says woof! \n";
       }
};
int main() {
       Dog d;
        d.name = " Rex ";
        d.b ark(); // prints "Rex says woof!"
}
```

A non-static member function is a function that belongs to a specific instance of a class, and always operates on that instance through this.

Passing Functions (Callbacks)

```
#include <iostream>
void sayHello()
{
        std ::cout << " Hel lo! " << std::endl;
}
void execut e(void (*call bac k)())
{
        cal lba ck();
}
int main()
{
        exe cut e(&sa yHe llo); // pass the function, not call it
}
```

In C++, functions can be passed as parameters to other functions using function pointers, allowing you to decide what code runs without hardcoding it. The receiving function holds onto the address of the passed function and calls it at the right time. This is the foundation of event--driven programming, where you say "call this function when something happens"

Defnition of a Thread

```
#include <thread>
#include <io str eam>
void doWork() {
        std ::cout << " thread runnin g\n ";
}
int main() {
        std ::t hread t(doWork); // thread starts here
        t.j oin(); // main waits for it to finish
}
```

An independent sequence of execution within a program. The OS can run multiple threads concurrently, each doing their own work, while sharing the same memory.

L values (Left Values)

```
int x = 10; // x is an lvalue
x = 20; // valid: lvalue on the left of =
int y = x + 1; // (x + 1) is an rvalue — temporary, no fixed address
int z = 42; // 42 is an rvalue literal
// &(x + 1); // ERROR: cannot take address of an rvalue
```

An lvalue (locator value) is an expression that refers to a memory location and can appear on the left-hand side of an assignment. It represents an object that persists beyond a single expression — it has an identifiable address in memory.

Definition of a Static Method

```
class MathHelper {
public:
        static int add(int a, int b) {
                return a + b;
       }
};
int result = MathHe lpe r:: add(3, 5); // 8
```

A static method is a function that belongs to the class itself rather than to any instance of it. It has no access to instance attributes and can be called without ever creating an object.

Definition of a Signature

```
#include <iostream>
void greet( std ::s tring name) { // signature: greet( std ::s tring)
        std ::cout << " Hello, " << name << " !\n ";
}
int main() {
        gre et( " Ali ce"); // → Hello, Alice!
}
```

A function signature in C++ is the part of a function declaration that identifies it uniquely to the compiler. It consists of:

Function name
Parameter types (number, order, and types)

Reference declaration (in type declarations)

```
int x = 42;
int& ref = x; // ref IS x — same memory location
ref = 100; // x is now 100
```

bitwise OR assignment operator

```
|=
// Example
bool result = false; // 0
result |= false; // 0 | 0 = 0
result |= false; // 0 | 0 = 0
result |= true; // 0 | 1 = 1
result |= false; // 1 | 0 = 1 — stays true
result |= false; // 1 | 0 = 1 — stays true
// result is true
```

Bitwise OR assignment (|=) — takes the existing bits of the left operand, ORs them column by column with the bits of the right operand, and stores the result b…Bitwise OR assignment (|=) — takes the existing bits of the left operand, ORs them column by column with the bits of the right operand, and stores the result back into the left operand. Can only turn bits on, never off.

Constant Member Functions

```
class Foo {
       int m_value = 0;
public:
       int getValue() const { return m_value; } // read → const
       void setVal ue(int n) { m_value = n; } // write → no const
};
```

Guarantees the method will not modify any member variables.

Macros

```
// Creation
#define OPERAT ION (input) input * multiplier
// Usage
int result = OPERAT ION (va lue); // expands to: int result = value * multip lier;
```

A preprocessor directive defined with #define that is expanded before the compiler sees the code. It is not a function or a class, and can be used for text substitution, constants, or conditional compilation. Macros are considered old fashioned in modern C++ but are still used in cases where compile time behaviour is needed.

The Two Main Ways of Initializing a Function

```
// 1. Default then assign
std::t hread t;
t = std::t hre ad( foo);
// 2. Declare and initialise in one line
std::t hread t(foo);
```

Passing This to Other Objects

```
#include <iostream>
class B {
public:
       void hello( class A* a);
};
class A {
public:
       int number = 42;
       void send() {
               B b;
                b.h ell o(t his); // passing this to B
       }
};
void B::hel lo(A* a) {
        std ::cout << a-> number << " \n"; // prints 42
}
int main() {
       A a;
        a.s end();
}
```

Anytime a class the keyword "this" is implicit and automatic. Once a class has to pass data from the class into another object the "this" keyword has to be passed.

Endless For-Loop

```
for (;;)
```

Lambda Functions

```
[&<OBJECT>] {
        <CO DE>;
        <CO DE>;
}
```

The brackets tell the lambda which variables from the surrounding scope it's allowed to use, and how it remembers them.

By default, a lambda body cannot see any local variable from the function it's written inside. The capture list is how you grant access, one variable (or one default rule) at a time.

Constructing an obect in C++

```
TypeName variableName(constructorArguments);
```

This is how it would look in Python:
variable_name = ClassName(constructor_arguments)

Dereferencing a Pointer

```
int x = 42;
int* p = &x; // p holds the address of x
*p; // derefe rence — follow the pointer to get 42
```

The * in front of a pointer means "go to the address this pointer holds and give me what's there."

Range Based For-Loops

```
for (const <type>& <element> : <collection>) {
       // use <el eme nt>
}
```

A range-based for loop is a loop that iterates over every element in a collection, from the first to the last, without needing to manually manage an index or i…A range-based for loop is a loop that iterates over every element in a collection, from the first to the last, without needing to manually manage an index or iterator.

const — omit if you need to modify <element>
& — omit if you want a copy of each element rather than a reference

Template Functions

```
template <typename T>
T add(T a, T b) {
        return a + b; // T must support the + operator
}
```

Try Catch

```
try {
       // code that might throw an exception
} catch (const std::e xce pti on& e) {
       // handle exception
}
```

catch(...). This is for any unkown exception that can't be caught by an std::exception.

Template Aliases

```
// The template
template <ty pename T>
class Box {
       T contents;
};
// Without an alias — verbose
Box<in t> myBox;
// The alias
using IntBox = Box<in t>;
// With the alias — cleaner, same thing
IntBox myBox;
```

=delete

```
class T {
        T(const T&) = delete; // copying forbidden
};
T a;
T b(a); // compiler error
```

= delete explicitly forbids a function from being used, turning a potential runtime crash into a compile time error.= delete explicitly forbids a function from being used, turning a potential runtime crash into a compile time error.

constexpr

constexpr <datatype> variableName = <value>;

constexpr declares a value that is fixed and known at compile time, meaning the compiler bakes it directly into the binary rather than storing it in memory. This eliminates any runtime cost of looking up the value, since the compiler simply substitutes it wherever it appears in the code. It also makes intent explicit — clearly signaling to other developers that this value is a compile time constant that will never change.

Factory Method

```
class MyClass {
public:
        static MyClass fromIn put (st d:: string input) {
                MyClass object;
                obj ect.m_ value = std::s toi (in put);
                return object;
       }
private:
       int m_value;
};
MyClass result = MyClas s:: fro mIn put (in put);
```

A static method that constructs and returns an object, hiding the construction details from the caller.

Member Initializer List

```
class Car {
        std ::s tring m_brand;
       int m_year;
public:
        Car (st d:: string brand, int year)
               : m_bran d(b rand) // initialize m_brand with brand
               , m_year (year) // initialize m_year with year
       {}
};
```

A way to set a class's member variables before the constructor body runs. Uses a colon after the constructor parameters, with each member and its starting value in parentheses. std::move can be used instead of copying when transferring ownership of data.

What goes in the (): The value you want the member to start with. This is usually the matching constructor parameter, but can also be a literal value like 0 or "default".

Commenting Out Unused Parameters in C++

```
// You must accept both parameters to match the signature
void proces s(int /*unused*/, int used)
{
        std ::cout << used; // unused not needed, but must be in signature
}
```

When a function signature is fixed and you can't change it, but don't need all the parameters, comment out the name to suppress unused warnings while keeping the type.