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
}
```

Building a class with a header file (cont)

```
> <ReturnType> Foo::<functionName>(<-
Type> <variable>)
{
   // implementation
}
<ReturnType> Foo::<functionName>(<Typ-
e> <variable>)
{
   // implementation
}
```

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();
```

Definition of a Dangling Pointer (cont)

```
>     *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.

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"

namespace alias

```
namespace short_name =
some::long::namespace;
// Instead of:
some:: lon g:: nam esp -
ace ::type myVar;
// You can write:
short_ nam e::type myVar;
```

Creates a shorthand alias for a long namespace to make code less verbose.

The alias can be used anywhere in the file in place of the full namespace name.

Commonly used for long standard library namespaces like std::filesystem.

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.

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_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.

Access Modifiers

```
class my_class {
public:
       int public _va riable; //
accessible from anywhere
       void public _fu nct -
ion(); // accessible from
anywhere
protected:
       int protec ted _va -
riable; // accessible by this
class and children
       void protec ted _fu nct -
ion(); // accessible by this
class and children
private:
       int privat e_v ari able;
// only accessible within this
class
       void privat e_f unc -
tion(); // only accessible
within this class
};
```

Controls who can access members (variables and functions) of a class.

public — accessible from anywhere.

protected — accessible only within the class and any class that inherits from it.

private — accessible only within the class itself, not even by children.

constexpr

```
constexpr int max_size = 100; //
evaluated at compile time
constexpr const char* my_string
= " hel lo"; // evaluated at
compile time
const int runtim e_size =
get_si ze(); // OK — const can
be runtime
constexpr int bad = get_si ze();
// error — constexpr must be
compile time
```

Declares that a value is constant and must be known at compile time.

Stronger than const — the compiler evaluates it before the program runs.

Allows the compiler to make deeper optimisations than const alone.

override

```
class parent_class {
public:
        virtual void my_fun -
ction() { }
};
class child_ class : public
parent _class {
public:
       void my_fun ction()
override { } // compiler
confirms this matches parent
       void my_fun cton()
override { } // compiler error —
typo caught
```

override (cont)

```
>     void my_functon() { } // no error —
silently creates new function
};
```

Tells the compiler you are intentionally overriding a virtual function from a parent class.

If the function signature doesn't match the parent's virtual function exactly, the compiler will error.

Without it, a typo or signature mismatch would silently create a new function instead of overriding the parent's.

Integer Literal Suffixes

```
1u // unsigned int
1l // long
1ul // unsigned long
1ll // long long
1ull // unsigned long long
```

Suffixes added to integer literals to specify their type explicitly.

Used to avoid signed/unsigned mismatch warnings when comparing with typed values.

Common suffixes are u for unsigned, l for long, and ul for unsigned long.

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 assign-ment. 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
```

Endless For-Loop

```
for (;;)
```

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.

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.

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);
```

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.

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.

For Loop with Internal Cursor

```
// reading lines from a file
for (std:: string line; std::g -
etl ine (fi le_ stream, line);)
{
       // process line
}
// reading words from a file
for (std:: string word; file_s -
tream >> word;) {
       // process word
}
// reading integers from a file
for (int number; file_s tream >>
number;) {
       // process number
}
```

Used when the object being read maintains its own internal cursor that advances automatically with each read.

The condition of the for loop both reads the next item AND advances the cursor in one step, so no increment is needed.

Only works with stream objects like std::ifstream, std::cin etc. that consume data sequentially.

Global Namespace ::

```
::testing::Test // explicitly
uses global testing::Test
testin g::Test // uses testin -
g::Test from current namespace
first
::std: :string // explicitly
global std::s tring
std::s tring // std::s tring
from current namespace first
```

A leading :: tells the compiler to start looking from the global namespace rather than the current one.

Used to avoid ambiguity when a name might exist in both the current namespace and the global namespace.

Without it the compiler searches the current namespace first, which could resolve to the wrong type.

const

```
const int max_size = 100; //
cannot be changed
const char* my_string = " hel -
lo"; // string cannot be
modified
max_size = 200; // error —
cannot modify const
```

Declares that a variable's value cannot be modified after initialisation.

Can be applied to variables, pointers, and function parameters.

Enforced at runtime.

char*

```
const char* my_string = "hello";
// h e l l o \0
// [0][1] [2] [3] [4][5]
my_string + 1; // points to [1],
reads " ell o"
my_string + 2; // points to [2],
reads " llo "
```

A pointer to a character, typically used to point to the start of a string.

Strings in C are just arrays of characters ending with a null terminator \0 — reading a string means reading every character from the pointer until \0 is reached.

Adding a number to the pointer moves it forward by that many characters, changing where reading begins.

Inheritance

```
class child_class : public
parent_class {
       // child_ class now has
access to all public members of
parent _class
};
```

Allows a class to inherit members and behaviour from a parent class using :.

public inheritance means the parent's public members stay public in the child.

private inheritance means the parent's public members become private in the child.

Use :: before the parent class name to explicitly reference the global namespace.

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 <eleme-nt>
& — 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;
```

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.

=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.

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".

static

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

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

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.

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.

R"delimiter(...)delimiter"

```
// Regular string — special characters need escaping
const char* regular = " lin e1 - \nli ne2 \t \" quo ted \"";
// Raw string — no escaping needed, written exactly as it should appear
const char* raw = R"de lim iter(
line1
line2
" quo ted "
$variable
)delim ite r";
```

Used when a string contains special characters like $, ", \, or newlines that would need escaping in a regular string.

Everything between the delimiters is treated as literal text — no escape sequences are processed.

The delimiter can be any text but must be unique enough to not appear inside the string content, and is often chosen to hint at the content type (e.g. sh for shell scripts, html for HTML).

Virtual Function

```
class parent_class {
public:
        virtual void my_fun - ction() {
                // base implem - ent ation
        }
};
class child_ class : public parent _class {
public:
        void my_fun ction() override {
                // child's own implem ent ation, replaces parent's
        }
};
```

A function in a base class that is designed to be overridden by child classes.

The override keyword confirms to the compiler that you are intentionally overriding a parent function.

Without virtual in the parent, the child's version would not be called polymorphically.

std::function

```
std::function<return_type(argument_type)> my_function;
// storing a regular function
std::f unc tio n<i nt(int, int)> add = [](int a, int b) { return a + b; };
// storing a lambda
```

std::function (cont)

```
> std::function<void(std::string)> printer = [](std::string s) {
    std::cout << s;
};
// calling it
add(2, 3); // returns 5
printer("hello"); // prints "hello"
```

A general purpose wrapper that can hold any callable — a function, lambda, or function object.

The template parameter describes the function signature: return type and argument types.

Useful for storing and passing functions as variables.