## 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 -
>)
{
>    // implementation
}
```

## Building a class with a header file (cont)

## 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.

## 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
```

Claude responded: 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.

## 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
}
```

## 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;
```

## 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.