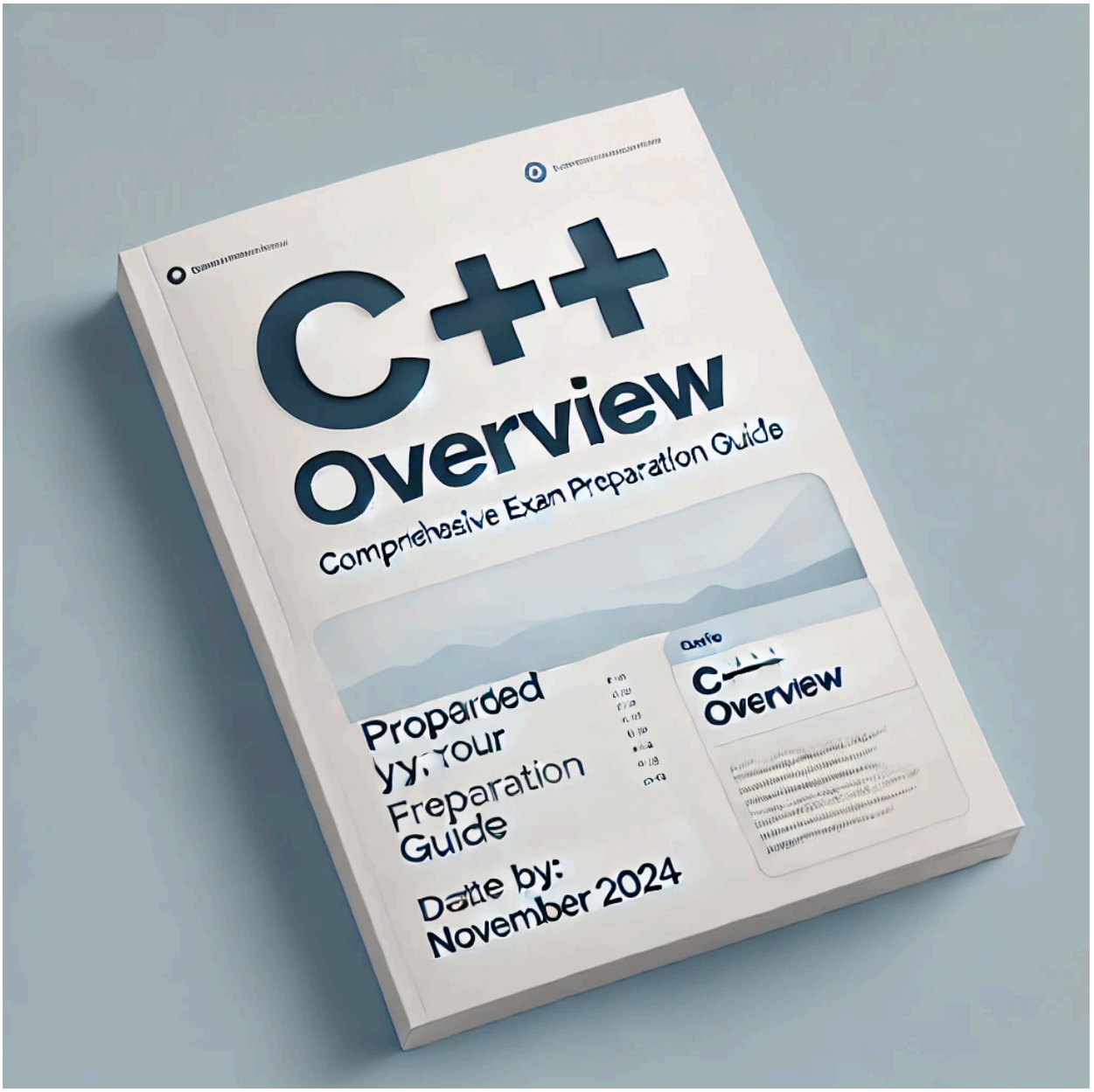


C++ Overview Guide

First * then () then []

Resolved in this order

- **How to differentiate if the "&" is used to represent a reference or an address?**

  The reference operator is always positioned to the left of the assignment operator "=" while the address operator is always to the right of it.

- **Function pointers**: is a pointer that points to the memory address of a function.

  Syntax: type(* funcptr)(list of parameters);

  Ex: void(*pTest)();

Summarized from the right (*pv++ == *(pv++))
array[i] == *(array + i)

Function pointer – Points to a function and can be used to run a function
type (* funcptr) (parameters);

```
int (*op) (int,int);
op = add; //add is a function which sums two numbers
cout << op(3,4) << endll; // sums 3 and 4 and prints the result
```

!!! Function pointers point to code not data

Also function pointers don't have a state which means they can only get arguments but they are stateless unless they access global variables which is a very bad design.

Function object (functor) – Any object for which the function call operator is defined -> operator()

Lambda expression:

https://www.modernescpp.com/index.php/c-core-guidelines-function-objects-and-lambas

- std::unique_ptr – owns a dynamically allocated resource; no other smart pointers can point to it

- std::shared_ptr – owns a shared dynamically allocated resource; several shared_ptrs may own the same resource (internal counter controls the references)

- std::weak_ptr – like shared_ptr, but without reference counting

unique_ptr
- owns the object it points to
- no other smart pointers can point to it
- when it goes out of scope the object is deleted

shared_ptr
- owns the object it points to
- allows for multiple references
- **reference counting** – when a shared_ptr pointing to the same resource goes out of scope an internal counter decreases -> when counter reaches 0 the data gets deallocated

weak_ptr
- similar to shared_ptr
- NO reference counting
- Holds a non-owning (or weak) reference to an object which is managed by another shared_ptr
- lock() method converts it to a shared_ptr

Smart pointers in functions

unique_ptr
- can't be passed by value because it can't be copied -> std::move should be used instead
- pass a non-const reference because otherwise the function cannot do anything with it

shared_ptr
- can be passed by value
- const reference is possible if the function will only read from it, otherwise similar to unique_ptr

weak_ptr
- can be passed by value
- no use case to pass a reference

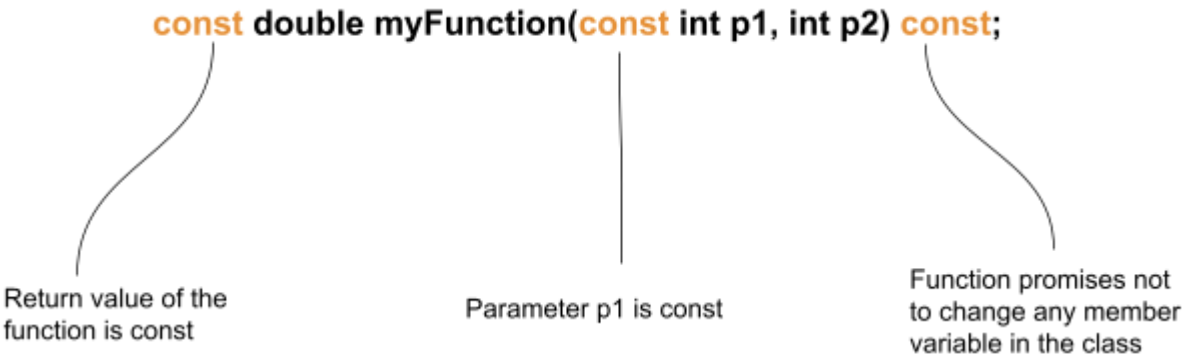
Use of Const:

Rule:

Const	*	Const
If const is before the * it means that **value of what the pointer is pointing to** is const Ex: int const* a; Or const int* a;		If const is after * it means that the **value of the pointer itself** is const Ex: int* const a;

NOTE: A const pointer (int* const a) MUST be initialized to a value upon declaration.

Const in functions:



Note: When a function is declared as const, it can be called on any type of object. Non-const functions can only be called by non-const objects.

Const correctness:

Const correctness is a form of type safety for mutable/immutable variables in c++. Especially in function declarations. Let's look at some examples:

```
13  int function1(int& e1) {
14      cout << e1 << endl;
15      return e1;
16  }
```

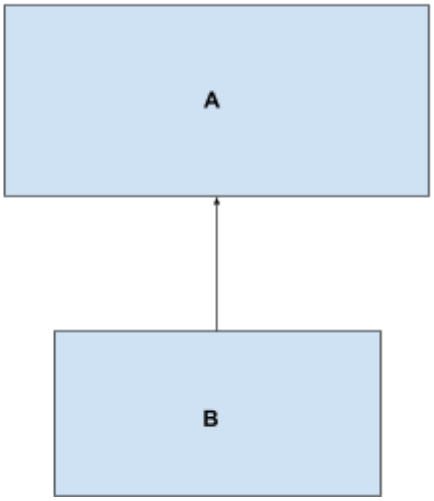
Substitution principle:

- Class B id derived from class A.

```
Class A {
A ObjA;
B ObjB;
void test(A a);
};

ObjA = ObjB; //ObjB is IMPLICITLY CONVERTED TO ObjA.

test(Obj B); //Since the method test takes only ObjA slicing will occurs and the copy constructor of class A will be called.
```



We can prohibit implicit conversion using the keyword explicit.
Example: explicit Clock(const double hour); //Converting constructor.
Clock c1;
C1 = 8.75; //syntax error.

Down-cast (Base class is cast to derived class) **IS POSSIBLE BUT DANGEROUS.**

Class B is derived from class A

- Object B can be assigned in all cases where object A is expected
- Object B is implicitly converted to an object A (**slicing**)

Composed objects:

- M is a part of C
- M can only belong to one C at a time
- M has its existence managed by C
- M does NOT know about the existence of C

Operator overloading:

Non-overloadable operators:

- :: => scope operator
- . => selection of member elements
- .* =>selection of member pointer element
- Sizeof => memory requirement
- ?: => conditional expression

Operators that should be overloaded as non-static members (because the first operand must be a left value)

= , [] , () , ->

Operator can only be overloaded if there is at least one user-defined type involved.

X operator++(); –> overloads the prefix increment ++obj
Y operator++(int); –> overloads the suffix increment obj++ (by using a dummy parameter)

Friend classes and functions
Marking as 'friend' allows them to access private elements and functions.
The class itself determines who is a friend.

Should only be used if the data accesses otherwise would become very expensive

```
class A {
        friend void someGlobalFunc(A * ptr);
}
```

Templates:

The whole code must be implemented in header file.

When we write any template based function or class, compiler creates a copy of that function/class whenever compiler sees that being used for a new data type or new set of data types(in case of multiple template arguments).

Specifying the data type when using a template (meaning in main function) determines which instance of the template can be used for.

Compiler Behavior

- Most modern compilers (e.g., GCC, Clang, MSVC) implement aggressive copy elision even in pre-C++17 code, as long as the result is equivalent to the standard's requirements.

Ranges:

For this topic it is simpler to understand if you read the following documentation:

https://learn.microsoft.com/en-us/cpp/standard-library/ranges?view=msvc-170

String functions:

Function	**Description**	**Return type**
.length()	Returns length of a string	int
.empty()	Returns true if string is empty	bool
.clear()	Clears a string	void
.append()	Appends a string to the end	void
.at(**X**)	Returns a character at a given index **X**	char
.substr(int x, int y)	Retrieve a portion of a string	string
s.insert(int **x**, string **m**)	Insert a string **m** at a give position **x** string **s**	void
.find(string n)	Returns the position (index) of n in a string	int
.erase(int x, int y)	Deletes a portion of a string	void