std::move

```
// Basic Usage
std::m ove (<o bje ct>);
------ --- --- --- --- --- --
- --- --- --- --- -----
// Not using move
MyObject a;
MyObject b = a; // a and b both
exist, a is unchanged
// With move — transfers
ownership (cheap, always works)
MyObject a;
MyObject b = std::m ove(a); // b
has everyt hing, a is now empty
// With a caller
MyObject buildT hing()
{
        MyO bject thing;
        return std::m ove (th -
ing); // transfers ownership to
the caller
}
MyObject result = buildT hing();
// result now owns everything
```

1. Transfers ownership of an object to another location, leaving the original empty. It prevents expensive or impossible copies by moving the underlying data instead.

std::error_code

```
std::error_code ec;
```

std::error_code is a lightweight C++ object for holding an error without throwing an exception.

std::make_unique

```
auto myObject =
std::make_unique<ClassName>
(arg1, arg2, ...);
```

You get back a std::unique_ptr<ClassNam-e> that automatically deletes the object when it goes out of scope.

<ClassName> — the type you want to allocate
<arg1, arg2, ...> — the arguments forwarded to its constructor (can be zero or more)

Note: Scope is defined as where it was originally declared.

std::memory_order_acquire

```
std::memory_order_acquire
```

memory_order_acquire is a constant within the memory_order enum. When a thread loads an atomic value with acquire, it is guaranteed to see all writes that the other thread made before its corresponding release store.

std::exception

```
try
{
       // code that might throw
}
catch (const std::e xce pti on&
e)
{
        std ::cout << e.what();
// returns a descri ption of the
error
}
```

std::exception is the base class for all standard C++ exceptions, caught using a const reference.

Give me one sentence on the method what

std::nullopt

```
// std::nullopt
// A special constant from <op -
tio nal >.
// Represents "no value" for a
std::o pti onal.
// Use it to clear an optional
or signal that a value is
absent.
#include <op tio nal>
std::o pti ona l<i nt> age = 25;
// has a value
age = std::n ullopt; // now has
no value
```

std::unique_ptr

```
std::unique_ptr<T>;
.reset() // If the pointer points to an object it will destroy it and set it back to being a null pointer.
```

Declares a pointer of a certain type. It starts as a null pointer.

std::lock_gaurd

```
std::lock_guard<MutexType> objectNam-e(mutexInstance);
```

A wrapper that takes a mutexInstance and ensures that is is locked and automatically unlocked when it goes out of scope.

std:string

```
std::string name = "<text>";
.clear() // Sets the string back to " ".
.c_str() // Converts a std::s -tring to a const char* (C-style string).
```

Make a string object that is dynamic and has methods associated with it. This is different than a primitive string.

std::optional<T>

```
std::optional<int> val = std::nullopt; // val holds nothing
val = 42; // val now holds 42
```

A wrapper that holds either a value of type T or nothing (std::nullopt). Use it as a return type when a function might produce no result, instead of returning a sentinel like -1 or nullptr.

std::thread

```
std::thread t(callable, arg1, arg2, arg3, ...);
// Methods
.joina ble() // Returns a bool determ ining if a thread can eventually be joined
.join() // Block the calling thread until this thread completes
```

std::queue

```
std::queue<dataType>
// Returns the first item in the queue
.front()
// Removes the first item from the queue
.pop()
```

Creates a queue data structure for holding information

std::unique_lock

```
// std::uniquelock is a scoped mutex wrapper that
// automa tically unlocks on destru ction.
// std::u niq ue_lock is a scoped mutex wrapper that automa tically unlocks on destru ction.
// Unlike lock_g uard, it can unlock and re-lock mid-scope, which is required by condit -ion _va ria ble ::w ait().
std::u niq ue_ loc k<s td: :mu -tex> lock(mtx);
cv.wai t(lock, [this] { return ready; });
```

std::getenv

```
const char* path = std::getenv("PATH");
if (path) {
        std ::s tring path_s -tring = path; // safe to use
}
// or with a fallback
const char* value = std::g ete -nv( " MY_ VAR ");
std::s tring result = value ? value : " "; // empty string if not set
```

Reads the value of an environment variable by name.

Returns a const char* to the value string, or nullptr if the variable doesn't exist.

Always check for nullptr before using the result to avoid crashes.

std::memory_order_relaxed

std::memory_order_relaxed

A value of a enum that signals that the compiler and CPU that it is allowed to reorder an operation relative to when the code was written in the most optimized way it thinks it should.

std::memory_order_released

```
std:memory_order_release
```

This is an memory_order enum flag that is used to suggest an atomic object can only perform an operation if every line above it has already ran.

std:condition_variable

```
std:condition_variable
<variableName>;
// Method to force a thread to
wait. If false is returned then
the thread waits.
.wait( lock, [this] { return
someCo ndi tion; });
// Method to wake up all
sleeping threads
.notif y_all()
// Just notifies one thread.
// The one chosen can't be
determined ahead of time.
// They are usually all lined up
to do the same thing.
.notif y_one()
```

This will force the thread to sleep that was holding a unique lock. It is only woken up once .notify_all() or .notify_one() is called. These commands wake up the lock and check the conditional in wait(). If it evaluates to true it will proceed. If false, it will go back to sleep.

std::mutex

```
std:mutex mutexInstance
mutexI nst anc e.l ock() // Lock
the process for the thread
mutexI nst anc e.u nlock() //
Unlock the process for the
thread.
```

This creates a mutexInstance. Has the ability to create a lock and unlock mechanism over processes. This prevents certain processes from happening simultaneously more than once.

std::atomic

```
std::atomic<T> name {
initial_value };
// Methods
.store (value, ordering) //
writes a new value to the atomic
object. An enum value from
memory _order
.load( ord ering) // Reads that
value so that it can be used by
other processes
```

std::atomic<T> wraps any type T and guarantees that reading and writing to it from multiple threads is always safe. Without it, two threads touching the same variable simultaneously is undefined behaviour in C++. It essentially puts a protection around the variable so every read and write is always a clean, complete operation.

std::ofstream

```
std::ofstream
my_file("file.txt");
my_file << "some text" <<
std::endl;
my_file << "more text" <<
std::endl;
```

Opens a file for writing, creating it if it doesn't exist.

Writing to it works just like std::cout using the << operator.

The file is automatically closed when the object goes out of scope.