Glossary

This lesson serves as a reference for the most important terms used in this course of concurrency with modern C++.

WE'LL COVER THE FOLLOWING

The idea of this glossary is by no means to be exhaustive

The idea of this glossary is by no means to be exhaustive.

ACID

A transaction is an action that has the properties **A**tomicity, **C**onsistency, **I**solation, and **D**urability (ACID). Except for durability, all properties hold for transactional memory in C++.

- **Atomicity**: either all or no statement of the block is performed.
- **Consistency**: the system is always in a consistent state. All transactions build a total order.
- **Isolation**: each transaction runs in total isolation from the other transactions.

Callable Unit

A callable unit (short callable) is something that behaves like a function. Not only are these named functions, but also function objects and lambda functions. If a callable accepts one argument, it's called unary callable; accepting two arguments is called binary callable.

Predicates are special functions that return a boolean as a result.

Concurrency

Concurrency means that the execution of several tasks overlaps. Concurrency is a superset of *parallelism*.

Critical Section

A critical section is a section of code that should be executed in an atomic fashion following the **ACID** properties.

The ACID properties are typically guaranteed with mutual exclusion primitives such as *mutexes*, or with transactions such as *transactional memory*.

Function Objects

First of all, don't call them functors. That's a *well-defined* term from a branch of mathematics called category theory.

Function objects are objects that behave like functions. They achieve this by implementing the function call operator. As function objects are objects, they can have attributes and therefore state.

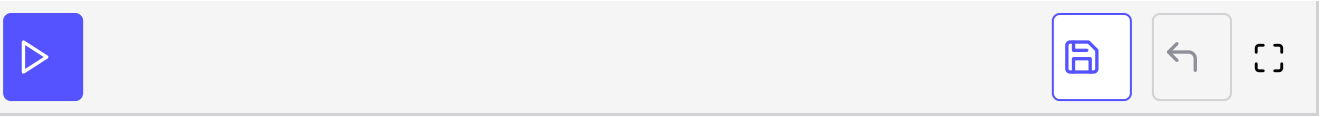
```
#include <iostream>
#include <vector>
#include <algorithm>

struct Square{
  void operator()(int& i){i= i*i;}
};

int main(){
  std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  std::for_each(myVec.begin(), myVec.end(), Square());

  for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
}
```



Instantiate function objects to use them

It's a common error that the name of the function object (`Square`) is used in an algorithm instead of the instance of function object (`Square()`) itself: `std::for_each(myVec.begin(), myVec.end(), Square)` . Of course, that's a typical error. You have to use the instance: `std::for_each(myVec.begin(), myVec.end(), Square())`

Lambda Functions

Lambda functions provide their functionality in-place. The compiler gets its information right on the spot and, therefore, has great optimization potential.

Lambda functions can receive their arguments by value or by reference. They

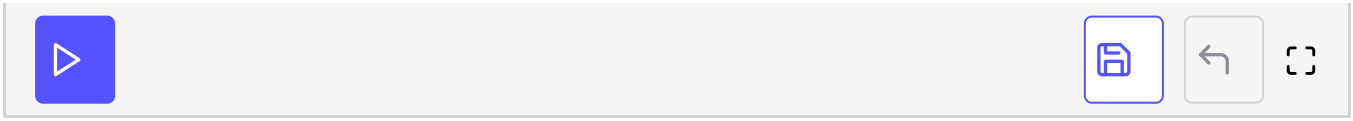
can capture the variables of their defining environment by value or by reference as well.

```
#include <iostream>
#include <vector>
#include <algorithm>

int main(){
  std::vector<int> myVec{1, 2, 3, 4, 5, 6, 7, 8, 9, 10};

  std::for_each(myVec.begin(), myVec.end(), [](int& i){ i= i*i; });

  for (auto v: myVec) std::cout << v << " "; // 1 4 9 16 25 36 49 64 81 100
}
```



Lambda functions should be your first choice

If the functionality of your callable is short and self-explanatory, use a lambda function. A lambda function is generally faster than a function, or a function object and easier to understand.

Lock-free

A non-blocking algorithm is lock-free if there is guaranteed system-wide progress.

Lost Wakeup

A lost wakeup is a situation in which a thread misses its wakeup notification due to a *race condition*.

That may happen if you use a *condition variable without a predicate*.

Modification Order

The modification order is the order in which each memory location is modified.

The memory model guarantees that each memory location has a total modification order; i.e. memory operations performed by the same thread on the same memory location cannot be reordered.

Monad

Haskell as a pure functional language has only pure functions. A key feature of these pure functions is that they will always return the same result when given the same arguments. Thanks to this property - called referential transparency - a Haskell function cannot have side effects; therefore, Haskell has a conceptional issue. The world is full of calculations that have side effects. These are calculations that can fail, that can return an unknown number of results, or that are dependent on the environment. To solve this conceptional issue, Haskell uses monads and embeds them in the purely functional language.

The classical monads encapsulate one side effect:

- **I/O monad**: Calculations that deal with input and output.
- **Maybe monad**: Calculations that maybe return a result.
- **Error monad**: Calculations that can fail.
- **List monad**: Calculations that can have an arbitrary number of results.
- **State monad**: Calculations that build a state.
- **Reader monad**: Calculations that read from the environment.

The concept of the monad is from category theory, which is a part of the mathematics that deals with objects and mapping between these objects. Monads are abstract data types (type classes), which transform simple types into enriched types. Values of these enriched type are called monadic values. Once in a monad, a value can only be transformed by a function composition into another monadic value.

This composition respects the special structure of a monad: therefore, the

error monad will interrupt its calculation if an error occurs or the state

monad builds its state.

To make this happen, a monad consists of three components:

- **Type constructor**: The type constructor defines how the simple data type becomes a monadic data type.
- **Functions**:
  - *Identity function*: Introduces a simple value into the monad.
  - *Bind operator*: Defines how a function is applied to a monadic value to get a new monadic value.
- **Rules for the functions**:
  - The identity function has to be the left and the right identity element.
  - The composition of functions has to be associative.

In order for the error monad to become an instance of the type class monad, the error monad has to support the identity function and the bind operator. Both functions define how the error monad deals with an error in the calculation. If you use an error monad, the error handling is done in the background.

A monad consists of two control flows: the explicit control for calculating the result, and the implicit control flow for dealing with the specific side effect.

Of course, you can define a monad in fewer words: “A monad is just a monoid in the category of endofunctors.”

Monads are becoming more and more import in C++. With C++17, we get `std::optional` which is a kind of a Maybe monad. With C++20, we will probably get *extended futures* and the ranges library from Eric Niebler; both are monads.

Non-blocking

An algorithm is called non-blocking if failure or suspension of any thread cannot cause failure or suspension of another thread. This definition is from the excellent book Java concurrency in practice.

Parallelism

Parallelism means that several tasks will be performed at the same time. Parallelism is a subset of *Concurrency*.

Predicate

Predicates are special *callable units* that return a boolean as the result. If a predicate has one argument, it's called a unary predicate. If a predicate has two arguments, it's called a binary predicate.

RAII

Resource **A**cquisition **I**s **I**nitialization (RAII) stands for a popular technique in C++, in which the resource acquisition and release are bound to the lifetime of an object. For a lock, this means that the mutex will be locked in the constructor and unlocked in the destructor.

Typical use cases in C++ are *locks* that handle the lifetime of its underlying *mutex*, or a smart pointer that handles the lifetime of its resource (memory).

Sequential Consistency

Sequential consistency has two key characteristics:

1. The instructions of a program are executed in source code order
2. There is a global order of all operations on all threads

Sequence Point

A sequence point defines any point in the execution of a program at which it is guaranteed that all effects of previous evaluations will have been

performed, and no effects from subsequent evaluations have yet been performed.

Spurious Wakeup

A spurious wakeup is a phenomenon of condition variables. It may happen that the waiting component of the condition variable erroneously gets a notification, although the notification component didn't notify it.

Thread

In computer science, a thread of execution is the smallest sequence of programmed instructions that can be managed independently by a scheduler, which is typically a part of the operating system. The implementation of threads and processes differs between operating systems, but a thread is a component of a process in most cases. Multiple threads can exist within one process, executing concurrently and sharing resources such as memory, while different processes do not share these resources. In particular, the threads of a process share its executable code and the values of its variables at any given time. For the details, read the Wikipedia article about threads.

Total Order

A total order is a binary relation (<=) on some set X which is antisymmetric, transitive, and total.

- *Antisymmetric*: if a <= b and b <= a then a == b
- *Transitivity*: if a <= b and b <= c then a <= c
- *Totality*: a <= b or b <= a

volatile

`volatile` is typically used to denote objects which can change independently

`volatile` is typically used to denote objects which can change independently of the regular program flow. These are, for example, objects in embedded programming which represent an external device (memory-mapped I/O). Because these objects can change independently of the regular program flow and their value will directly be written into main memory, no optimized storing in caches takes place.

Wait-free

A non-blocking algorithm is wait-free if there is guaranteed per-thread progress.