| Value | Reference |
| --- | --- |
| Everything is copied over to the new object. | A reference is used to pass arguments. |
| C++ | Need garbage collector which will stop the execution to collect garbage. |
| Lots of memory needed | Java |

| Heap | Stack |
| --- | --- |
| Memory allocated during runtime. | Memory allocated during compile time. |

Failing to release the memory will cause memory leak.

# Flattend two dimensional array

we can calculate offset of each element like:

```
array [ m * num cols + n]

m = # row
n = # column
```

This way is better for performance critical programs.

Multipication is easier on hardware than Pointer dereferencing.

If the child thread closes after the main thread, it becomes a zombie thread, since it has no parent thread.

Data Race: when two threads want to access the same shared memory and at least one of them will modify the memory block.

Data Race example:

A "Torn" write:

Thread A start writing 0x12??

Thread B interleaves and writes 0x9567

Thread A continues 0x4534

Critical section: The part of code that only one thread can enter at anytime.

# Mutex

Mutual Exclusion is an object which is used to exclude the threads from the critical section.

IF the mutex is locked, no thread can enter.

IF the mutex is unlocked, one thread can enter only.

# Read-write lock

When many threads read a shared data but only few write, we can use a read-write lock. This is very useful for Financial data and multimedia players.

shared_mutex and shared_lock is used to implement this.

* static variables are thread safe, meaning they won't be initialized twice

* Thread-local variables make sure each thread has its own copy of the object.

# Dead lock

Dead lock happens when two threads wait for each other but never finish.

Example:

1- Thread A locks the mutex.
2- Thread B locks another mutex.
3- Thread A waits to lock mutex 2.
4- Thread B waits to lock mutex 1.

* call_once (flag, func) makes sure the func is called once by only one thread. flag= once_flag

## Conditional Variables

1- The conditional variable can create a loop on the thread, so Thread 1 can wait for thread 2. Thread 1 will continue whenever notify_one() or notify_all() are called.

# Lost wakeup avoidance

The .wait() function takes a second argument which is a predicate (function which returns a bool.). If this bool is true, the conditional variable will move on and not get stuck in the loop. If the bool is false, the conditional variable will behave as usual.

# future and promises

These two classes allow us to share data between two threads. They are in the <future> header. A promise will be passed to the function which generates the value. The generate func can call .set_value() or .set_exception() and after that the consumer func will continue and .get() will return.

If we have multiple consumers, we have to pass a unique future object to each.

## Atomic Types and operations

Atomic <type> variables can be many different built-in type, with the difference that they won't be interrupted while their operations are going on. operations like "++" "--" "+=" and more.

atomic_flag is an atomic boolean, and a spin lock can be implemented using this flag.

Atomic_flag methods:

- test_and_set(): returns the last value of flag and sets its value to true.
- Clear(): makes the flag equal to false.

# Async Function

An async function can be used to write an asynchronous program. The async function takes a callable object as an argument (just like thread constructor). Async function returns a future object instantly.

```
auto res= async(task, arg1);
```

Concurrency describes conceptually distinct tasks which can interact with each other, and is a feature of the program structure.

ex. Team Sport

Explicit parallelism: The programmer decides how to parallelize the work. The tasks are identical and running at the same time.

ex. individual competetive sports

Implicit parallelism: The decision is left to the implementation.

Task parallelism: Also known as "Thread-level parallelism (TLP)" split a large task into smaller tasks. The sub-tasks run concurrently on seperate threads.

Data parallelism: A data set is divided up into several subsets. core 1 processes half the data

and core 2 processes the other half. Also known as "vectorization".

Pipelining: If we have dependent tasks but we want to implement Parallelism as much as possible.