Contents

Alignment general: (last mod is for offset == 0 to work)

```
new_offset = offset + padding = offset + (align - (offset mod align)) mod align
```

power-of-2:

```
new_offset = (offset + align - 1) & ~(align - 1)
```

What are *closures*? What are *lambdas*? Lambda = anonymous local function

Closure = lambda + reference to the environment (a table storing a reference to each of the non-local captured variables)

***move assignment*? When to use it?** Tied to the concept of rvalue-reference (or xvalue)

The move assignment operator is called whenever it is selected by overload resolution, e.g. when an object appears on the left side of an assignment expression, where the right-hand side is an rvalue of the same or implicitly convertible type.

Move assignment operators typically "steal" the resources held by the argument (e.g. pointers to dynamically-allocated objects, file descriptors, TCP sockets, I/O streams, running threads, etc), rather than make copies of them, and leave the argument in some valid but otherwise indeterminate state. For example, move-assigning from a std::string or from a std::vector leaves the right-hand side argument empty.

More http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2006/n2027.html#Move_Semantics

std::move() Obtains an rvalue reference to its argument and converts it to an xvalue.

Code that receives such an xvalue has the opportunity to optimize away unnecessary overhead by moving data out of the argument, leaving it in a valid but unspecified state.

Basically it's just a cast:

```
template<typename _Tp>
constexpr typename std::remove_reference<_Tp>::type&& move(_Tp&& __t) noexcept
{
    return static_cast<typename std::remove_reference<_Tp>::type&&>(__t);
}
```

***static* - all the different meanings?**

1. at *file scope*: signifies "internal-linkage" i.e. not shared between translation units

2. at *function scope*: variable retains value between function calls

3. at *class scope*: signifies independence of class instance

const* and *mutable Const member function doesn't alter the data it operates on; except the one marked as *mutable*

***volatile* keyword** Depends on language and compiler. Usually marks *atomicity* for data (but not guarantees it): reads from threads are guaranteed to have latest; marks that variable can be modified "externally"

***restrict* keyword** Optimisation hint to limit pointer aliasing and aid caching - it means a particular data is accessed only thru that pointer thus making optimisations like storing the ptr value in a registry for subsequent access

http://stackoverflow.com/questions/776283/what-does-the-restrict-keyword-mean-in-c

***in place* new** Allows to explicitly specify the memory management of individual objects, i.e. their "placement" in memory.

new (expression) [(arguments)]; for example:

```
char buffer[] = new char[256];
string *str = new (buffer) string("Hello world");
```

there is no placement delete syntax (but both *new* and *delete* functions can be overrided to specify the in-place)

***typename* keyword**

1. alias for the *class* keyword when declaring template parameters

2. a method to indicate that a dependent name is a type

If the compiler can't tell if a dependent name (one that contains a template parameter) is a value or a type, then it will assume that it is a value.

```
template <typename T>
void foo() {
   T::bar * p; // won't compile, because without the typename prefix it will be interpreted
}

struct Gotcha {
   typedef int bar;
};

foo<Gotcha>();
```

What is a *virtual* function? A virtual function allows derived classes to replace the implementation provided by the base class.

When you refer to a derived class object using a pointer or a reference to the base class, you can call a virtual function for that object and execute the derived class's version of the function. Virtual functions ensure that the correct function is called for an object, regardless of the expression used to make the function call

***pure* virtual function** A virtual function that is required to be implemented by a derived class.

Classes containing pure virtual methods are termed "abstract"; they cannot be instantiated directly.

virtual *destructor* - when/why? At the root of a class hierarchy to insure proper cleanup

virtual call in *assembly*

```
mov eax, dword ptr [this]
mov edx, dword ptr [eax]
mov eax, dword ptr [edx+4]
mov ecx, dword ptr [this]
call eax
```

co-variant return types http://katyscode.wordpress.com/2013/08/22/c-polymorphic-cloning-and-the-crtp-curiously-recurring-template-pattern/

Container	Implementation	Insert	Remove	Index	Find
vector	dynamic array	O(n)	O(n)	O(1)	O(log n)
list	double link list	O(1)	O(1)	-	O(n)
map	red-black b tree	O(log n)	O(log n)	O(1)	O(log n)
hashmap	hash table	O(1)	O(1)	O(1)	O(1)

STL containers speed

---

common thread *synchronization primitives*

1. *CompareAndSwap* instructions - atomic (that is no other thread can preempt it) only writes after the compare with a known value is true

2. *Mutex* - locking mechanism used to synchronize access to a resource. only one "thing" at a time can acquire the mutex and the same "thing" must release it – thus OWNERSHIP property

3. *Semaphore* - generalized mutex based on counting. NO ownership property, multiple threads can increase/decrease, when 0 it waits

4. *Condition Variable* - used for signalling/event passing

5. *Memory Fence* - a class of instructions that mean memory read/writes occur in the order you expect – for example a 'full fence' means all read/writes before the fence are committed before those after the fence.

Multi-Threading problems

Race conditions where the output is dependent on the sequence or timing of other uncontrollable events. It becomes a bug when events don't happen in the order that the programmer intended

Deadlock a situation in which 2 or more competing threads are waiting on each other to finish, and thus neither one advances. Ex: 2 threads wait on each other to release acquired resources

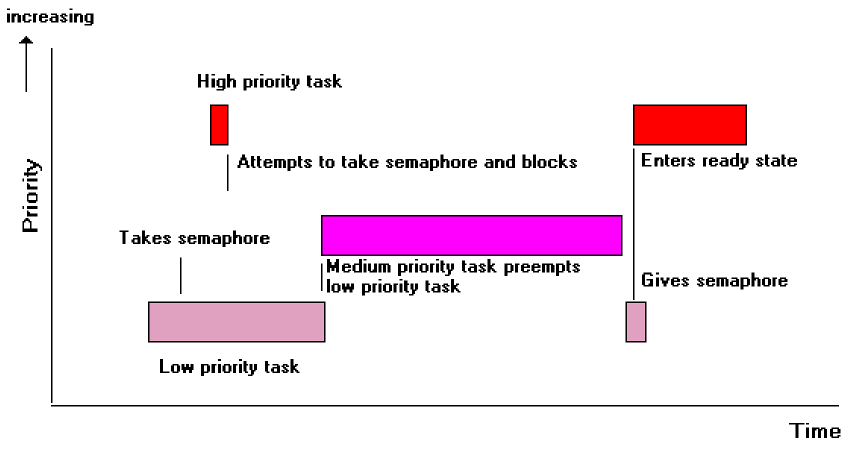
Livelock similar to deadlock but instead of waiting, they change state in regard to eachother - none progressing.

A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

Starvation is where a process is perpetually denied the necessary resources.

Starvation is often caused by errors in a scheduling algorithm, but can also be caused by resource leaks, and can be intentionally caused via a denial-of-service attack such as a fork bomb.

Priority inversion is a problematic scenario in scheduling in which a high priority task is indirectly preempted by a medium priority task effectively "inverting" the relative priorities of the two tasks.


increasing
Priority
High priority task
Attempts to take semaphore and blocks
Enters ready state
Takes semaphore
Medium priority task preempts low priority task
Gives semaphore
Low priority task
Time