A facility, mechanism or construction of a high level programming language is considered to be low level if any of the following affirmations is true:

* the portability characteristic is violated partially or totally;
* is directly making use or is directly correlated with details of representation;
* necessitates knowledge regarding how certain operations are translated in low-level code;
* requires low-level code provided by the programmer;
* places the responsibility of memory management in the hands of the programmer;
* it enables the possibility of writing an operating system.

In the following 10 such features are going to be presented and be awarded a grade (on the scale from 1 to 10) at the end. The grade quantifies the “closeness” of the mechanism/facility/ construction to a low level language (10 being the highest). As a rule of thumb, the grade is awarded in regards to how many affirmations are true for that particular feature, with each affirmation also having a different weight, alongside in how many high-level programming languages is present (to the author’s awareness).

**Union**

A union is a value that may have any of several representations or formats within the same position in memory. To describe such values and variables, some programming languages support special data types, called union types. What a union type definition will specify mainly is which of a number of permitted primitive types may be stored in its instances, e.g., "float or long integer". In contrast with a struct from C for example, which could be defined to contain a float *and* an integer, in a union there is only one value at any given time.

Another way to view a union is as a chunk of memory that is used to store variables of different data types. Once a new value is assigned to a field, the existing data is overwritten with new data. The memory area storing the value has no intrinsic type (other than just bytes or words of memory), but the value can be treated (or interpreted) as one of several abstract data types, having the type of the value that was last written to the memory area.

Languages in which a union data type can be found are C, C++, Cobol and Algol 68. An example for C++ and one for Cobol are provided below.

C++ Cobol

**union** { *01* VERS-INFO.

uint8\_t a; \\ occupies 1 byte 05 VERS-NUM PIC S9(4) COMP.

int32\_t b; \\ occupies 4 bytes05 VERS-BYTES PIC X(2)

int16\_t c; \\ occupies 2 bytes REDEFINES VERS-NUM

}; \\whole union occupies 4 bytes \*vers-num and vers-bytes share memory

The main purpose of union is *to save memory* by using the same memory region for storing different objects at different times**.** This can be better understood if one imagines a union to be similar to a room in a hotel. Different people live in it for non-overlapping periods of time. These people never meet, and generally don't know anything about each other. By properly managing the time-sharing of the rooms (i.e. by making sure different people don't get assigned to one room at the same time), a relatively small hotel can provide accommodations to a relatively large number of people, which is what hotels are for.

The above analogy directly points to the 6th affirmation from our definition since an OS has to behave exactly like the hotel described above and an mechanism like union (or hotel room) would surely come in handy . Another affirmation which could be brought into question is the 2nd one because we are clearly differencing types in this context through their representation (or length of representation to be more precise).

Grade: 7

**Atomic data types**

Before understanding atomicity we first have to consider the problem this concept is trying to solve, namely race conditions. These can be defined as an undesirable situation that occurs when a device or system attempts to perform two or more operations at the same time, but because of the nature of the device or system, the operations must be done in the proper sequence to be done correctly. In computer memory or storage, a race condition may occur if commands to read and write a data are received at almost the same instant, and the machine attempts to overwrite some or all of the old data while that old data is still being read. This happens mainly because a single instruction written in a high level programming language is usually translated in multiple instructions at the level of the processor (for example, just for increasing a counter there are necessary 3 steps: reading the value from memory, increasing it in a register, writing back the value to memory). Combine this with the fact that in concurrent systems with no synchronization mechanisms the scheduler of the operating system decides the order of operations (and it does not care about the point of view of the programmer) and the recipe for unexpected behaviour is completed. Fortunately, concepts like atomicity were introduced to help coming to terms with this problem.

In computer science, an operation done by a computer is considered *atomic* if it is guaranteed to be isolated from other operations that may be happening at the same time. Put another way, atomic operations are indivisible (at least as far as any external observers are concerned).

An atomic data type goes further and ensures that all the operations which are exposed to concurrency issues for that data type are atomic operations.

Languages in which an atomic data type can be found include C++ and Java. Below are examples for both of them

C++ Java

std::atomic<bool> ready (false); AtomicBoolean ready = new AtomicBoolean(false);

std::atomic<int> counter = 0; AtomicInteger counter = new AtomicInteger(0);

From the perspective of our definition, affirmation number 3 is true for this construction since we have to know more than meets the eye when it comes to simple instructions and how they are actually implemented underneath the surface. Indirectly, the 5th affirmation could also be put into question since certain memory regions arrive to be managed (protected) by the programmer.

Grade 7

**Inline assembler**

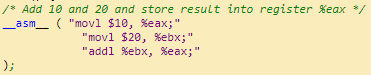
An inline assembler is a feature of some [compilers](https://en.wikipedia.org/wiki/Compiler) that allows low-level code written in [assembly language](https://en.wikipedia.org/wiki/Assembly_language) to be embedded within a program, among code that otherwise has been compiled from a [higher-level language](https://en.wikipedia.org/wiki/High-level_language). such as [C](https://en.wikipedia.org/wiki/C_(programming_language)), C++ or [Ada](https://en.wikipedia.org/wiki/Ada_programming_language).

The embedding of assembly language code is usually done for one of the following reasons:

* optimization
* access to processor specific instructions
* access to special calling conventions not yet supported by the compiler
* system calls and interrupts
* to emit special directives for the linker or assembler

Nevertheless, inline assembler poses a direct problem for the compiler itself as it complicates the analysis of what is done to each variable, a key part of register allocation. This means the performance might actually decrease.

Languages in which an inline assembler is present include C, C++and Ada. An example for C++ is shown below:



The affirmations which are respected from our definition are the following: the 1st one since it makes the program architecture and even compiler specific in this way, alongside the 3rd and 4th ones (obviously). The other 3 affirmations can also be considered to be included by default in the 2 mentioned before.

Grade: 10

**Heap allocation**

The heap is memory set aside for dynamic allocation. Unlike the stack, there is no enforced pattern to the allocation and deallocation of blocks from the heap; you can allocate a block at any time and free it at any time. This makes it much more complex to keep track of which parts of the heap are allocated or free at any given time. Besides maintenance complexity, another problem can arise. If all references to allocated memory are lost (e.g. you don't store a pointer to it anymore), you have what is called a *memory leak*. This is where the memory has still been allocated, but you have no easy way of accessing it anymore. Leaked memory cannot be reclaimed for future memory allocations, but when the program ends the memory will be freed up by the operating system.

Languages in which heap allocation is possible include C, C++, Rust and Fortran. Below are examples for allocating and deallocating memory on the heap in C++ and Fortran.

C++ Fortran

int\* p = **new** int(0); real, **allocatable** **::** ptr

**delete** p; **allocate**( ptr )

ptr = 1.

**deallocate**( ptr )

This functionality clearly puts the control of memory in the hands of the programmers, thus satisfying the 4th affirmation, while also indirectly making use of details of representation (affirmation 2) in the sense that it requires to know how memory is separated and used.

Grade: 6

**Sizeof**

Sizeof (or other under different names) is a method which is used by various programming languages which returns how many bytes of memory a certain data type occupies. The utility of such a method resides in the possibility of applying it to understand the memory usage of certain structures.

Languages in which sizeof (or similar methods) is present contain C, C++ or Python.

C++ Python

int x = 2; import sys

cout << sizeof(x); x = 2

print(sys.getsizeof(x))

From the perspective of our definition, the only affirmation which is fulfilled is the second one since we find information about the length of representation for certain data types.

Grade: 3

**Bitwise operations**

In digital computer programming, a bitwise operation operates on one or more bit patterns or binary numerals at the level of their individual bits. It is a fast and simple action, directly supported by the processor, and is used to manipulate values for comparisons and calculations.

On simple low-cost processors, typically, bitwise operations are substantially faster than division, several times faster than multiplication, and sometimes significantly faster than addition.

The operations are usually applied through specific operators (each with a certain meaning) or combination of operators.

Bitwise operations are necessary particularly in lower-level programming such as device drivers, low-level graphics, communications protocol packet assembly, and decoding.

They are present in the majority of the programming languages. Below are examples for a left shift in C++ and Pascal.

C++ Pascal

int y = 2 ; y := 2;

int x = y << 2 ; x := y **shl** 2;

Regarding our definition, the only affirmation which is true for this feature is the 2nd one since we are making use of the fact that data is represented using bytes.

Grade: 5

**Explicit type conversion**

A cast, or explicit type conversion, is a special programming instruction which specifies what data type to treat a variable as (or an intermediate calculation result) in a given expression.

Each programming language has its own rules on how types can be converted. Languages with strong typing typically do little implicit conversion and discourage the reinterpretation of representations, while languages with weak typing perform many implicit conversions between data types. Weak typing language often allows forcing the compiler to arbitrarily interpret a data item as having different representations—this can be a non-obvious programming error, or a technical method to directly deal with underlying hardware.

The feature is present in a big number of high level programming languages. Below are examples in C++ and Python.

C++ Python

a = 21.09399; a = 21.09399

int c = (int) a; c = int(a)

From the perspective of our definition, the affirmation number 2 is respected since casting is concerned to whether the underlying data representation is converted from one representation into another, or a given representation is merely *reinterpreted* as the representation of another data type.

Grade: 6

**Fill memory blocks**

Filling memory blocks are methods of high level programming languages to set the memory specified through a pointer and a size with a certain value. Their usefulness stems usually from the speed with which they are doing its job (much faster than a “for” instruction for example) and when in combination with other features (for example union).

Languages in which such methods can be found are C, C++ and C#. An example for C++ and one for C# are provided below.

C++

char str[] = "take me away!";

memset (str,'-',6);

C#

<https://gist.github.com/thomaslevesque/6f653d8b3a82b1d038e1>

With reference to our definition, the affirmation number 2 seems to be the most affected since we can specify the how many bytes we are altering alongside their location.

Grade: 5

**Pointer arithmetic**

Pointer arithmetic is slightly different from arithmetic we normally use in our daily life. The only valid arithmetic operations applicable on pointers are:

1. Addition of integer to a pointer
2. Subtraction of integer to a pointer
3. Subtracting two pointers of the same type

The pointer arithmetic is performed relative to the base type of the pointer. For example, if we have an integer pointer ip which contains address 1000, then on incrementing it by 1, we will get 1004 (i.e 1000 + 1 \* 4) instead of 1001 because the size of the int data type is 4 bytes. If we had been using a system where the size of int is 2 bytes then we would get 1002 ( i.e 1000 + 1 \* 2 ).

Languages in which pointer arithmetic is supported are C, C++ and Rust. An example for C++ and one for Rust are provided below.

C++ Rust

**void** main () { fn main() {

**int** arr[5] = {1, 2, 3, 4, 5}; let items = [1usize, 2, 3, 4];

**int** \*p = arr; let ptr = &items[1] as \*const usize;

**for**(int i = 0; i< 5; i++){ println!("{}", unsafe { \*ptr });

printf("%d ",\*(p+i)); println!("{}", unsafe { \*ptr.offset(-1) });

} println!("{}", unsafe { \*ptr.offset(1) });

} }

Pointer arithmetic is strongly coupled with the length of representation of the data, thus being correlated to the second affirmation from our definition. It can also be connected to the 4th one, since pointer arithmetic is quite often combined with heap allocation and thus placing memory managment in the hands of the programmer.

Grade: 6

**Memory barriers**

A memory barrier, also known as a membar, memory fence or fence instruction, is a type of barrier instruction that causes a central processing unit (CPU) or compiler to enforce an ordering constraint on memory operations issued before and after the barrier instruction. This typically means that operations issued prior to the barrier are guaranteed to be performed before operations issued after the barrier.

Memory barriers are typically used when implementing low-level machine code that operates on memory shared by multiple devices. Such code includes synchronization primitives and lock-free data structures on multiprocessor systems, and device drivers that communicate with computer hardware.

Multithreaded programs usually use synchronization primitives provided by a high-level programming environment, such as Java and .NET Framework, or an application programming interface (API) such as POSIX Threads or Windows API. Synchronization primitives such as mutexes and semaphores are provided to synchronize access to resources from parallel threads of execution. These primitives are usually implemented with the memory barriers required to provide the expected memory visibility semantics.

The following link provides an example in C++ of this mechanism: <https://www.modernescpp.com/index.php/acquire-release-fences>

As in the case of atomic, the usage of memory barriers is dependent on understating how certain instructions at a high level are decomposed in multiple ones at a low level (affirmation 3), while also being useful when writing an OS for guarding certain memory areas (affirmation 6).

Grade 7