CHAPTER 1
INTRODUCTION

The aspiration of this thesis is to quantities with how to design efficient algorithms of data structures that can be shared among several execution entities, here each execution entity can be either a process or a thread. Thus, the data structures can be accessed concurrently by the execution entities; either in an interleaved manner where the execution entities gets continuously pre-empted, or fully in parallel.

1.1 Concurrent access of Data Structure in shared memory systems

Concurrent access to a systems objective at supporting computation capacity for large concurrent applications in many research areas like high-energy physics, biomedical sciences and earth sciences. Such applications comprises of tasks/processes that run concurrently and share common data/resources. On the other hand, since the data structures allow many processes to concurrently access them, the contention level on them is high, subsequently degrading performance. Generally, simple data structures perform well in the absence of contention but perform poorly in high-congestion situations. Contrarily, sophisticated data structures that can scale and perform well in the presence of high contention usually suffer unnecessary high latency when there is no contention. Efficient concurrent access of data structures should be able to adapt their algorithmic complexity to contention variation. This fact raises a question on constructing reactive concurrent access to data structures and algorithms that can react to contention variation so as to achieve good performance in all conditions. This thesis deals with how to design efficient algorithms of
data structures that can be shared among several execution entities (i.e. computer programs), where each execution entity can be either a process or a thread. Thus, the data structures can be accessed concurrently by the execution entities; either in an interleaved manner where the execution entities gets continuously pre-empted, or fully in parallel.

In a shared memory system the processes have access to a set of shared memory locations which they may use to communicate. A process can read data from and write data to each shared memory location. The number of processes can be much larger than the number processors due to multiprogramming, which may interleave the execution of several processes on the same processor. The processes are often considered to be asynchronous, that is, their rate of execution might vary arbitrarily, because of the interleaving. This has certain implications for the possibilities for the synchronization and coordination of processes which we will discuss below.

The definition of a modern computer suitable for concurrent programming is quite involved. Depending on the number of processors that are closely connected to each other, a computer can be either a uni or multi-processor system. The computer system can also be constructed out of several separate computation nodes, where each node (which can be a computer system of its own, either a uni or multi-processor) is physically separated from the others, thus forming a distributed system. These computation nodes and whole computer systems can be tied together using either a shared memory system or a message passing system. Depending on how physically separated each computation node is from the others, the system is interpreted as a
single computer or as a cluster system. In a message passing system, the inter-communication between each computation node is done by exchanging information packages over the supported inter-connection network. A shared access to memory system gives a higher abstraction level, and gives the impression to the connected computation nodes of the existence of a global memory with shared access possibilities. See Figure 1.1 & 1.2 for an example scenario where three processes are communicating using message passing versus shared memory techniques.
1.2 Shared Memory System

The real memory that constitutes the shared memory can be either centrally located or distributed in parts over the individual computation nodes. The inter-node communication network needed in order to establish the shared memory can be implemented in either hardware or software on top of message passing hardware. The shared memory system can be either uniformly distributed as in the uniform memory access (UMA) architecture {Alpern [2]} or non-uniformly distributed as in the non-uniform memory access (NUMA) architecture {LaRowe[81]}. In a NUMA system, the response time of a memory access depends on the actual distance between the processor and the real memory, although it is the same for each processor on the same node. For the UMA system, the response time for memory accesses is the same all over the system, see Figure 1.3. The shared memory implementations on computer systems with clearly physically separated computation nodes are usually called distributed shared memory. As the bandwidth of the memory bus or inter-node network is limited, it is important to avoid contention on the shared memory if possible. Heavy contention can lead to significantly lower overall performance of the shared memory system (especially for NUMA systems), and therefore memory {Haken[ 34] }.

1.3 Consistency

In a concurrent system to coordinate the possibly parallel accesses to the shared memory and achieve a consistent view of the shared memory’s state, i.e. each user sees a consistent view of the shared data, including visible changes made by the user's own transactions and transactions of other users. It is necessary that the implementation of
the shared memory guarantees the fulfillment of some formally defined memory consistency model. There are more realistic models have been proposed in the literature.

### 1.3.1 Sequential consistency model

Which is one of the widest accepted models. In this model the results of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program. the underneath figure 1.3 is a sequential consistency model.

![Sequential consistency Model](image)

Figure: 1.3 Sequential consistency Model

### 1.3.2 Casual consistency model

It defines a consistency criteria weaker than sequential consistency. Casual consistency allows for a wait free implementation of read and write operations in a distributed environment, i.e casual consistency allows for cheap read/write operation.

### 1.3.3 Relaxed consistency model

Programmer specifies regions within which global memory operations can be reordered Processor has fence instruction i.e all data operations
before fence in program order must complete before fence is executed and all data operations after fence in program order must wait for fence to complete. Fences are performed in program order. {Adve and Gharachorloo [88]}. 

1.4 Synchronization

Synchronization refers to the idea that multiple processes/tasks are to join up at certain point, in order to reach an agreement or commit to a certain sequence of action. Without synchronization it is highly possible and probable that the concurrent access of some shared resource will lead to inconsistency. The most of shared resources don’t allow more than one process/tasks to access them concurrently in a predictive manner.

1.4.1 Mutual Exclusion

Mutual exclusion means that only a single thread should be able to access the shared resource at any given point of time. Mutual exclusion is often provided by the operating system as a primitive and mostly incorporates the support of shared memory. Using shared memory the tasks can synchronize and share information in shared data structures. Using mutual exclusion the tasks can make sure that only one task can have access to a shared resource or data structure at one time. A mutual exclusion algorithm consists of the Entry and Exit sections. Mutual exclusion algorithms must satisfy the two following requirements {Anderson, Kim, And Herman [33]}

Exclusion : at most one process can be at the critical section at any point in time.

Liveloack-freedom : if there are some processes in the entry section, one process will eventually enter the critical section.
Moreover, an efficient mutual exclusion algorithm should generate small overhead, i.e delay time, due to the execution of entry and exit sections. However, mutual exclusion (also called locking) has some significant drawbacks:

- **BLOCKING**: This means that tasks that are eligible to run have to wait for some other task to finish the access of the shared resource. Blocking also makes the computation of worst-case response times more complicated, and the currently used computation methods are quite pessimistic. Blocking also can cause unwanted delays of other tasks as the effect propagates through the task schedule, also called the convoy effect.

- **DEADLOCK**: one task slowing down can make the whole system consisting of many tasks slow down. If the task that is holding the lock is delayed for some reason, e.g. due to preemption, page faults, cache misses or interrupt handling, other tasks waiting for the lock must suffer the delay even though they are running on other independent and fast tasks.

- **PRIORITY INVERSION**: In real-time systems, a high-priority task must be executed before lower-priority tasks in order to meet its deadline. However, if the tasks communicate using lock-based synchronization, a low-priority task can delay a higher-priority task even if they do not share any objects.

These problems have been recognized since long, especially for real-time systems, and are thoroughly researched. Several solutions exist, such as software packages that fit as a layer between the mutual exclusion control of the shared resource and the scheduler of the real-time system. The most common solutions to mutual exclusion are
called the priority ceiling protocol (PCP) and immediate priority ceiling protocol (IPCP) \cite{Baker92,LehoczkyShaStrosnider41,Rajkumar82,ShaRajkumarLehoczky41}.

1.4.2 Non-blocking Synchronization

There are a number of differently solutions to the problems of synchronizing the access of shared resources. As non-blocking algorithms do not involve mutual exclusion, all steps of the defined operations can be executed concurrently. \cite{MichaelScott54} of common data structures and simple applications, and the study by \cite{LumettaCuller86} of a simple queue structure. Non-blocking techniques can significantly improve the systems’ overall performance as showed by \cite{TsigasZhang72}, on both application as well as operating system level. Several successful attempts to incorporate non-blocking techniques into operating system kernels has been made, for example the Synthesis OS by Massalin and Pu\cite{MassalinPu19} and the Asterix real-time OS by Thane et al. \cite{Thane25}.

This means that the criteria for consistency correctness are a bit more complex than for mutual exclusion. The correctness condition used for concurrent operations (in general) is called linearizability. Linearizability basically means that for each real concurrent execution there exists an equivalent imaginary sequential execution that preserves the partial order of the real execution. The fulfillment of the linearizability property enables the shared resources to still be accessed in a predictive manner, called atomic. In this context, atomic means that the operation can be viewed by the processes as it occurred at a unique instant in time, i.e. the effect of two operations can’t be viewed as taking place at the same time. Traditionally, there are three basic
levels of non-blocking algorithms, called lock-free and wait-free and obstruction-free.

Common with most non-blocking algorithms are that they take advantage of atomic primitives. The definitions of lock-free and wait-free algorithms guarantee the progress of always at least one (all for wait-free) operation, independent of the actions performed by the concurrent operations. As this allows other processes to even completely halt, lock-free and wait-free operations are in this sense strongly fault tolerant. obstruction-free algorithms to be non-blocking, although this kind of algorithms do not give any progress guarantees. Several studies exist that indicate a possibly superior performance for non-blocking implementations compared to traditional ones using mutual exclusion.

1.4.2.1 Lock-Free

Roughly speaking, lock-free is the weakest form of non-blocking synchronization. Lock-free algorithms are designed with having the idea in mind that synchronization conflicts are quite rare and should be handled as exceptions rather than a rule. However, it must then always be possible to detect when those exceptions appear. When a synchronization conflict is noticed during a lock-free operation then that operation is simply restarted from the beginning. The implementations of lock-free concurrent access must rely on strong primitives {Maurice [63] }, e.g., compare & swap (CAS), which atomically updates a memory location if its content is some expected value. see Figure 1.4.
The basic ideas of most lock-free algorithms for updating shared data objects are as follows:

1. Prepare a (partly) copy of the object you want to update.
2. Update the copy
3. Make the copy valid using some strong atomic primitive like Compare-And-Swap (CAS) or similar. The CAS operation makes sure that the update from the old to the new value is done in one atomic step, unless a conflict with some other operation occurred.

If a conflict is noticed, then retry from step 1 again.

This of course assumes that the shared object is constructed using a dynamic structure with linked pointers for each separate node. There exist other lock-free algorithms that rely on other means than swinging pointers with strong atomic primitives. Although lock-free algorithms solve the priority inversion, deadlock and blocking problem, other problems might appear instead. For instance, it might be that a race condition arises between the competing tasks. In the worst case, the
possible retries might continue forever and the operation will never terminate, causing the operation to suffer from what is called starvation. It is important to note that starvation also can occur with many implementations of locks as the definition of mutual exclusion does not guarantee the fairness of concurrent operations, i.e. it is not guaranteed that each process will get the lock in a fair (with respect to the response times for other processes) or even limited time. One way to lower the risk of race conditions as well as the contention on the shared memory is called back-off \cite{Herlihy}. This means that before retrying again after a failed attempt to update, the operation will sleep for a time period selected according to some method. The length of this time period could for example be set initially to some value dependent on the level of concurrency, and then increase either linearly or exponentially with each subsequent retry. Another method that could be combined with the previous or just by itself is randomization.

1.4.2.2 Wait-Free

A wait-free synchronization has additional property that every thread accessing the shared data object can make complete its operation within a bounded number of steps, regardless of the behavior of other threads. Algorithms that can involve an unbounded number of retries due to clashes with other threads are thus not wait-free.

This property means that high-priority threads accessing the share object never have to wait for low-priority threads to complete their operations, and every thread will always be able to make progress when it is scheduled to run by the OS. For real-time or semi-real-time systems this can be an essential property, as the indefinite wait-periods of blocking or non-wait-free lock-free data structures do not allow
their use within time-limited operations. Any wait-free operation will terminate in a finite number of execution steps regardless of the actual level of concurrency, see Figure 1.5.

The downside of wait-free approach is that they are more complex than their non-wait-free counterparts. This imposes an overhead on each operation, potentially making the average time taken to perform an operation considerably longer than the same operation on an equivalent non-wait-free approach.

![Figure:1.5 wait free](image)

### 1.4.2.3 Obstruction-freedom:

A concept of obstruction-freedom has been proposed by {Herlihy, Luchangco, and Moir [51] }due to excessive helping overhead in lock-free/wait-free implementations. The absence of It's a weak non-blocking property, which they believe can provide many of the practical benefits of lock-freedom but with reduced programming complexity and the potential for more efficient data-structure designs. Since efficiently allowing operations to help each other to complete is a major source of complexity in many lock-free algorithms, and excessive helping can generate harmful memory contention, obstruction-freedom can reduce overheads by allowing a conflicting
operation to instead be aborted and retried later. Obstruction-freedom implementations guarantee termination only in the absence of step contention {Anderson, Kim, and Herman [33]}.

More formally, a data structure is obstruction-free if and only if every operation on the structure completes after executing a finite number of steps that do not contend with any concurrent operation for access to any memory location. Thus, although obstruction-freedom is strong enough to prevent effects such as deadlock or priority inversion, an out-of-band mechanism is required to deal with live-lock (which might be caused by two mutually conflicting operations continually aborting each other). The cost of avoiding live-lock in obstruction-free algorithms has not yet been investigated empirically. For example, if exponential back-off is used when retrying a contended operation then it is not certain that there will be a ‘sweet spot’ for the back-off factor in all applications.

1.5 Atomic Primitives

Synchronization is an essential point of hardware/software interaction. In order to enable synchronization between processes accessing the shared memory, the system has to provide some kind of atomic primitives for this purpose. Atomic means that operations which manipulate memory in a way that appears indivisible: No thread can observe the operation half-complete. On modern processors, lots of operations are already atomic. For example, aligned reads and writes of simple types are usually atomic.

In order to make consistent updates of words in the shared memory, stronger atomic primitives than read and writes are needed in practice, although {Lamport [39]} has showed how to achieve mutual exclusion
using only reads and writes. Using mutual exclusion the process is guaranteed to be alone accessing and modifying some part of the shared memory. However, using hardware atomic primitives for updates have several benefits, like that they give better and more predicted performance, as they either take full or no effect. As they are not based on explicit mutual exclusion techniques they also have better fault-tolerance.

There are different kinds of atomic primitives available on different platforms, some less powerful than the others. All platforms do not directly support all known atomic primitives; some only support a limited subset or even none. The latter is especially common on older 8-bit platforms often used for embedded systems. The most collective atomic primitives for synchronization are Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS), which are described in Figure 1.6.

```plaintext
function TAS(value:pointer to word):boolean
atomic do
if *value = 0 then
*value := 1;
return true;
else return false;

procedure FAA(address:pointer to word, number:integer)
atomic do
*address := *address + number;
```
function CAS(address:pointer to word, oldvalue:word, newvalue:word):boolean
atomic do
if *address = oldvalue then
*address := newvalue;
return true;
else return false;

Figure 1.6: The Test-And-Set (TAS), Fetch-And-Add (FAA) and Compare-And-Swap (CAS) atomic primitives.

All the previous primitives are used to make consistent updates of one variable. There are also atomic primitives defined for two and more variables, like Double-Word Compare-And-Swap (CAS2) {Herlihy and Moss [48] }, it’s differ from CAS for double word-size (i.e. 64-bit integers on 32-bit systems). CAS2 do not exist as an implementation in real hardware on any modern platform, the only architecture that supported it was the Motorola 68020 family of processors { Kelly-Bootle[85] }.

1.6 Concurrent Access to Data Structures

A concurrent access of data structure is a particular way of storing and organizing data for access by multiple computing threads (or processes). The data structure which is access concurrently is usually considered to reside in an abstract storage environment called shared memory, though this memory may be physically implemented as either a "tightly coupled" or a distributed collection of storage modules. Concurrent access data structures, intended for use in parallel or
distributed computing environments, differ from "sequential" data structures, intended for use on a uni-processor machine. In a sequential environment one specifies the data structure's properties and checks that they are implemented correctly, by providing safety properties. In a concurrent environment, the specification must also describe liveness properties which an implementation must provide. Safety properties usually state that something bad never happens, while liveness properties state that something good keeps happening.

1.7 Memory Management for Concurrent Data Structures

The problem of managing dynamically allocated memory in a concurrent environment has two parts, keeping track of the free memory available for allocation and safely reclaim allocated memory when it is no longer in use:

1. Memory allocation
2. Memory reclamation.

1.7.1 Memory Allocation

A memory allocator manages a contiguous range of addresses or a set of such ranges, keeping track of which parts of that memory are currently given to the application and which parts are unused and can be used to meet future allocation requests from the application. A traditional malloc general purpose memory allocator is not allowed to move or otherwise disturb memories blocks that are currently owned by the application. Some of the most important properties that distinguish memory allocators for concurrent applications in the literature are: Fragmentation. To minimize fragmentation is to minimize the amount of free memory that cannot be used (allocated) by
the application due to the size of the memory blocks. The concurrent memory allocator should be as fast as a good sequential one when executed on a single processor and its performance should scale with the load in the system.

Early work done on lock-free memory allocation is the work on non-blocking operating systems by {Massalin & Pu.[19] [20] } and {Greenwald [46],Greenwald and Cheriton [45].{Dice and Garth waite [9]} presented LFMalloc, a memory allocator based on the architecture of the {Berger, McKinley, Blumofe, and Hoard [3] } lock-based concurrent memory allocator but with reduced use of locks. {Michael [59]} presented a fully lock-free allocator. {Gidenstam, Papatriantafilou, and Tsigas [14] } presented NBmalloc, another lock-free memory allocator loosely based on the Hoard architecture.

NBmalloc is designed from the requirement that the first-remove-then-insert approach to moving references to large internal blocks of memory (superblocks) around should be avoided and therefore introduces and uses a move operation that can move a reference between different internal data-structures atomically.{Schneider, Antonopoulos, and Nikolopoulos. [87] }presented Streamflow, a lock-free memory allocator that has improved performance over previous solutions due to allowing thread local allocations and deal locations without synchronization.

**1.7.2 Memory Reclamation**

To manage dynamically allocated memory in non-blocking algorithms is difficult due to overlapping operations that might read, change or dereference (i.e. follow) references to dynamically allocated blocks of memory concurrently. One of the most problematic cases is when a
slow process dereferences a pointer value that it previously read from a shared variable. This dereference of the pointer value could occur an arbitrarily long time after the shared pointer holding that value was overwritten and the memory designated by the pointer removed from the shared data structure. Consequently it is impossible to safely free or reuse the block of memory designated by this pointer value until we are sure that there are no such slow processes with pointers to that block. There are several reclamation schemes with a wide and varying range of properties:

I. **Safety of local references**

For local references, which are stored in private variables accessible only by one thread, to be safe the memory reclamation scheme must guarantee that a dynamically allocated node is never reclaimed while there still are local references pointing to it.

II. **Safety of shared references**

Additionally, a memory reclamation scheme could also guarantee that it is always safe for a thread to dereference any shared references located within a dynamic node the thread has a local reference to. Property I alone does not guarantee this, since for a node that has been deleted but cannot be reclaimed yet any shared references within it could reference nodes that have been deleted and reclaimed since the node was removed from the data structure.

III. **Automatic or explicit deletion**

A dynamically allocated node could either be reclaimed automatically when it is no longer accessible through any local or shared reference, that is, the scheme provides automatic garbage collection, or the user
algorithm or data structure could be required to explicitly tell the memory reclamation scheme when a node is removed from the active data structure and should be reclaimed as soon as it has become safe. While automatic garbage collection is convenient for the user, explicit deletion by the user gives the reclamation scheme more information to work with and can help to provide stronger guarantees, e.g. bounds on the amount of deleted but yet unreclaimed memory.

IV. Requirements on the memory allocator

Some memory reclamation schemes require special properties from the memory allocator, like, for example, that each allocable node has a permanent (i.e. for the rest of the system’s lifetime) reference counter associated with it. Other schemes are compatible with the well-known and simple allocate/free allocator interface where the node has ceased to exist after the call to free.

V. Required synchronization primitives

Some memory reclamation schemes are defined using synchronization primitives that few if any current processor architectures provide in hardware, such as for example double word CAS, which then have to be implemented in software often adding considerable overhead. Other schemes make do with single word CAS, single word LL/SC or even just reads and writes alone.

Among these memory reclamation there is non-blocking ones by {Michel [57],[68]}, {Herlihy [69]}. The following one are l based on reference counting: {Detlefs [95] }, {Herlihy [96] } and {Gidenstam [14],[ 15]} and the potentially blocking epoch-based scheme by {Fraser [37] }.
1.8 Proposed Work

The objective of thesis is to study the approach of concurrency applied on different data structure, how a data structure behave in concurrent environment. A data structure named modified skip list is introduced in concurrent environment. There is already a sequential form of this structure is available. A suitable concurrency control approach is applied on the MSL. The concurrency control technique can be based on locking or non-blocking. A concurrent access of priority queue named threaded modified skip list (TMSL) is also introduced. The existing threaded modified skip list is reconstructed with some changes and introduced as priority queue. The presented work also study how the concurrent MSL is better choice than existing data structure.

1.9 Organization of the thesis

Chapter 2 presents a preliminary and survey of literature. Preliminary section cover up basics of different structure like linked list, skip list, modified skip list, out of them modified skip list is the main subject of thesis. This chapter elaborates the basic operations that can be applied on modified skip list like searching, insertion, deletion. This survey categorized the concurrent access to shared data structure on the bases of concurrent access approach i.e. lock based and lock-free. The shared data structures that are covered up in this survey are stack, queue, tree, linked list, priority queue, skip list. A main purpose of this survey is to examine how the different operations that can be applied on conventional data structure are implemented in concurrent environment and what are the needs for concurrent data structure. To meet the performance requirements of concurrent access of data structure, they need to be designed with special consideration on concurrent access.
approach like techniques to access shared data, synchronization of multiple thread, correctness of concurrent algorithm. In the end of this chapter literature survey is summarize in the form of comparison of data structure with different algorithms and their respective pros & cons.

**Chapter 3** We propose a sequential modified skip list with lock based approach of concurrency. In this concurrent structure, locks are used to prevent concurrent threads from interfering with each other. A concurrency scheme chosen for modified skip list data structure must assure the integrity of the data structure, avoid deadlock and have a serializable schedule. Within those restrictions, the presented algorithm is simple, efficient and concurrent as possible. we describe methods for performing concurrent access and update on this MSL with locking approach. Experimental result shows that MSL structure is faster than original skip list structure for representation of dictionaries this is just the beginning to see how MSL behave in concurrent environment.

**Chapter 4** We discuss an efficient and practical lock-free implementation of modified skip list data structure. That is suitable for both fully concurrent systems as well as pre-emptive systems. The approach discussed in previous chapter for concurrent access of MSL based on mutual exclusion, Causes blocking which has several drawbacks and degrades the system’s .The algorithm is implemented using common synchronization primitives that are available in modern systems.

The MSL is structured on the basis of Double link list data structure. Due to concurrent structure of MSL, shared memory multiprocessor system configuration is contemplated. The issue of deletion of a partially inserted node is resolved by using one bit of the pointer values as a deletion mark. Any concurrent insert operation will then be
notified about the deletion, when its CAS operation will fail. For the safe handling of memory the memory management function like READ_NODE, COPY_NODE, RELEASE_NODE are defined.

**Chapter 5** This chapter proposes an alternative approach to base the design of concurrent priority queues on the Modified Skip List data structure. It comprises a concurrent modified Skip List structure, following a simple set of modifications, provides a concurrent priority queue with a higher level of parallelism, it is presented in the simple form and produced significant performance gain. A highly distributed data structures named as Threaded Modified Skip List is designed with a change in the structure of modified skip list.

**Chapter 6** This chapter discuss about how the presented Modified Skip List data structure is better than some other related data structure both in sequential and concurrent access.

**Chapter 7** gives the conclusion and direction for future research work. The given is about the introduction of a new data structure in concurrent system, how it can be implemented with concurrent access technology. A concurrent priority queue based on TMSL is also represented. The future work can how we can apply this concurrent data structure in much better way than skip list in dictionaries data structure in much. Another scope is to determine, whether it’s feasible to apply scaling approach of concurrency on concurrent MSL.

### 1.9 Pseudocode conventions

All the pseudo code fragments in this thesis are written using a C-style programming language. C is a simple and transparent language, which prevents important design details from being hidden behind complex language-level constructs. I introduce the following new primitives:

- A data type bool, taking values TRUE and FALSE.
• An integer data type word, representing a word of memory in the native machine architecture.

• Some standard C operators using a clearer representation: Operator class C representation Pseudo code representation.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assignment</td>
<td>=, :=</td>
</tr>
<tr>
<td>Equality</td>
<td>==, !=, =,</td>
</tr>
<tr>
<td>Relational</td>
<td>&lt;, &gt;, &lt;=, &gt;= &lt;, &gt;, ≤, ≥</td>
</tr>
<tr>
<td>Logical</td>
<td></td>
</tr>
<tr>
<td>Point-to-member</td>
<td>−&gt; →</td>
</tr>
</tbody>
</table>

Figure 1.7: Operator used