# Symbolic I/O interview preparation

# 1. Multi-threading

## 1.1. Deadlock, livelock, starvation

**Deadlock:**

A **deadlock** occurs when the waiting process is still holding on to another resource that the first needs before it can finish.

Resource A and resource B are used by process X and process Y

1. X starts to use A.
2. X and Y try to start using B
3. Y 'wins' and gets B first
4. now Y needs to use A
5. A is locked by X, which is waiting for Y

The best way to avoid deadlocks is to avoid having processes cross over in this way. Reduce the need to lock anything as much as you can.

**Starvation**

*Starvation* describes a situation where a thread is unable to gain regular access to shared resources and is unable to make progress. This happens when shared resources are made unavailable for long periods by "greedy" threads. For example, suppose an object provides a synchronized method that often takes a long time to return. If one thread invokes this method frequently, other threads that also need frequent synchronized access to the same object will often be blocked.

**Livelock**

A thread often acts in response to the action of another thread. If the other thread's action is also a response to the action of another thread, then *livelock* may result. As with deadlock, livelocked threads are unable to make further progress. However, the threads are not blocked — they are simply too busy responding to each other to resume work. This is comparable to two people attempting to pass each other in a corridor: Alphonse moves to his left to let Gaston pass, while Gaston moves to his right to let Alphonse pass. Seeing that they are still blocking each other, Alphone moves to his right, while Gaston moves to his left. They're still blocking each other, so...

## 1.2. mutex vs semaphores

**Atomic operation:**

Atomic operation can be interpreted as an unbreakable or an uninterrupted operation.

**Critical section:**

When executing concurrent programs, multi-threaded operations may acquire shared data simultaneously. So it’s very critical for us to provide synchronization of shared data so that the result will be as predicted.

**Mutex:**

A mutex provides mutual exclusion, either producer or consumer can have the key (mutex) and proceed with their work. As long as the buffer is filled by producer, the consumer needs to wait, and vice versa.

At any point of time, **only one thread can work with the** **entire** **buffer**. The concept can be generalized using semaphore.

**Semaphore:**

A **semaphore is a generalized mutex**. In lieu of single buffer, we can split the 4 KB buffer into four 1 KB buffers (identical resources). A semaphore can be associated with these four buffers. The **consumer and producer can work on different buffers at the same time**.

Strictly speaking, a mutex is **locking mechanism** used to synchronize access to a resource. Only one task (can be a thread or process based on OS abstraction) can acquire the mutex. It means there is ownership associated with mutex, and only the owner can release the lock (mutex).

Semaphore is **signaling mechanism** (“I am done, you can carry on” kind of signal). For example, if you are listening songs (assume it as one task) on your mobile and at the same time your friend calls you, an interrupt is triggered upon which an interrupt service routine (ISR) signals the call processing task to wakeup.

## 1.3. Process vs Thread

**Processes**

A process has a self-contained execution environment. A process generally has a complete, private set of basic run-time resources; in particular, each process has its own memory space.

Processes are often seen as synonymous with programs or applications. However, what the user sees as a single application may in fact be a set of cooperating processes. To facilitate communication between processes, most operating systems support *Inter Process Communication* (IPC) resources, such as pipes and sockets. IPC is used not just for communication between processes on the same system, but processes on different systems.

**Threads**

Threads are sometimes called *lightweight processes*. Both processes and threads provide an execution environment, but creating a new thread requires fewer resources than creating a new process.

Threads exist within a process — every process has at least one. Threads share the process's resources, including memory and open files. This makes for efficient, but potentially problematic, communication.

Multithreaded execution is an essential feature of the Java platform. Every application has at least one thread — or several, if you count "system" threads that do things like memory management and signal handling. But from the application programmer's point of view, you start with just one thread, called the *main thread*. This thread has the ability to create additional threads, as we'll demonstrate in the next section.

1. Processes are independent while thread is within a process.

2. Processes have separate address spaces while threads share their address spaces.

3. Processes communicate each other through inter-process communication.

4. Processes carry considerable state (e.g., ready, running, waiting, or stopped) information, whereas multiple threads within a process share state as well as memory and other resources.

5. [Context switching](http://www.bogotobogo.com/cplusplus/multithreaded.php" \l "ContextSwitch" \t "_blank) between threads in the same process is typically faster than context switching between processes.

**6. Multithreading** has some advantages over **multiple processes**. Threads require less overhead to manage than processes, and intraprocess thread communication is less expensive than interprocess communication.

**7. Multiple process** concurrent programs do have one advantage: Each process can execute on a different machine (**distribute program**). Examples of distributed programs are file servers (NFS), file transfer clients and servers (FTP), remote log-in clients and servers (Telnet), groupware programs, and Web browsers and servers.

## 1.4. Producer/Consumer problem

Single producer – single consumer:



## 1.5. Inter-process communication

**1. Pipes**



**2. Signals**

**3. Message Queues**

The key is an arbitrary value or one that can be derived from a common seed at run time. One way is with ftok() , which converts a filename to a key value that is unique within the system. Functions that initialize or get access to messages (also semaphores or shared memory see later) return an ID number of type int. IPC functions that perform read, write, and control operations use this ID. If the key argument is specified as IPC\_PRIVATE, the call initializes a new instance of an IPC facility that is private to the creating process. When the IPC\_CREAT flag is supplied in the flags argument appropriate to the call, the function tries to create the facility if it does not exist already. When called with both the IPC\_CREAT and IPC\_EXCL flags, the function fails if the facility already exists. This can be useful when more than one process might attempt to initialize the facility. One such case might involve several server processes having access to the same facility. If they all attempt to create the facility with IPC\_EXCL in effect, only the first attempt succeeds. If neither of these flags is given and the facility already exists, the functions to get access simply return the ID of the facility. If IPC\_CREAT is omitted and the facility is not already initialized, the calls fail. These control flags are combined, using logical (bitwise) OR, with the octal permission modes to form the flags argument. For example, the statement below initializes a new message queue if the queue does not exist.

**4. Semaphores**

Semaphores let processes query or alter status information. They are often used to monitor and control the availability of system resources such as shared memory segments.

Semaphores can be operated on as individual units or as elements in a set. Because System V IPC semaphores can be in a large array, they are extremely heavy weight. Much lighter weight semaphores are available in the threads library (see man semaphore and also Chapter [30.3](https://www.cs.cf.ac.uk/Dave/C/node31.html#ch:thread_sem)) and POSIX semaphores (see below briefly). Threads library semaphores must be used with mapped memory . A semaphore set consists of a control structure and an array of individual semaphores. A set of semaphores can contain up to 25 elements.

In a similar fashion to message queues, the semaphore set must be initialized using semget(); the semaphore creator can change its ownership or permissions using semctl(); and semaphore operations are performed via the semop() function.

**5. Shared Memory**

A process creates a shared memory segment using shmget()|. The original owner of a shared memory segment can assign ownership to another user with shmctl(). It can also revoke this assignment. Other processes with proper permission can perform various control functions on the shared memory segment using shmctl(). Once created, a shared segment can be attached to a process address space using shmat(). It can be detached using shmdt() (see shmop()). The attaching process must have the appropriate permissions for shmat(). Once attached, the process can read or write to the segment, as allowed by the permission requested in the attach operation. A shared segment can be attached multiple times by the same process. A shared memory segment is described by a control structure with a unique ID that points to an area of physical memory. The identifier of the segment is called the shmid. The structure definition for the shared memory segment control structures and prototypews can be found in <sys/shm.h>.

**6. Sockets**

Sockets provide point-to-point, two-way communication between two processes. Sockets are very versatile and are a basic component of interprocess and intersystem communication. **A socket is an endpoint of communication to which a name can be bound. It has a type and one or more associated processes.**

Sockets exist in communication domains. A socket domain is an abstraction that provides an addressing structure and a set of protocols. Sockets connect only with sockets in the same domain. Twenty three socket domains are identified (see <sys/socket.h>), of which only the UNIX and Internet domains are normally used Solaris 2.x Sockets can be used to communicate between processes on a single system, like other forms of IPC.

The UNIX domain provides a socket address space on a single system. UNIX domain sockets are named with UNIX paths. Sockets can also be used to communicate between processes on different systems. The socket address space between connected systems is called the Internet domain.

Internet domain communication uses the TCP/IP internet protocol suite.

***Socket types*** define the communication properties visible to the application. Processes communicate only between sockets of the same type. There are five types of socket.

**A stream socket**

-- provides two-way, sequenced, reliable, and unduplicated flow of data with no record boundaries. A stream operates much like a telephone conversation. The socket type is SOCK\_STREAM, which, in the Internet domain, uses Transmission Control Protocol (TCP).

**A datagram socket**

-- supports a two-way flow of messages. A on a datagram socket may receive messages in a different order from the sequence in which the messages were sent. Record boundaries in the data are preserved. Datagram sockets operate much like passing letters back and forth in the mail. The socket type is SOCK\_DGRAM, which, in the Internet domain, uses User Datagram Protocol (UDP).

**A sequential packet socket**

-- provides a two-way, sequenced, reliable, connection, for datagrams of a fixed maximum length. The socket type is SOCK\_SEQPACKET. No protocol for this type has been implemented for any protocol family.

**A raw socket**

provides access to the underlying communication protocols.

These sockets are usually datagram oriented, but their exact characteristics depend on the interface provided by the protocol.

# 2. C++

## 2.1. Smart pointer vs raw pointer



Smart pointers are defined in the **std** namespace in the [<memory>](https://msdn.microsoft.com/en-us/library/k11k2x83.aspx) header file. They are crucial to the [RAII](https://msdn.microsoft.com/en-us/library/hh438480.aspx) or **Resource Acquisition Is Initialization** programming idiom. The main goal of this idiom is to ensure that resource acquisition occurs at the same time that the object is initialized, so that all resources for the object are created and made ready in one line of code. In practical terms, the main principle of RAII is to give ownership of any heap-allocated resource—for example, dynamically-allocated memory or system object handles—to a stack-allocated object whose destructor contains the code to delete or free the resource and also any associated cleanup code.

In most cases, when you initialize a raw pointer or resource handle to point to an actual resource, pass the pointer to a smart pointer immediately. In modern C++, raw pointers are only used in small code blocks of limited scope, loops, or helper functions where performance is critical and there is no chance of confusion about ownership.

As shown in the example, a smart pointer is a class template that you declare on the stack, and initialize by using a raw pointer that points to a heap-allocated object. After the smart pointer is initialized, it owns the raw pointer. This means that the smart pointer is responsible for deleting the memory that the raw pointer specifies. The smart pointer destructor contains the call to delete, and because the smart pointer is declared on the stack, its destructor is invoked when the smart pointer goes out of scope, even if an exception is thrown somewhere further up the stack.

## 2.2. Polymophysm

The word **polymorphism** means having many forms. Typically, polymorphism occurs when there is a hierarchy of classes and they are related by inheritance.

C++ polymorphism means that a call to a member function will cause a different function to be executed depending on the type of object that invokes the function.

This is how **polymorphism** is generally used. You have different classes with a function of the same name, and even the same parameters, but with different implementations.

## 2.3. Reference and pointer

1. References must be initialized, while pointers don’t have to be.
2. Pointers can be null, references can’t.
3. Pointers may be altered during run of a program, references, on the other hand, always refer to the object initialized.

In general, you should use a pointer whenever you need to take into account the possibility that there's nothing to refer to (in which case you can set the pointer to null) or whenever you need to be able to refer to different things at different times (in which case you can change where the pointer points). You should use a reference whenever you know there will always be an object to refer to and you also know that once you're referring to that object, you'll never want to refer to anything else.

There is one other situation in which you should use a reference, and that's when you're implementing certain operators. The most common example is operator[]. This operator typically needs to return something that can be used as the target of an assignment.

## 2.4. Virtual table

To implement virtual functions, C++ uses a special form of late binding known as the virtual table. The **virtual table** is a lookup table of functions used to resolve function calls in a dynamic/late binding manner.

The virtual table is actually quite simple, though it’s a little complex to describe in words. First, every class that uses virtual functions (or is derived from a class that uses virtual functions) is given it’s own virtual table. This table is simply a static array that the compiler sets up at compile time. **A virtual table contains one entry for each virtual function that can be called by objects of the class.** Each entry in this table is simply a function pointer that points to the most-derived function accessible by that class.

Second, the compiler also adds a hidden pointer to the base class, which we will call \*\_\_vptr. \*\_\_vptr is set (automatically) when a class instance is created so that it points to the virtual table for that class. Unlike the \*this pointer, which is actually a function parameter used by the compiler to resolve self-references, \*\_\_vptr is a real pointer. Consequently, it makes each class object allocated bigger by the size of one pointer. It also means that \*\_\_vptr is inherited by derived classes, which is important.

Calling a virtual function is slower than calling a non-virtual function for a couple of reasons: First, we have to use the \*\_\_vptr to get to the appropriate virtual table. Second, we have to index the virtual table to find the correct function to call. Only then can we call the function. As a result, we have to do 3 operations to find the function to call, as opposed to 2 operations for a normal indirect function call, or one operation for a direct function call. However, with modern computers, this added time is usually fairly insignificant.

## 2.6. lvalue and rvalue

Every C++ expression is either an lvalue or an rvalue. An lvalue refers to an object that persists beyond a single expression. You can think of an lvalue as an object that has a name. All variables, including nonmodifiable (**const**) variables, are lvalues. An rvalue is a temporary value that does not persist beyond the expression that uses it.



## 2.7. Static

Static members of a class are not associated with the objects of the class: they are independent objects with static storage duration or regular functions defined in namespace scope, only once in the program.

Static member functions

Static member functions are not associated with any object. When called, they have no this pointer.

Static member functions cannot be virtual, const, or volatile.

The address of a static member function may be stored in a regular pointer to function, but not in a pointer to member function.

Static data members

Static data members are not associated with any object. They exist even if no objects of the class have been defined. If the static member is declared thread\_local(since C++11), there is one such object per thread. Otherwise, there is only one instance of the static data member in the entire program, with static storage duration.

Static data members cannot be mutable.

Static data members of a class in namespace scope have external linkage if the class itself has external linkage (i.e. is not a member of unnamed namespace). Local classes (classes defined inside functions) and unnamed classes, including member classes of unnamed classes, cannot have static data members.

# 3. Memory allocation

global variable-> .DATA

static variale-> .BSS if initialized as 0, .DATA else

local variable -> Stack

constant type -> code or .DATA

variables in main() -> Stack

pointers -> .DATA or stack

dynamically allocated variable -> Heap

# 4. On Linux system, how does program run

## Send signals to process in UNIX/Linux

1) kill command:

1) SIGHUP 2) SIGINT 3) SIGQUIT 4) SIGILL 5) SIGTRAP 6) SIGABRT 7) SIGEMT 8) SIGFP 9) SIGKILL 10) SIGBUS 11) SIGSEGV 12) SIGSYS 13) SIGPIPE 14) SIGALRM 15) SIGTERM 16) SIGURG 17) SIGSTOP 18) SIGTSTP 19) SIGCONT 20) SIGCHLD 21) SIGTTIN 22) SIGTTOU 23) SIGIO 24) SIGXCPU 25) SIGXFSZ 26) SIGVTALRM 27) SIGPROF 28) SIGWINCH 29) SIGINFO 30) SIGUSR1 31) SIGUSR2

2) Write a c++ program to send signal



3) Keyboard

# 5. Type of search and sorting algorithms

[gitgub](https://github.com/allenwhc/Algorithm/blob/master/Sorting/ArraySorting/src/Sorting.java)

# 6. UDP vs. TCP

# 7. Command to debug in gdb

# 8. Exception handling

# 9. Design patterns

A design pattern is neither a static solution, nor is it an algorithm. A **pattern** is a way to describe and address by name (mostly a simplistic description of its goal), a repeatable solution or approach to a common design problem, that is, a common way to solve a generic problem (how generic or complex, depends on how restricted the target goal is). Patterns can emerge on their own or by design. This is why design patterns are useful as an abstraction over the implementation and a help at design stage. With this concept, an easier way to facilitate communication over a design choice as normalization technique is given so that every person can share the design concept.

Patterns are commonly found in objected-oriented programming languages like C++ or Java. They can be seen as a template for how to solve a problem that occurs in many different situations or applications. It is not code reuse, as it usually does not specify code, but code can be easily created from a design pattern. Object-oriented design patterns typically show relationships and interactions between classes or objects without specifying the final application classes or objects that are involved.

Each design pattern consists of the following parts:

**Problem/requirement**

To use a design pattern, we need to go through a mini analysis design that may be coded to test out the solution. This section states the requirements of the problem we want to solve. This is usually a common problem that will occur in more than one application.

**Forces**

This section states the technological boundaries, that helps and guides the creation of the solution.

**Solution**

This section describes how to write the code to solve the above problem. This is the design part of the design pattern. It may contain class diagrams, sequence diagrams, and or whatever is needed to describe how to code the solution.

Design patterns can be considered as a standardization of commonly agreed best practices to solve specific design problems. One should understand them as a way to implement good design patterns within applications. Doing so will reduce the use of inefficient and obscure solutions design patterns speeds up your design and helps to communicate it to other programmers.

# 10. Event driven programming, e.g. function callback

# 11. RAII

*Resource Acquisition Is Initialization* or RAII, is a C++ programming technique which binds the life cycle of a resource (allocated memory, thread of execution, open socket, open file, locked mutex, database connection—anything that exists in limited supply) to the lifetime of an object.

RAII guarantees that the resource is available to any function that may access the object (resource availability is a class invariant). It also guarantees that all resources are released when the lifetime of their controlling object ends, in reverse order of acquisition. Likewise, if resource acquisition fails (the constructor exits with an exception), all resources acquired by every fully-constructed member and base sub-object are released in reverse order of initialization. This leverages the core language features (object lifetime, scope exit, order of initialization and stack unwinding) to eliminate resource leaks and guarantee exception safety. Another name for this technique is *Scope-Bound Resource Management* (SBRM), after the basic use case where the lifetime of an RAII object ends due to scope exit.

RAII can be summarized as follows:

Encapsulate each resource into a class, where

The constructor acquires the resource and establishes all class invariants or throws an exception if that cannot be done,

The destructor releases the resource and never throws exceptions:

Always use the resource via an instance of a RAII-class that either

Has automatic storage duration or temporary lifetime itself, or

Has lifetime that is bounded by the lifetime of an automatic or temporary object

# 12. Algorithm problem

**1. Merge intervals:**



**2. Clone graph**

