Both consumer.cpp and producer.cpp are formatted into three sections:

1. Semaphore and Shared Memory Creation
2. Interacting with Memory
3. Closing Semaphore and Shared Memory

Section 1 - Semaphore and Shared Memory Creation

For the sake of this project, I designated producer.cpp to be responsible for the initial creation and final destruction of the semaphore and shared memory and consumer.cpp would open and close a connection with the tools provided by producer.cpp. This does not look too much different for the creation of the semaphore, but for the creation of the shared memory blocks, producer.cpp had the added task of listing how much space should be reserved for each memory block.

There is one Semaphore implemented, betweenProcess, and two bits of shared memory, one named my_shared_memory and the other named my_shared_counter. The semaphore is used to control the flow of access to the shared memory, in order to implement Mutual Exclusion. The memory designated my_shared_memory was used to hold an array of characters with a size of 2, and the memory designated my_shared_counter held a value corresponding to how many characters in my_shared_memory needed to be consumed still.

Section 2 - Interacting with Memory

In this section, there is a separate implementation for each file.

Producer.cpp

The first task is establishing a string to hold the characters which need to be sent to the other process. After this, we loop through each character in the string and sent them to consumer.cpp one at a time, stopping only when counter is equal to maxBufferHold, which is set to 2, as this means that the buffer is full and producer needs to wait for consumer.cpp to take the provided information. When the loop is stopped to wait for the counter to go down, we send the text “Full” to the terminal to verify that we are not over filling the buffer.

When sending the characters, we first determine which slot in the array we should place the information in, and then we give that slot in the array the next value. Finally, we increase the counter by 1 to show that there is more information available.

After every character has been sent, one last character is sent, the null terminating character (‘\0’) to signal to the consumer that it is done sending information. Then producer.cpp waits for consumer.cpp to change the counter’s value to -1, signaling that it is time to close the semaphore and the shared memory.

Consumer.cpp

First we set up a loop to continuously check if there is information to pull from the shared memory array. When there is not, the counter should have a value of 0, which tells the consumer program to wait for information to be added, and the program also prints out the text “empty” to signal that it is not trying to pull information from the array that is not there.

When there is information to pull, consumer.cpp will grab the character from the next expected slot and add it on to the end of the string holding every character it has already collected. Finally, it decreases the counter by 1 to show that it has pulled one piece of information from the array.

When consumer.cpp receives the null terminating character, it prints the final message out to the terminal, and sets counter to -1 to signify that it has received all of the information.

Section 3 - Closing Semaphore and Shared Memory

In this section both files go through each connection to either a semaphore or shared memory and tells the operating system that it is done using it by calling the provided function. Finally, producer.cpp unlinks the semaphore and shared memory used to make the space and name for these available for use by the next program.

Three major concepts are implemented in these programs:

1. Shared Memory
2. Mutual Exclusion
3. Semaphores

Shared Memory is memory reserved in the system outside of either designated process, and as such multiple processes have access to its contents. This gives processes a way to communicate information between them. Mutual Exclusion is the concept of making sure that only one process can read or write over the shared memory at a time, so that there is a predictable flow of information. Mutual Exclusion is then implemented in this program using Semaphores. Semaphores are like a boolean value that tells all other processes that the current process is in its critical section, where it is manipulating the shared memory.

In these programs we use predefined libraries in C++ POSIX to implement shared memory and semaphores in our program, using the semaphores before any part of the program in which shared memory is being read or written over, as to implement Mutual Exclusion.