**匿名管道-官网介绍**

**Anonymous Pipes**

An *anonymous pipe* is an unnamed, one-way pipe that typically transfers data between a parent process and a child process. Anonymous pipes are always local; they cannot be used for communication over a network.

For more information, see the following topics:

* [Anonymous Pipe Operations](https://docs.microsoft.com/zh-cn/windows/win32/ipc/anonymous-pipe-operations)
* [Pipe Handle Inheritance](https://docs.microsoft.com/zh-cn/windows/win32/ipc/pipe-handle-inheritance)
* [Anonymous Pipe Security and Access Rights](https://docs.microsoft.com/zh-cn/windows/win32/ipc/anonymous-pipe-security-and-access-rights)

# Anonymous Pipe Operations

The [**CreatePipe**](https://msdn.microsoft.com/en-us/library/Aa365152(v=VS.85).aspx) function creates an anonymous pipe and returns two handles: a read handle to the pipe and a write handle to the pipe. The read handle has read-only access to the pipe, and the write handle has write-only access to the pipe. To communicate using the pipe, the pipe server must pass a pipe handle to another process. Usually, this is done through inheritance; that is, the process allows the handle to be inherited by a child process. The process can also duplicate a pipe handle using the [**DuplicateHandle**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-duplicatehandle) function and send it to an unrelated process using some form of interprocess communication, such as DDE or shared memory.

CreatePipe函数创建一个匿名管道并返回两个句柄:管道的读句柄和管道的写句柄。读句柄对管道具有只读访问，写句柄对管道具有只写访问。要使用管道进行通信，管道服务器必须将管道句柄传递给另一个进程。通常，这是通过继承完成的; 也就是说，进程允许子进程继承句柄。进程还可以使用DuplicateHandle函数复制管道句柄，并使用某种进程间通信形式(如DDE或共享内存)将其发送给不相关的进程。

A pipe server can send either the read handle or the write handle to the pipe client, depending on whether the client should use the anonymous pipe to send information or receive information. To read from the pipe, use the pipe's read handle in a call to the [**ReadFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-readfile) function. The **ReadFile** call returns when another process has written to the pipe. The **ReadFile** call can also return if all write handles to the pipe have been closed or if an error occurs before the read operation has been completed.

管道服务器可以向管道客户机发送读句柄或写句柄，这取决于客户机是使用匿名管道发送信息还是接收信息。要从管道中读取数据，请在调用ReadFile函数时使用管道的read句柄。当另一个进程写入管道时，ReadFile调用返回。如果对管道的所有写句柄都已关闭，或者在完成读操作之前发生错误，ReadFile调用也可以返回。

To write to the pipe, use the pipe's write handle in a call to the [**WriteFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-writefile) function. The **WriteFile** call does not return until it has written the specified number of bytes to the pipe or an error occurs. If the pipe buffer is full and there are more bytes to be written, **WriteFile** does not return until another process reads from the pipe, making more buffer space available. The pipe server specifies the buffer size for the pipe when it calls [**CreatePipe**](https://msdn.microsoft.com/en-us/library/Aa365152(v=VS.85).aspx).

若要向管道写入，请在调用WriteFile函数时使用管道的写句柄。在将指定的字节数写入管道或发生错误之前，WriteFile调用不会返回。如果管道缓冲区已满，需要写入更多字节，则WriteFile直到另一个进程从管道读取数据时才返回，从而使更多的缓冲区空间可用。管道服务器在调用CreatePipe时为管道指定缓冲区大小。

Asynchronous (overlapped) read and write operations are not supported by anonymous pipes. This means that you cannot use the [**ReadFileEx**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-readfileex) and [**WriteFileEx**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-writefileex) functions with anonymous pipes. In addition, the *lpOverlapped* parameter of [**ReadFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-readfile) and [**WriteFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-writefile) is ignored when these functions are used with anonymous pipes.

匿名管道不支持异步(重叠)读写操作。这意味着您不能将ReadFileEx和WriteFileEx函数与匿名管道一起使用。此外，当这些函数与匿名管道一起使用时，ReadFile和WriteFile的llapped参数将被忽略。

An anonymous pipe exists until all pipe handles, both read and write, have been closed. A process can close its pipe handles by using the **[CloseHandle](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-closehandle)** function. All pipe handles are also closed when the process terminates.

匿名管道存在，直到所有管道句柄(读和写)都已关闭。进程可以使用[**CloseHandle**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-closehandle)函数来关闭它的管道句柄。当进程终止时，所有的管道句柄也被关闭。

Anonymous pipes are implemented using a named pipe with a unique name. Therefore, you can often pass a handle to an anonymous pipe to a function that requires a handle to a named pipe.

匿名管道是使用具有唯一名称的命名管道实现的。因此，通常可以将匿名管道的句柄传递给需要指定管道句柄的函数。

# Pipe Handle Inheritance

The pipe server controls whether its handles can be inherited in the following ways:

管道服务器控制是否可以通过以下方式继承其句柄

* The [**CreatePipe**](https://msdn.microsoft.com/en-us/library/Aa365152(v=VS.85).aspx) function receives a [**SECURITY\_ATTRIBUTES**](https://docs.microsoft.com/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) structure. If the pipe server sets the **bInheritHandle** member of this structure to **TRUE**, the handles created by **CreatePipe** can be inherited.

CreatePipe函数接收一个SECURITY\_ATTRIBUTES结构。如果管道服务器将该结构的bInheritHandle成员设置为TRUE，则可以继承CreatePipe创建的句柄

* The pipe server can use the [**DuplicateHandle**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-duplicatehandle) function to change the inheritance of a pipe handle. The pipe server can create a noninheritable duplicate of an inheritable pipe handle or an inheritable duplicate of a noninheritable pipe handle.

管道服务器可以使用DuplicateHandle函数来更改管道句柄的继承。管道服务器可以创建可继承管道句柄的不可继承副本或不可继承管道句柄的可继承副本。

* The [**CreateProcess**](https://docs.microsoft.com/windows/desktop/api/processthreadsapi/nf-processthreadsapi-createprocessa) function enables the pipe server to specify whether a child process inherits all or none of its inheritable handles.

CreateProcess函数使管道服务器能够指定子进程是否继承其所有可继承句柄或不继承句柄。

When a child process inherits a pipe handle, the system enables the process to access the pipe. However, the parent process must communicate the handle value to the child process. The parent process typically does this by redirecting the standard output handle to the child process, as shown in the following steps:

当子进程继承管道句柄时，系统允许该进程访问管道。但是，父进程必须将句柄值传递给子进程。父进程通常通过将标准输出句柄重定向到子进程来做到这一点，如下面的步骤所示:

1. Call the [**GetStdHandle**](https://docs.microsoft.com/windows/console/getstdhandle) function to get the current standard output handle; save this handle so you can restore the original standard output handle after the child process has been created.

调用GetStdHandle函数获取当前标准输出句柄;保存此句柄，以便在创建子进程之后可以恢复原始的标准输出句柄

1. Call the **[SetStdHandle](https://docs.microsoft.com/windows/console/setstdhandle)** function to set the standard output handle to the write handle to the pipe. Now the parent process can create the child process.

调用SetStdHandle函数将标准输出句柄设置为管道的写句柄。现在父进程可以创建子进程了

1. Call the [**CloseHandle**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-closehandle) function to close the write handle to the pipe. After the child process inherits the write handle, the parent process no longer needs its copy.

调用CloseHandle函数关闭管道的写句柄。子进程继承写句柄后，父进程不再需要它的副本。

1. Call [**SetStdHandle**](https://docs.microsoft.com/windows/console/setstdhandle) to restore the original standard output handle.

调用SetStdHandle来恢复原来的标准输出句柄。

The child process uses the [**GetStdHandle**](https://docs.microsoft.com/windows/console/getstdhandle) function to get its standard output handle, which is now a handle to the write end of a pipe. The child process then uses the [**WriteFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-writefile) function to send its output to the pipe. When the child has finished with the pipe, it should close the pipe handle by calling [**CloseHandle**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-closehandle) or by terminating, which automatically closes the handle.

子进程使用GetStdHandle函数获得其标准输出句柄，该句柄现在是管道的写端句柄。然后，子进程使用WriteFile函数将其输出发送到管道。当子进程使用完管道后，它应该通过调用CloseHandle或terminating来关闭管道句柄，这会自动关闭句柄。

The parent process uses the [**ReadFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-readfile) function to receive input from the pipe. Data is written to an anonymous pipe as a stream of bytes. This means that the parent process reading from a pipe cannot distinguish between the bytes written in separate write operations, unless both the parent and child processes use a protocol to indicate where the write operation ends. When all write handles to the pipe are closed, the **ReadFile** function returns zero. It is important for the parent process to close its handle to the write end of the pipe before calling **ReadFile**. If this is not done, the **ReadFile** operation cannot return zero because the parent process has an open handle to the write end of the pipe.

父进程使用ReadFile函数来接收来自管道的输入。数据以字节流的形式写入匿名管道。这意味着从管道中读取数据的父进程无法区分在单独的写操作中写入的字节，除非父进程和子进程都使用协议来指示写操作的结束位置。当对管道的所有写句柄都关闭时，ReadFile函数返回零。父进程在调用ReadFile之前关闭管道的写端句柄是很重要的。如果不这样做，ReadFile操作就不能返回零，因为父进程对管道的写端有一个打开的句柄。

The procedure for redirecting the standard input handle is similar to that for redirecting the standard output handle, except that the pipe's read handle is used as the child's standard input handle. In this case, the parent process must ensure that the child process does not inherit the pipe's write handle. If this is not done, the [**ReadFile**](https://docs.microsoft.com/windows/desktop/api/fileapi/nf-fileapi-readfile) operation performed by the child process cannot return zero because the child process has an open handle to the write end of the pipe.

重定向标准输入句柄的过程与重定向标准输出句柄的过程类似，只是管道的read句柄用作子输入句柄。在这种情况下，父进程必须确保子进程不继承管道的写句柄。如果不这样做，子进程执行的ReadFile操作将不能返回零，因为子进程对管道的写端有一个打开的句柄。

For an example program that uses anonymous pipes to redirect the standard handles of a child process, see [Creating a Child Process with Redirected Input and Output](https://docs.microsoft.com/windows/desktop/ProcThread/creating-a-child-process-with-redirected-input-and-output).

有关使用匿名管道重定向子进程的标准句柄的示例程序，请参见使用[Creating a Child Process with Redirected Input and Output](https://docs.microsoft.com/windows/desktop/ProcThread/creating-a-child-process-with-redirected-input-and-output)

# Anonymous Pipe Security and Access Rights（匿名管道安全和访问权限）

Windows security enables you to control access to anonymous pipes. For more information about security, see [Access-Control Model](https://docs.microsoft.com/windows/desktop/SecAuthZ/access-control-model).

Windows安全性允许您控制对匿名管道的访问。有关安全性的更多信息，请参见[Access-Control Model](https://docs.microsoft.com/windows/desktop/SecAuthZ/access-control-model).

You can specify a [security descriptor](https://docs.microsoft.com/windows/desktop/SecAuthZ/security-descriptors) for a pipe when you call the [**CreatePipe**](https://msdn.microsoft.com/en-us/library/Aa365152(v=VS.85).aspx) function. The security descriptor controls access to both the read and write ends of the pipe. If you specify **NULL**, the pipe gets a default security descriptor. The ACLs in the default security descriptor for a pipe come from the primary or impersonation token of the creator.

在调用CreatePipe函数时，可以为管道指定安全描述符。安全描述符控制对管道的读写两端的访问。如果指定NULL，管道将获得默认的安全描述符。管道的默认安全描述符中的acl来自创建者的主或模拟令牌。

To retrieve a pipe's security descriptor, call the [**GetSecurityInfo**](https://docs.microsoft.com/windows/desktop/api/aclapi/nf-aclapi-getsecurityinfo) function. To change a pipe's security descriptor, call the [**SetSecurityInfo**](https://docs.microsoft.com/windows/desktop/api/aclapi/nf-aclapi-setsecurityinfo) function.

要检索管道的安全描述符，请调用GetSecurityInfo函数。要更改管道的安全描述符，请调用SetSecurityInfo函数

The [**CreatePipe**](https://msdn.microsoft.com/en-us/library/Aa365152(v=VS.85).aspx) function returns two handles to the anonymous pipe: a read handle with GENERIC\_READ and SYNCHRONIZE access; and a write handle with GENERIC\_WRITE and SYNCHRONIZE access. GENERIC\_READ and GENERIC\_WRITE access use the same access rights mapping as for named pipes.

CreatePipe函数向匿名管道返回两个句柄:一个具有GENERIC\_READ和同步访问权限的读句柄;和一个具有GENERIC\_WRITE和同步访问的写句柄。GENERIC\_READ和GENERIC\_WRITE访问使用与命名管道相同的访问权限映射。

GENERIC\_READ access for an anonymous pipe combines the rights to read data from the pipe, read pipe attributes, read extended attributes, and read the pipe's DACL.

匿名管道的GENERIC\_READ访问组合了从管道中读取数据、读取管道属性、读取扩展属性和读取管道的DACL的权限。

GENERIC\_WRITE access for an anonymous pipe combines the rights to write data to the pipe, append data to it, write pipe attributes, write extended attributes, and read the pipe's DACL.

匿名管道的GENERIC\_WRITE访问组合了向管道写入数据、向管道追加数据、写入管道属性、写入扩展属性和读取管道的DACL的权限。

# [Creating a Child Process with Redirected Input and Output](https://docs.microsoft.com/windows/desktop/ProcThread/creating-a-child-process-with-redirected-input-and-output)

The example in this topic demonstrates how to create a child process using the [**CreateProcess**](https://msdn.microsoft.com/en-us/library/ms682425(v=VS.85).aspx) function from a console process. It also demonstrates a technique for using anonymous pipes to redirect the child process's standard input and output handles. Note that named pipes can also be used to redirect process I/O.

本主题中的示例演示如何使用控制台流程中的CreateProcess函数创建子流程。它还演示了使用匿名管道重定向子进程的标准输入和输出句柄的技术。注意，命名管道也可以用来重定向进程I/O

The [**CreatePipe**](https://docs.microsoft.com/windows/desktop/api/namedpipeapi/nf-namedpipeapi-createpipe) function uses the [**SECURITY\_ATTRIBUTES**](https://docs.microsoft.com/previous-versions/windows/desktop/legacy/aa379560(v=vs.85)) structure to create inheritable handles to the read and write ends of two pipes. The read end of one pipe serves as standard input for the child process, and the write end of the other pipe is the standard output for the child process. These pipe handles are specified in the [**STARTUPINFO**](https://msdn.microsoft.com/en-us/library/ms686331(v=VS.85).aspx) structure, which makes them the standard handles inherited by the child process.

CreatePipe函数使用SECURITY\_ATTRIBUTES结构创建两个管道的读写端可继承句柄。一个管道的读端作为子进程的标准输入，另一个管道的写端作为子进程的标准输出。这些管道句柄在STARTUPINFO结构中指定，这使它们成为子进程继承的标准句柄。

The parent process uses the opposite ends of these two pipes to write to the child process's input and read from the child process's output. As specified in the [**STARTUPINFO**](https://msdn.microsoft.com/en-us/library/ms686331(v=VS.85).aspx) structure, these handles are also inheritable. However, these handles must not be inherited. Therefore, before creating the child process, the parent process uses the [**SetHandleInformation**](https://docs.microsoft.com/windows/desktop/api/handleapi/nf-handleapi-sethandleinformation) function to ensure that the write handle for the child process's standard input and the read handle for the child process's standard output cannot be inherited. For more information, see [Pipes](https://docs.microsoft.com/windows/desktop/ipc/pipes).

父进程使用这两个管道的相反两端来写入子进程的输入，并从子进程的输出中读取。正如在STARTUPINFO结构中指定的，这些句柄也是可继承的。但是，这些句柄不能被继承。因此，在创建子进程之前，父进程使用SetHandleInformation函数来确保不能继承子进程标准输入的写句柄和子进程标准输出的读句柄。有关更多信息，请参见[Pipes](https://docs.microsoft.com/windows/desktop/ipc/pipes).。

The following is the code for the parent process. It takes a single command-line argument: the name of a text file.

|  |
| --- |
| C++  #include <windows.h>  #include <tchar.h>  #include <stdio.h>  #include <strsafe.h>  #define BUFSIZE 4096    HANDLE g\_hChildStd\_IN\_Rd = NULL;  HANDLE g\_hChildStd\_IN\_Wr = NULL;  HANDLE g\_hChildStd\_OUT\_Rd = NULL;  HANDLE g\_hChildStd\_OUT\_Wr = NULL;  HANDLE g\_hInputFile = NULL;    void CreateChildProcess(void);  void WriteToPipe(void);  void ReadFromPipe(void);  void ErrorExit(PTSTR);    int \_tmain(int argc, TCHAR \*argv[])  {  SECURITY\_ATTRIBUTES saAttr;    printf("\n->Start of parent execution.\n");  // Set the bInheritHandle flag so pipe handles are inherited.    saAttr.nLength = sizeof(SECURITY\_ATTRIBUTES);  saAttr.bInheritHandle = TRUE;  saAttr.lpSecurityDescriptor = NULL;  // Create a pipe for the child process's STDOUT.    if ( ! CreatePipe(&g\_hChildStd\_OUT\_Rd, &g\_hChildStd\_OUT\_Wr, &saAttr, 0) )  ErrorExit(TEXT("StdoutRd CreatePipe"));  // Ensure the read handle to the pipe for STDOUT is not inherited.  if ( ! SetHandleInformation(g\_hChildStd\_OUT\_Rd, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdout SetHandleInformation"));  // Create a pipe for the child process's STDIN.    if (! CreatePipe(&g\_hChildStd\_IN\_Rd, &g\_hChildStd\_IN\_Wr, &saAttr, 0))  ErrorExit(TEXT("Stdin CreatePipe"));  // Ensure the write handle to the pipe for STDIN is not inherited.    if ( ! SetHandleInformation(g\_hChildStd\_IN\_Wr, HANDLE\_FLAG\_INHERIT, 0) )  ErrorExit(TEXT("Stdin SetHandleInformation"));    // Create the child process.    CreateChildProcess();  // Get a handle to an input file for the parent.  // This example assumes a plain text file and uses string output to verify data flow.    if (argc == 1)  ErrorExit(TEXT("Please specify an input file.\n"));  g\_hInputFile = CreateFile(  argv[1],  GENERIC\_READ,  0,  NULL,  OPEN\_EXISTING,  FILE\_ATTRIBUTE\_READONLY,  NULL);  if ( g\_hInputFile == INVALID\_HANDLE\_VALUE )  ErrorExit(TEXT("CreateFile"));    // Write to the pipe that is the standard input for a child process.  // Data is written to the pipe's buffers, so it is not necessary to wait  // until the child process is running before writing data.    WriteToPipe();  printf( "\n->Contents of %s written to child STDIN pipe.\n", argv[1]);    // Read from pipe that is the standard output for child process.    printf( "\n->Contents of child process STDOUT:\n\n", argv[1]);  ReadFromPipe();  printf("\n->End of parent execution.\n");  // The remaining open handles are cleaned up when this process terminates.  // To avoid resource leaks in a larger application, close handles explicitly.  return 0;  }    void CreateChildProcess()  // Create a child process that uses the previously created pipes for STDIN and STDOUT.  {  TCHAR szCmdline[]=TEXT("child");  PROCESS\_INFORMATION piProcInfo;  STARTUPINFO siStartInfo;  BOOL bSuccess = FALSE;    // Set up members of the PROCESS\_INFORMATION structure.    ZeroMemory( &piProcInfo, sizeof(PROCESS\_INFORMATION) );    // Set up members of the STARTUPINFO structure.  // This structure specifies the STDIN and STDOUT handles for redirection.    ZeroMemory( &siStartInfo, sizeof(STARTUPINFO) );  siStartInfo.cb = sizeof(STARTUPINFO);  siStartInfo.hStdError = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdOutput = g\_hChildStd\_OUT\_Wr;  siStartInfo.hStdInput = g\_hChildStd\_IN\_Rd;  siStartInfo.dwFlags |= STARTF\_USESTDHANDLES;    // Create the child process.    bSuccess = CreateProcess(NULL,  szCmdline, // command line  NULL, // process security attributes  NULL, // primary thread security attributes  TRUE, // handles are inherited  0, // creation flags  NULL, // use parent's environment  NULL, // use parent's current directory  &siStartInfo, // STARTUPINFO pointer  &piProcInfo); // receives PROCESS\_INFORMATION    // If an error occurs, exit the application.  if ( ! bSuccess )  ErrorExit(TEXT("CreateProcess"));  else  {  // Close handles to the child process and its primary thread.  // Some applications might keep these handles to monitor the status  // of the child process, for example.  CloseHandle(piProcInfo.hProcess);  CloseHandle(piProcInfo.hThread);  }  }    void WriteToPipe(void)  // Read from a file and write its contents to the pipe for the child's STDIN.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;    for (;;)  {  bSuccess = ReadFile(g\_hInputFile, chBuf, BUFSIZE, &dwRead, NULL);  if ( ! bSuccess || dwRead == 0 ) break;    bSuccess = WriteFile(g\_hChildStd\_IN\_Wr, chBuf, dwRead, &dwWritten, NULL);  if ( ! bSuccess ) break;  }    // Close the pipe handle so the child process stops reading.    if ( ! CloseHandle(g\_hChildStd\_IN\_Wr) )  ErrorExit(TEXT("StdInWr CloseHandle"));  }    void ReadFromPipe(void)  // Read output from the child process's pipe for STDOUT  // and write to the parent process's pipe for STDOUT.  // Stop when there is no more data.  {  DWORD dwRead, dwWritten;  CHAR chBuf[BUFSIZE];  BOOL bSuccess = FALSE;  HANDLE hParentStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);  for (;;)  {  bSuccess = ReadFile( g\_hChildStd\_OUT\_Rd, chBuf, BUFSIZE, &dwRead, NULL);  if( ! bSuccess || dwRead == 0 ) break;  bSuccess = WriteFile(hParentStdOut, chBuf,  dwRead, &dwWritten, NULL);  if (! bSuccess ) break;  }  }    void ErrorExit(PTSTR lpszFunction)  // Format a readable error message, display a message box,  // and exit from the application.  {  LPVOID lpMsgBuf;  LPVOID lpDisplayBuf;  DWORD dw = GetLastError();  FormatMessage(  FORMAT\_MESSAGE\_ALLOCATE\_BUFFER |  FORMAT\_MESSAGE\_FROM\_SYSTEM |  FORMAT\_MESSAGE\_IGNORE\_INSERTS,  NULL,  dw,  MAKELANGID(LANG\_NEUTRAL, SUBLANG\_DEFAULT),  (LPTSTR) &lpMsgBuf,  0, NULL );  lpDisplayBuf = (LPVOID)LocalAlloc(LMEM\_ZEROINIT,  (lstrlen((LPCTSTR)lpMsgBuf)+lstrlen((LPCTSTR)lpszFunction)+40)\*sizeof(TCHAR));  StringCchPrintf((LPTSTR)lpDisplayBuf,  LocalSize(lpDisplayBuf) / sizeof(TCHAR),  TEXT("%s failed with error %d: %s"),  lpszFunction, dw, lpMsgBuf);  MessageBox(NULL, (LPCTSTR)lpDisplayBuf, TEXT("Error"), MB\_OK);  LocalFree(lpMsgBuf);  LocalFree(lpDisplayBuf);  ExitProcess(1);  } |
|  |

The following is the code for the child process. It uses the inherited handles for STDIN and STDOUT to access the pipe created by the parent. The parent process reads from its input file and writes the information to a pipe. The child receives text through the pipe using STDIN and writes to the pipe using STDOUT. The parent reads from the read end of the pipe and displays the information to its STDOUT.

|  |
| --- |
| C++  #include <windows.h>  #include <stdio.h>  #define BUFSIZE 4096    int main(void)  {  CHAR chBuf[BUFSIZE];  DWORD dwRead, dwWritten;  HANDLE hStdin, hStdout;  BOOL bSuccess;    hStdout = GetStdHandle(STD\_OUTPUT\_HANDLE);  hStdin = GetStdHandle(STD\_INPUT\_HANDLE);  if (  (hStdout == INVALID\_HANDLE\_VALUE) ||  (hStdin == INVALID\_HANDLE\_VALUE)  )  ExitProcess(1);    // Send something to this process's stdout using printf.  printf("\n \*\* This is a message from the child process. \*\* \n");  // This simple algorithm uses the existence of the pipes to control execution.  // It relies on the pipe buffers to ensure that no data is lost.  // Larger applications would use more advanced process control.  for (;;)  {  // Read from standard input and stop on error or no data.  bSuccess = ReadFile(hStdin, chBuf, BUFSIZE, &dwRead, NULL);    if (! bSuccess || dwRead == 0)  break;    // Write to standard output and stop on error.  bSuccess = WriteFile(hStdout, chBuf, dwRead, &dwWritten, NULL);    if (! bSuccess)  break;  }  return 0;  } |