|  |  |
| --- | --- |
| **stdout file pointer**  Output data to a screen, file, or elsewhere.  A **FILE\***, called a "file pointer," is a pointer to a FILE structure that allows programs to read and write to files. It is included in <stdio.h>  The FILE structure maintains the information needed to access files.  The FILE structure typically maintains an output buffer that temporarily stores characters until the system copies those characters to disk or screen.  **stdout** is a predefined FILE\* that is pre-associated with a system's standard output, usually a computer screen. Look at the animation on ZyBooks section 10.1 | /\* fprintf function – file print  First argument - the FILE\* to the file being written  Second argument – string & format specifiers  Remaining arguments – expressions for specifiers  \*/  fprintf(stdout, "Num");  fprintf(stdout, "%d", 9); |
| **stdin file pointer**  Input data taken from a keyboard, touchscreen, file, or elsewhere.  The FILE structure typically maintains an input buffer that temporarily stores characters until the system reads characters from the data buffer up to the next whitespace, converts to the target variable's data type, and stores the result into the variable.  **stdin** is a predefined FILE\* that is pre-associated with a system's standard input, usually a computer keyboard. Look at the animation on ZyBooks section 10.2 | /\* fscanf function – file scan  First argument - the FILE\* to the file being read  Second argument – format specifiers  Remaining arguments – location/variable to store values  \*/  fscanf(stdin, "%s", firstName);  fscanf(stdin, "%d", &studentID); |
| **Flushing output**  Printing characters from the buffer to the output device (e.g., screen) requires a time-consuming reservation of processor resources; once those resources are reserved, moving characters is fast, whether there is 1 character or 50 characters to print.  As such, the system may wait until the buffer is full, or at least has a certain number of characters before moving them to the output device.  Or, with fewer characters in the buffer, the system may wait until the resources are not busy.  However, sometimes a programmer does not want the system to wait. For example, in a very processor-intensive program, such waiting could cause delayed and/or jittery output. The programmer can use the function fflush(). The fflush() function will immediately flush the contents of the buffer for the specified FILE\*. | fflush(stdout);  /\* The above function will write the contents of the buffer for stdout to the computer screen. \*/ |
| **File Input**  Get input from another file, rather than the predefined input file stdin that comes from the standard input (keyboard).  // File variable  FILE\* f;  // Open file for reading  f = fopen("myfile.txt", "r");  // First arg – file name  // Second arg – file mode  // If fail, return NULL  // Read value of N from file  fscanf(f, “%d”, &N);  // Close file  fclose(f);  // Check for end of the file while reading  feof(f);  // Returns 1 if the previous fscanf reached the end of the file. Use this in the while loop as a condition.  while(i<N && feof(f)!=1) | /\* Read an array from file “testIn.txt”  5  10  20  30  40  50  \*/  #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  FILE\* inFile = NULL;  int\* arr;  int N;  int i;  inFile = fopen(“testIn.txt”, “r”);    if (inFile == NULL)  {  printf(“Could not open file testIn.txt.\n”);  return -1; // -1 indicates error  }  fscanf(inFile, “%d”, &N);  arr = (int\*)malloc(sizeof(int)\*N);  if (arr == NULL)  {  fclose(inFile);  return -1;  }  i = 0;  while (i < N)  fscanf(inFile, “%d”, &(arr[i++]));  fclose(inFile);  printf("Numbers: ");  i = 0;  while (i < N)  printf("%d ", arr[i++]);  printf("\n");  return 0;  } |
| **File Output**  Print output to another file, rather than the predefined output file stdout that goes to the standard output (screen).  // File variable  FILE\* f;  // Open file for writing  f = fopen("myfile.txt", "w");  // First arg – file name  // Second arg – file mode  // If fail, return NULL  // Read value of N from file  fscanf(f, “%d”, &N);  // Close file  fclose(f); | /\* Write an array to file “testOut.txt”  5  10  20  30  40  50  \*/    #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  FILE\* outFile = NULL;  int N=5;  int arr[N] = {10,20,30,40,50};  int i;  outFile = fopen(“testOut.txt”, “w”);  if (outFile == NULL)  {  printf(“Could not open file testOut.txt.\n”);  return -1; // -1 indicates error  }  fprintf(outFile, “%d\n”, N);  i = 0;  while (i < N)  fprintf(outFile, “%d\n”, arr[i++]);  fclose(outFile);  return 0;  } |
| **Binary File read/write**  Read or write files directly in binary format. This is not easily readable by humans. But it is extremely efficient since no conversion from its raw form to human readable form needs to be performed. Also, binary files are more space efficient.  Read/write mode in fopen - “rb” & “wb”  fread & fwrite instead of fscanf & fprintf | // Store an integer 12345  // Size of integer??  // Binary file size??  // Standard file size? |
| **Command Line Arguments**  Get the name of the file directly in the command line while running the program. So, it can be used again and again without changing the code.  Add arguments when running the compiled executable program. Write all arguments after the executable name, separated by space.  ./execName arg1 arg2 arg3 ...  // Prints argc and argv values  printf(“argc: %d\n”, argc);  for (i = 0; i < argc; i++)  {  printf(“argv[%d]: %s\n”, i, argv[i]);  } | /\* main function definition \*/  int main(int argc, char \*argv[])  /\* Consider you compile as follows \*/  gcc test.c –o testEXE  ./testEXE  // argc = 1  // argv[0] = ./testEXE  // Add another argument at the end e.g. filename  ./testEXE testOut.txt  // argc = 2  // argv[0] = ./testEXE  // argv[1] = testOut.txt  #include <stdio.h>  #include <stdlib.h>  int main(int argc, char \*argv[])  {  if(argc != 2)  {  printf(“Usage: ./execName outputFileName\n”);  return -1; // -1 indicates error  }  FILE\* outFile = NULL;  int N=5;  int arr[N] = {10,20,30,40,50};  int i=0;    outFile = fopen(argv[1], “w”);  if (outFile == NULL)  {  printf(“Could not open file testOut.txt.\n”);  return -1; // -1 indicates error  }  fprintf(outFile, “%d\n”, N);    while (i < N)  fprintf(outFile, “%d\n”, arr[i++]);    fclose(outFile);  return 0;  } |