Functions with Variable Number of Parameters

- Must include **`<stdarg.h>`**
- When declaring the function there must be
  - at least one fixed parameter
  - and an ellipsis (...)
- One of the fixed parameters must tell the function how many parameters are passed in (...)
- The function must know the type of each argument in the variable list
  - The example will have a single type so there should not be a problem
  - Otherwise you'll have to use something like **`printf`**'s approach

Retrieving Arguments

- **`stdarg.h`** has four macros:
  - **`va_list`** :
    - a pointer data type
  - **`va_start()`** :
    - a macro used to initialize the argument list
  - **`va_arg()`** :
    - a macro used to retrieve each argument in list
  - **`va_end()`** :
    - a macro used to clean up when all retrieved

Retrieving Arguments (Steps)

- Each of the following steps are required in the function
  - Declare a pointer of type **`va_list`**.
    - used to access individual arguments (**`arg_ptr`**)
  - Call **`va_start()`**
    - pass it the **`arg_ptr`** and the name of the last fixed argument
    - no value is returned
    - it initializes **`arg_ptr`** to the first variable argument
  - Call **`va_arg()`** passing **`arg_ptr`** and type of next argument
    - returns “value” of next argument
    - if **`n`** arguments are present this can be called **`n`** times
    - all are retrieved in order
  - Call **`va_end()`**
    - pass the **`arg_ptr`**

An Example

```
#include <stdio.h>
#include <stdarg.h>
float average (int num, ...);
int main ()
{
  float x;
  x = average (10, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10);
  printf ("The first average is %f.\n", x);
  x = average (5, 121, 206, 76, 31, 5);
  printf ("The second average is %f.\n", x);
  return 0;
}

float average (int num, ...)
{
  va_list arg_ptr;
  int count, total = 0;
  va_start (arg_ptr, num);

  for (count = 0; count < num; count++)
    total += va_arg (arg_ptr, int);

  va_end (arg_ptr);

  return ((float) total / num);
}
```

string.h

- **#include <string.h>**
- Functions of two families:
  - Memory functions - **mem**... **()**
    - Manipulating blocks of memory of specified size.
    - Can be thought of as array of bytes
  - String functions – **str**... **()**
    - Manipulating null terminated strings.

`string.h` - Memory Handling Functions

- **`void *memset(void *p, int c, size_t n)`**
  - Fills the first **`n`** bytes of the memory area pointed to by **`p`** with the constant byte **`c`**.
- **`void *memcpy(void *to, const void *from, size_t n)`**
  - Copies **`n`** bytes from memory area **`from`** to memory area **`to`**. The memory areas may not overlap.
- **`void *memmove(void *to, const void *from, size_t n)`**
  - Same as above, can handle also overlapping blocks.

memmove example

```
#include <stdio.h>
#include <string.h>
int
main ()
{
 char x[] = "Home Sweet Home";

 printf ("%s%s\n", "The string in array x before memmove is: ", x);
 printf ("%s%s\n",
     "The string in array x after memmove is: ",
     (char*)memmove(x, &x[5], 10));
 return 0;
}
```

```
The string in array x before memmove is: Home Sweet Home
The string in array x after memmove is: Sweet Home Home
```

`string.h` - Memory Handling Functions

- `int memcmp(const void *p, const void *q, size_t n)`
  - Lexicographic comparison of byte sequences
  - Return value is less than, equal to, or greater than zero according to comparison
- `void *memchr(const void *p, int c, size_t n )`
  - Scans the first `n` bytes of the memory area pointed to by `s` for the character `c`. The first byte to match `c` (interpreted as an unsigned character) stops the operation. Returns a pointer to the matching byte or `NULL` if the character does not occur in the given memory area.

memchr example

```
#include <string.h>
#include <stdio.h>

int
main ()
{
 char s[] = "This is a string";

 printf ("%s%s\n", "The remainder of s after character 'r' is found is: ",
     (char *) memchr (s, 'r', 16));
 return 0;
}
```

```
The remainder of s after character 'r' is found is: ring
```

`string.h` - String Handling Functions

- **`size_t strlen(const char *s)`**
  - Return the length of the string **`s`**, not counting the terminating null character
- **`char *strcpy(char *s1, const char *s2)`**
  - Copies the string pointed to by **`s2`** (including the terminating `\0' character) to the array pointed to by **`s1`**. The strings may not overlap, and the destination string **`s1`** must be large enough to receive the copy.
- **`char *strncpy(char *s1, const char *s2, size_t n)`**
  - Similar, except that not more than **`n`** bytes of **`s2`** are copied. Thus, if there is no null byte among the first **`n`** bytes of **`s2`**, the result will not be null-terminated.

`string.h` - String Handling Functions

- **`char *strcat(char *s1, const char *s2)`**
- **`char *strncat(char *s1, const char *s2, size_t n)`**
  - **`strcat()`** appends the **`s2`** string to the **`s1`** string overwriting the `\0' character at the end of **`s1`**, and then adds a terminating `\0' character. The strings may not overlap, and the dest string must have enough space for the result.
  - **`strncat()`** is similar, except that only the first **`n`** characters of **`s2`** are appended to **`s1`**.
- **`int strcmp(const char *s1, const char *s2)`**
- **`int strncmp(const char *s1, const char *s2, size_t n)`**
  - Lexicographic comparison
  - Return value is less than, equal to, or greater than zero according comparison
  - **`strncmp()`** compares at most **`n`** characters

strcat/strncat example

```
#include <stdio.h>
#include <string.h>

int
main ()
{
 char s1[16] = "Happy ";
 char s2[] = "New Year ";
 char s3[22] = "";

 printf ("s1 = %s\ns2 = %s\n", s1, s2);
 printf ("strcat( s1, s2 ) = %s\n", strcat (s1, s2));
 printf ("strncat( s3, s1, 6 ) = %s\n", strncat (s3, s1, 6));
 printf ("strcat( s3, s1 ) = %s\n", strcat (s3, s1));
 return 0;
}
```

```
s1 = Happy
s2 = New Year
strcat( s1, s2 ) = Happy New Year
strncat( s3, s1, 6 ) = Happy
strcat( s3, s1 ) = Happy Happy New Year
```

strtok

- **`char *strtok(char *s1, const char *s2)`**
  - Initialization call (with **`s1`**) and than consecutive calls with **`NULL`**
  - Returns the next token in each call
  - Destroys the original string

```
#include <stdio.h>
#include <string.h>
int
main ()
{
 char string[] =
 "This is a sentence with 7 tokens";
 char *tokenPtr;
 printf ("%s\n%s\n\n%s\n",
     "The string to be tokenized is:"
     , string, "The tokens are:");
 tokenPtr = strtok (string, " ");
 while (tokenPtr != NULL)
  {
   printf ("%s\n", tokenPtr);
   tokenPtr = strtok (NULL, " ");
  }
 return 0;
}
```

```
The string to be tokenized is:
This is a sentence with 7 tokens

The tokens are:
This
is
a
sentence
with
7
tokens
```

`string.h` - String Search Functions

- `char *strchr(const char *s, int c)`
- `char *strrchr(const char *s, int c)`
  - `strchr()` returns a pointer to the first occurrence of the character `c` in the string `s`.
  - `strrchr()` returns a pointer to the last occurrence of the character `c` in the string `s`.
  - Both return `NULL` if the character is not found.
- `size_t strspn(const char *s, const char *accept)`
- `size_t strcspn(const char *s, const char *reject)`
  - `strspn()` returns the number of characters in the initial segment of `s` which consist only of characters from `accept`.
  - `strcspn()` returns the number of characters in the initial segment of `s` which are not in the string `reject`.

strspn Example

```
#include <stdio.h>
#include <string.h>

int
main ()
{
 const char *string1 = "The value is 3.14159";
 const char *string2 = "aehi lsTuv";

 printf ("%s%s\n%s%s\n\n%s\n%s%u\n",
     "string1 = ", string1, "string2 = ", string2,
     "The length of the initial segment of string1",
     "containing only characters from string2 = ",
     strspn (string1, string2));
 return 0;
}
```

```
string1 = The value is 3.14159
string2 = aehi lsTuv

The length of the initial segment of string1
containing only characters from string2 = 13
```

`string.h` - String Search Functions

- **`char *strpbrk(const char *s,`**
  `const char *accept)`
  - Returns a pointer to the character in **`s`** that matches one of the characters in accept, or **`NULL`** if no such character is found.
- **`char *strstr(const char *haystack,`**
  `const char *needle)`
  - Finds the first occurrence of the substring **`needle`** in the string **`haystack`** Returns a pointer to the beginning of the substring, or **`NULL`** if the substring is not found.

String Conversion Functions

- Convert to numeric values, searching, comparison
- **`<stdlib.h>`**
- Most functions take **`const char *`**
- Do not modify string

String Conversion Functions

- **`double atof( const char *nPtr )`**
  - Converts string to floating point number (double)
  - Returns 0 if cannot be converted
- **`int atoi( const char *nPtr )`**
  - Converts string to integer
  - Returns 0 if cannot be converted
- **`long atol( const char *nPtr )`**
  - Converts string to long integer
  - If **`int`** and **`long`** have the same size, then **`atoi`** and **`atol`** identical

The Standard C Library Error Handling

- Most library functions return a special value to indicate that they have failed. The special value is typically -1, a null pointer, or a constant such as EOF that is defined for that purpose.
- To find out what kind of error it was, you need to look at the error code stored in the variable **`errno`**. This variable is declared in the header file **`<errno.h>`**
  - The initial value of **`errno`** at program startup is zero.
  - Many library functions are guaranteed to set it to certain nonzero values when they encounter certain kinds of errors. These error conditions are listed for each function.
  - These functions do not change **`errno`** when they succeed; thus, the value of **`errno`** after a successful call is not necessarily zero,
    - You should not use **`errno`** to determine whether a call failed.

perror

- You can use the **perror** function to inform user about error conditions
  - **void perror(const char *s)**
- It produces a message on the standard error output, describing the error contained in the **errno** variable. The argument string **s** is printed first, then a colon and a blank, then the message and a new-line.

```
#include <stdio.h>

int
main ()
{
  FILE *pFile;
  pFile = fopen ("unexist.ent", "rb");
  if (pFile == NULL)
    perror ("This error has occurred");
  else
    fclose (pFile);
  return 0;
}
```

```
This error has occurred: No such file or directory
```

String Conversion Functions

- **`double strtod(const char *nPtr, char **endPtr)`**
  - Converts first argument to **`double`**, returns that value
  - Sets second argument to location of first character after converted portion of string
  - If no conversion is performed, zero is returned and the value of **`nPtr`** is stored in the location referenced by **`endPtr`**
  - If the correct value would cause overflow or underflow, **`errno`** is set to **`ERANGE`**

  - strtod("123.4this is a test", &stringPtr);
    - Returns 123.4
    - stringPtr points to "this is a test"

strtod

```
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

int main (int argc, char *argv[])
{
  char *stringPtr;
  double v;
  int i;

  for (i = 1; i < argc; i++)
    {
      v = strtod (argv[i], &stringPtr);
      if (errno == ERANGE)
        printf ("Argument %d is an invalid number\n", i);
      else if (argv[i] == stringPtr)
        printf ("Argument %d is not a number\n", i);
      else
        printf ("Argument %d is a number %f\n", i, v);
    }
  return 0;
}
```

```
./strtod She is 17 now 1e123131231
Argument 1 is not a number
Argument 2 is not a number
Argument 3 is a number 17.000000
Argument 4 is not a number
Argument 5 is an invalid number
```

String Conversion Functions

- **`long strtol(const char *nPtr, char **endPtr, int base )`**
  - Converts first argument to long, returns that value
  - Sets second argument to location of first character after converted portion of string
  - Third argument is base of value being converted
    - Any number 2 - 36
    - 0 specifies octal, decimal, or hexadecimal
  - On overflow or underflow, **`errno`** is set to **`ERANGE`**
- **`unsigned long int strtoul(const char *nptr, char **endptr, int base)`**
  - As above, with unsigned long

File Handling in C

- Storage of data in variables, arrays or dynamic data structures is temporary
  - when the program terminates all data is lost
- Files are used for the retention of large amounts of data
- ASCII or text data:
  - Character ‘1’ ‘2’ ‘3’
  - ASCII Decimal value 49 50 51
  - ASCII Binary value 00110011 00110010 00110011
- Binary Data: The decimal number 123 could be stored as the binary number 01111011, less storage than ASCII.

Binary files

- Binary files are very similar to arrays of structures, except the structures are in a disk file rather than in an array in memory. Because the structures in a binary file are on disk, you can create very large collections of them (limited only by your available disk space). They are also permanent and always available. The only disadvantage is the slowness that comes from disk access time.

Binary files

- Binary files have two features that distinguish them from text files:
  - You can jump instantly to any structure in the file, which provides random access as in an array; and you can change the contents of a structure anywhere in the file at any time.
  - Binary files usually have faster read and write times than text files, because a binary image of the record is stored directly from memory to disk (or vice versa). In a text file, everything has to be converted back and forth to text, and this takes time.

Streams

- All input and output in C is performed with streams
- Up to now - a stream is a sequence of characters organized into lines (text stream)
  - each line consists of 0 or more characters and ends with the newline character ‘\n’
- Streams provide communications channels between files and programs
  - A stream can be connected to a file by opening it and the connection is broken by closing the stream.

Streams

- When program execution begins, three files and their associated streams are connected to the program automatically
  - the standard input stream
  - the standard output stream
  - the standard error stream

Streams

- Standard input is connected to the keyboard
  - it enables a program to read data from the keyboard
- Standard output is connected to the screen
  - it enables a program to write data to the screen
- Standard error is connected to the screen
  - all error messages are output to standard error
- Operating systems often allow these streams to be redirected to other devices

Streams

- Files are accessed via a pointer to a **`FILE`** structure

- The **`FILE`** structure
  - Contains information used to process a file
  - Is declared in **`<stdio.h>`**
  - You don't need to worry about the details of the structure.
  - In fact it may vary from system to system.

- Standard input, standard output and standard error are manipulated with file pointers **`stdin`**, **`stdout`** and **`stderr`**

Opening Files

- A file must be opened before its contents can be accessed
- To open a file
  - `FILE *fopen (const char *name, const char *mode)`

- **`fopen`** accepts two arguments
  - **`name`**: a string containing the name of the file
  - **`mode`**: a string containing the file open mode or the type of access that the file is opened for

- **`fopen`** returns a file pointer which is needed for all future access to the file

File I/O

```
FILE* fopen( const char* name,const char* mode);
```

- Mode can be (from man page)
  - r Open file for reading
  - w Truncate to zero length or create file for writing
  - a Append; open or create file for writing at end-of-file
  - r+ Open file for update (reading and writing)
  - w+ Truncate to zero length or create file for update
  - a+ Append; open or create file for update, writing at end-of-file
  - b Can be added to the modes above to indicate binary files

fopen modes

mode	open stream for read	open stream for write	truncate file	create file	starting position
"r"	y	n	n	n	beginning
"r+"	y	y	n	n	beginning
"w"	n	y	y	y	beginning
"w+"	y	y	y	y	beginning
"a"	n	y	n	y	end-of-file
"a+"	y	y	n	y	end-of-file

To read the first line, "r" will open a stream for read, the stream will not be opened for write, will not truncate the file to zero length, will not create the file if it doesn't already exist and will be positioned at the beginning of the stream.

Closing Files

- A file must be closed after use
- To close a file use the function
  - `int fclose (FILE *fptr)`
- **`fclose`** takes one argument
  - the file pointer to the file to be closed
- **`fclose`** returns zero if no errors occur and returns **`EOF`** otherwise

Unformatted Text Files

- **`<stdio.h>`** provides many functions for reading and writing to unformatted text files
  - **`int fgetc (FILE *stream)`**
    - to read the next character from the stream
    - returns the character (as an integer) or **`EOF`** if end of file or an error occurs
  - **`int fputc (int c, FILE *stream)`**
    - writes the character **`c`** to the stream
    - returns the character or **`EOF`** for error

Copy File Example

```
#include <stdio.h>

int
main (int argc, char *argv[])
{
  FILE *ifp, *ofp;
  int c;
  if (argc != 3) /*check for valid input */
    printf ("Incorrect number of parameters");
  else if ((ifp = fopen (*++argv, "r")) == NULL)
    printf ("Cannot open %s for reading \n", *argv);
  else if ((ofp = fopen (*++argv, "w")) == NULL)
    printf ("Cannot open %s for writing \n", *argv);
  else
    while ((c = fgetc (ifp)) != EOF)
      fputc (c, ofp);
  fclose (ifp);
  fclose (ofp);
  return 0;
}
```

Unformatted Text Files

- **`char* fgets (char *s,int n,FILE *stream)`**
  - reads at most the next **`n-1`** characters from the stream into the array **`s`**. Reading stops after an **`EOF`** or a newline. If a newline is read, it is stored into the buffer. A **`'\0'`** is stored after the last character in the buffer.
  - returns **`s`** or **`NULL`** if **`EOF`** or error occurs

- **`int fputs (const char *s, FILE *stream)`**
  - write the string **`s`** to the stream,
  - returns **`EOF`** for error

Unformatted Text

- These file handling functions can be used instead of reading and writing characters and strings from the keyboard and to the screen
  - **`int fgetc (FILE *stream)`**
    - is similar to **`getchar()`**
  - **`int fputc (int c, FILE *stream)`**
    - is similar to **`putchar(int)`**
  - **`char *fgets (char *s,int n,FILE *stream)`**
    - is similar to **`gets(char* )`** (never use **`gets()`**!)
  - **`int fputs (const char *s, FILE *stream)`**
    - is similar to **`puts(const char*)`**

Convert File to Uppercase Example

```
#include <stdio.h>
#include <ctype.h>
#include <stdlib.h>

int
main ()
{
  char szLine[1000];
  char *p = 0;
  FILE *fileIn, *fileOut;
  fileIn = fopen ("mary.txt", "r");
  if (!fileIn) {
      fprintf (stderr, "Cannot open input file\n");
      exit (1); };
  fileOut = fopen ("mary.new", "w");
  if (!fileOut) {
      fprintf (stderr, "Cannot open output file\n");
      exit (2); };
  while (fgets (szLine, 1000, fileIn))
    {
      for (p = szLine; *p; ++p)
        *p = toupper (*p);
      fputs (szLine, fileOut);
    }
  fclose (fileOut);
  fclose (fileIn);
  return 0;
}
```

Formatted Text File I/O

- For formatted input and output to files use the following functions:
  - `int fprintf (FILE *fptr, const char* fmt,...)`
  - `int fscanf (FILE *fptr, const char* fmt, ... )`
- These functions are the same as **`printf`** and **`scanf`** except that the output is directed to the stream accessed by the file pointer, **`fptr`**

`feof` Function

- The function **`feof`** tests for end of file
- **`int feof(FILE *fptr)`**

- **`feof`** accepts a pointer to a **`FILE`**
- **`feof`** returns non-zero if **`EOF`** and zero otherwise

- Used when reading a file to check whether the **`EOF`** has been reached

Simple Accounts Problem

- Consider a simple accounting system to keep track of the amounts owed by a company's clients
- For each client the following information has to be kept
  - client account number
  - client name
  - client balance
- This information constitutes the "record" for the client which must be provided by the program

Processing as a Text File

- Consider that the file to hold the account information is a text file
- Need to use the formatted I/O functions to write and read from the file
  - **`fscanf`** and **`fprintf`**
- File operations that need to be implemented include
  - reading from the file
  - writing to the file - setting up the client records
  - searching for particular record(s) in the file
  - appending new clients to the existing master file
  - updating existing client records in the file

Writing to The Text File

- Writing a record to the file:

```
fprintf (fptr,"%d %s %.2f\n",
          accNo,name,balance);
```

- where the following variables have been declared and initialized appropriately

```
int   accNo;
char  name[21];
float balance;
```

- **`fprintf`** returns the number of characters written or negative if an error occurs

Example Writer

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

int main (int argc, char *argv[])
{
  FILE *fptr;
  int accNo = 1234;
  char names[][21] = { "Achison", "Agnew", "Barry", "Cunningham", "White" };
  float balance;
  int i;
  if ((fptr = fopen ("clients.dat", "w")) == NULL)
    {
      perror ("Error opening clients.dat");
      exit (1);
    }
  srand (time (NULL));
  for (i = 0; i < sizeof (names) / sizeof (names[0]); i++)
    {
      balance = (1000.0 * rand () / (RAND_MAX + 1.0)) - 500.0;
      /* random balance in range [-500,500[ */
      if (fprintf (fptr, "%d %s %.2f\n", accNo, names[i], balance) < 0)
        {
          perror ("Error writing data to clients.dat");
          exit (2);
        }
      accNo += 16;
    }
  if (fclose (fptr) != 0)
    {
      perror ("Error closing clients.dat");
      exit (3);
    }
  return 0;
}
```

Reading From The Text File

- Reading a record from the file:
- **`fscanf (fptr,"%d %20s %f",`**
  `&accNo,name, &balance);`

- where the following variables have been declared appropriately

```
int   accNo;
char  name[21];
float balance;
```

- **`fscanf`** returns the number of input items assigned or **`EOF`** if an error or **`EOF`** occurs before any characters are input
- **`rewind(FILE* ptr)`** - resets the current file position to the start of the file.

Searching The File

- Accessing a particular record(s) requires
  - a sequential read from the start of the file
  - testing each record to see is it being searched for
  - reset the file pointer to the start of the file ready for the next search

- Example:
  - Print the details of all the accounts with a negative balance or
  - Create a report of all accounts with a negative balance

Searching a Text File

```
while (1)
  {
    int args = fscanf (fptr, "%d %20s %f", &accNo, name, &balance);
    if (args == EOF)
      break;
    if (args != 3)
      {
        fprintf (stderr, "Error in clients.dat");
        exit (2);
      }
    if (balance < 0)
      printf ("%6d %30s %6.2f\n", accNo, name, balance);
  }
rewind (fptr);
```

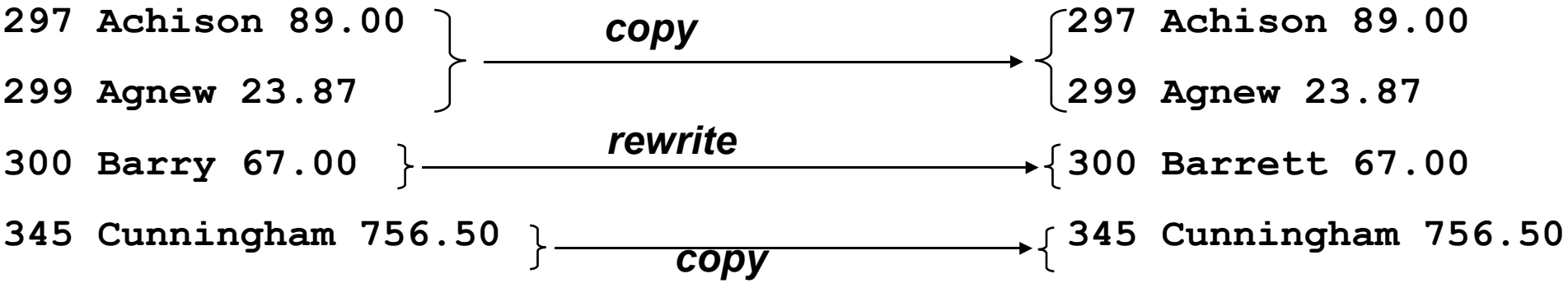
Updating Records in A Text File

- The text file contains variable length records
  - `300 Barry 67.00`
  - `345 Cunningham 756.50`
- A record cannot be modified in place as it may destroy other data in the file
- Consider changing Barry to Barrett
  - `300 Barry 67.00 345 Cunningham 756.50 ...`

  changes to
  - `300 Barrett 67.0045 Cunningham 756.50`

Updating Records in a Text File

- Records cannot be updated in place
- To update a text file the entire file is rewritten



Update Example

```
while (1)
  {
    char *tname;
    int args = fscanf (infile, "%d %20s %f", &accNo, name, &balance);
    if (args == EOF)
      break;
    if (args != 3)
      {
        fprintf (stderr, "Error in clients.dat");
        exit (3);
      }
    if (strcmp (name, "Barry"))
      tname = name;
    else                          /* replace Barry with Barrett */
      tname = "Barrett";
    if (fprintf (outfile, "%d %s %.2f\n", accNo, tname, balance) < 0)
      {
        perror ("Error writing data to clients.new");
        exit (4);
      }
  }
```

Positioning the File Pointer

- The function **`fseek`** sets the file position for a stream
- Subsequent reads and writes will access data in the file beginning at the new position
  - **`int fseek (FILE *fp, long offset,int origin);`**

- **`fseek`** takes 3 arguments
  - **`fp`** the file pointer for the file in question
  - **`offset`** is an offset in a file we want to seek to
  - **`origin`** is the position that the file pointer is set to

Positioning the File Pointer

- **`origin`** can be either
  - **`SEEK_SET`** - beginning of file
  - **`SEEK_CUR`** - current position in file
  - **`SEEK_END`** - end of file

- It may be necessary when performing a number of different types of access to a single file to reset the file pointer in certain instances
  - e.g. searching for a particular record in a file may leave the file pointer in the middle of the file
  - to write a new record to this file the pointer should be reset to point to the end of file

Positioning the File Pointer

- The function **`rewind`** resets the file pointer back to the start of file
  - **`void rewind (FILE *fp)`**
  - **`rewind(fp)`**
    is equivalent to
  - **`fseek(fp,0,SEEK_SET)`**

Binary Files

- Formatted Text files
  - contain variable length records
  - must be accessed sequentially, processing all records from the start of file to access a particular record
- Binary files
  - contain fixed length records
  - can be accessed directly, directly accessing the record that is required
- Binary files are appropriate for online transaction processing systems,
  - e.g. airline reservation, order processing, banking systems

Binary Files

- The main points about binary files:
  - Binary files are in binary format (machine readable) not a human readable format like text files
  - Binary files can be accessed directly (i.e. random access)
  - The record structure for a binary file is created using structures
  - They are more efficient than text files as conversion from ASCII to binary (reading) and vice versa for writing does not have to occur
  - They cannot be read easily by other non-C programs

Writing to a Binary File

- The function **`fwrite`** is used to write to a binary file
  - **`size_t fwrite (void *ptr, size_t size, size_t n, FILE *fptr);`**
  - **`fwrite`** writes from array **`ptr`**, **`n`** objects of size size to file pointed to by **`fptr`**
  - **`fwrite`** returns the number of objects written
    - which is less than **`n`** if an error occurs

Reading from a Binary File

- The function **`fread`** is used to read from a binary file
  - **`size_t fread (void *ptr, size_t size, size_t n, FILE *fptr);`**
  - **`fread`** reads **`n`** objects of size **`size`** from a file pointed to by **`fptr`**, and places them in array **`ptr`**
  - **`fread`** returns the number of objects read
    - which may be less than the number requested
    - call to **`feof()`** is necessary to distinguish **`EOF`** and error condition

Simple Account Problem

- Consider a simple accounting system to keep track of the amounts owed by a company's clients
- For each client the following information has to be kept
  - client account number
  - client name
  - client balance
- This information constitutes the "record" for the client which must be provided by the program

Processing as a Binary File

- Consider that the file to hold the account information is a binary file
- File operations that need to be implemented include
  - searching and reading specific records from the file
  - adding new customer details to the file
  - updating records on the file
  - reading the records sequentially from the file

Record Structure

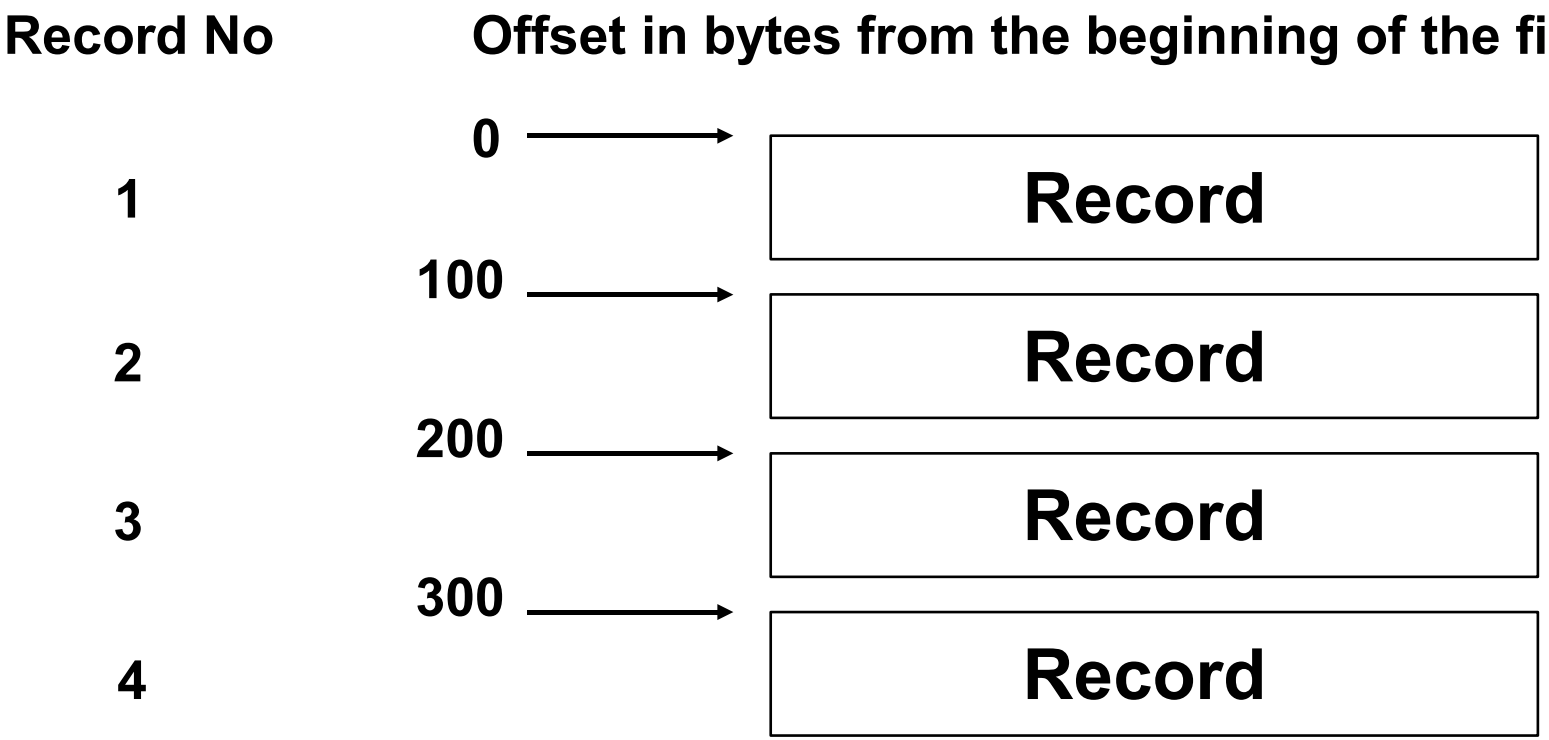
- Set up a user defined type that represents a record on the file

```
typedef struct {
char  name[21];
float balance;
} CustAccount;
```

- Note, that the account number is determined by a position in a file

Structure of The Accounts File

- Accessing a particular record in a binary file depends on accessing a fixed number of bytes at a particular location in the file. If a single record occupies 100 bytes, you get the following illustration:



Structure of The Accounts File

- With binary files:
- The records must be stored in ascending order

- The position of the record is determined by the key of the data in that record
  - e.g. record with **`accNo`** is stored at byte

    `(accNo-1) * sizeof(CustAccount)`

- The binary file must be set up initially - with all records blank

Creating the Binary File

- Open the file for writing

```
fp= fopen ("clients.dat", "wb");
```

- Set up a CustAccount variable that is initialised as a blank record

```
CustAccount cust={"",0.0};
```

- Loop around for n records writing a “blank” record each time

```
for (i=0;i<n;i++)
   fwrite (&cust,sizeof(CustAccount),1, fptr);
```

Positioning The File Pointer

- **`fseek`** is used to position the file pointer at the appropriate record in the file before reading or writing a record
  - **`int fseek (FILE *fp, long offset,int origin)`**
  - The position is set to the offset from the origin
  - **`fseek`** returns non zero on error
  - To position the file pointer at the record for accNo

```
fseek (fptr,(accNo-1)*sizeof(CustAccount),
  SEEK_SET)
```

Example Binary File Writer

```
for (i = 0; i < ACCOUNTS; i++)
  {
    if (fwrite (&cust, sizeof (CustAccount), 1, fptr) != 1)
      {
        perror ("Error during initialization of clients.dat");
        exit (2);
      }
  }
srand (time (NULL));
for (i = 0; i < sizeof (names) / sizeof (names[0]); i++)
  {
    cust.balance = (1000.0 * rand () / (RAND_MAX + 1.0)) - 500.0;
   /* random balance in range [-500,500[ */
    strcpy (cust.name, names[i]);
    if (fseek (fptr, (accNo - 1) * sizeof (CustAccount), SEEK_SET))
      {
        perror ("Error seeking in clients.dat");
        exit (3);
      }

    if (fwrite (&cust, sizeof (CustAccount), 1, fptr) != 1)
      {
        perror ("Error during initialization of clients.dat");
        exit (2);
      };

    accNo += 16;
  }
```

Example Search in A Binary File

```
printf ("Accounts with debit balance \n\n");
for (i = 0; i < ACCOUNTS; i++)
    {
      if (fread (&cust, sizeof (CustAccount), 1, fptr) != 1)
          {
            perror ("Error reading from clients.dat");
            exit (2);
          }
      if ((cust.balance < 0) && (cust.name[0]))
        printf ("%6d %30s %6.2f\n", i + 1, cust.name, cust.balance);
    }

rewind (fptr);

printf ("\n\nAccounts with credit balance \n\n");
for (i = 0; i < ACCOUNTS; i++)

  {
    if (fread (&cust, sizeof (CustAccount), 1, fptr) != 1)
        {
          perror ("Error reading from clients.dat");
          exit (2);
        }
    if ((cust.balance >= 0) && (cust.name[0]))
      printf ("%6d %30s %6.2f\n", i + 1, cust.name, cust.balance);
  }
```

Updating a Binary File

- To update a record in a binary file,
  - Read the record from the file
  - Change any of the details
  - Write the record back to the appropriate position in the file

```
for (i = 0; i < ACCOUNTS; i++)
  {
    if (fread (&cust, sizeof (CustAccount), 1, fptr) != 1)
      {
        perror ("Error reading from clients.dat");
        exit (2);
      }
    if (!strcmp (cust.name, "Barry"))
      {
        strcpy (cust.name, "Barrett");
        if (fseek (fptr, i * sizeof (CustAccount), SEEK_SET))
          {
            perror ("Error seeking in clients.dat");
            exit (3);
          }

        if (fwrite (&cust, sizeof (CustAccount), 1, fptr) != 1)
          {
            perror ("Error reading from clients.dat");
            exit (4);
          }
        break;
      }
  }
```